Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems

Tobias Blass*†, Arne Hamann*, Ralph Lange*, Dirk Ziegenbein*, Björn B. Brandenburg‡
*Robert Bosch GmbH
†Saarland University, Saarland Informatics Campus
‡Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus

***Abstract*—Robotic systems are typically subject to real-time constraints. Still, the ROS ecosystem—the most popular repository of open-source robotics software—exhibits little evidence of the use of real-time theory to bound or control worst-case response times. Hurdles to adoption are the amount of expertise required to correctly use real-time scheduling mechanisms and the inherent unpredictability of typical robotics workloads, which defy static provisioning. To overcome these hurdles, *ROS-Llama*, an *automatic latency manager* for ROS 2, is proposed. Crucially, use of *ROS-Llama* requires only little effort and knowledge of real-time concepts. Relevant properties of ROS 2 and essential requirements of the robotics domain are identified, and the conceptual and practical challenges in developing such a mostly automatic tool are discussed. Experiments on a mobile robot demonstrate the viability of the approach and show that ROS-Llama reduces the maximum observed latency under load compared to the default Linux scheduler. Finally, open problems in the underlying real-time analysis and major platform limitations in Linux and ROS 2 that prevent further improvements are identified.**

I. Introduction

In complex inter-disciplinary application domains such as robotics that require deep expert knowledge in diverse fields of study, writing all or even most software from scratch is usually not an option. Roboticists instead commonly embrace the integration of existing *third-party components* providing standard functionalities, which are readily available in popular robotics frameworks such as *ROS*. The advantages are numerous and easy to see. For instance, why painstakingly develop a new navigation subsystem if a complete navigation stack with multiple state-of-the-art path-planning algorithms and 3-D visualization support is just one download away?

To build a complete robotics system, many interacting components need to be integrated. Due to the distributed, open-source nature of the ROS development process, these components are usually developed in isolation by multiple independent *component developers* who do not necessarily know (of) each other. Similarly, the *system integrator*, who composes the selected components on a deployment platform with application- and mission-specific logic and "glue code," usually does not coordinate closely with the respective component developers.

To keep the integration of components as simple as possible, ROS employs the classic *topic-based publish-subscribe* paradigm to enable a loose coupling of components. Conceptually, each component can be understood as a "black box" containing a number of *callbacks* that subscribe to certain *topics*. Whenever a message is published pertaining to a given topic, all subscribing callbacks are invoked, perform some computation, and may then publish subsequent messages to other topics, which in turn triggers further callbacks, and so on. Integrators compose components by connecting the "input callbacks" of one component to the "output topics" of another. ROS systems thus form complex networks of interconnected topics and callbacks, where data (such as environmental stimuli) propagates along *cause-effect chains* through the network in an event-driven manner, transparently crossing component boundaries as desired by the integrator.

A typical example of such cause-effect chains is the sense-compute-actuate pipeline in a mobile robot that needs to detect and react to obstacles in its path. For instance, a hardware driver component may acquire a new sample from, say, a laser scanner (the *cause*), which then passes through multiple mapping, coordinate-transform, path-planning, and wheel-control components before ultimately resulting in a change in wheel velocity (the *effect*). Obviously, the maximum latency from cause until effect along such data-processing chains plays an important role for the correct functioning of a robot, and is often also crucial for safety considerations.

Importantly, to leave as many deployment choices as possible to the system integrator and thus maximize the opportunity for component reuse, the execution management layer of ROS and the underlying operating system are intentionally not exposed to component developers. Rather, ROS's central callback abstraction is simply a procedure with run-to-completion semantics, without any awareness of how or when callback procedures are scheduled, how the execution of callbacks is organized across threads or processes, or how the networking layer handles the sending and receiving of messages.

Since ROS is open-source software, it is, in principle, possible to gain a full understanding and control of a system's execution and communication behavior. From a real-time expert's point-of-view, it may thus seems to be a logical step to enrich ROS with well-known techniques from real-time systems research. However, there are several hurdles that make this more difficult than it appears at first glance.

First of all, the integrator lacks the required information. Most real-time analyses presume in-depth knowledge of many low-level system details such as the number of concurrent tasks, their activation semantics and functional interactions, arrival patterns of messages, worst-case execution times, *etc.* ROS components do not come with a manifest that would

provide this kind of information. To make matters worse, real-time analyses do not cope well with faulty or incomplete information. A single mistake or oversight while manually reverse-engineering a third-party component for modeling purposes could silently invalidate the entire effort.

Secondly, the required system details cannot be statically determined and described at the component level. One reason is that many robotics algorithms exhibit vastly varying execution times and activation patterns that depend on use-case- and platform-specific aspects. For example, consider a generic object-tracking component that identifies objects in a video stream and infers their trajectories (*e.g.*, cars in a neighboring lane). The execution time of this functionality depends on a variety of parameters, including the frame rate, resolution, and codec of the video stream, and various other parameters related to the specific tracking algorithm. None of these parameters can be known or fixed upfront by the developer of a generic object-tracking component. Such use-case specific information is only known to the integrator building a specific robot, who in turn is not necessarily an expert in object tracking or real-time systems and thus cannot always predict the impact of specific configuration choices. The resource demands and real-time behavior of a component must therefore always be evaluated in the context of its use in a specific deployment, which is not compatible with the modular reuse of "black box" components that underlies the popularity of the ROS framework.

Last but not least, even if the integrator were to discuss each component with the respective experts and would somehow obtain all details necessary for a timing analysis, a third fundamental problem remains: the resource requirements and performance characteristics of many components inherently depend on a robot's dynamic environment and thus vary over time, rendering a static (worst-case) resource provisioning infeasible.

For instance, consider the object-tracking component again, and suppose the robot also relies on a landmark-based self-localization component. On the one hand, the object tracker will demand much more processor time moving through a bustling city than through sparsely populated countryside. On the other hand, self-localization is likely much easier in a city with its many recognizable landmarks than in a mostly uniform landscape. To guarantee sufficient resources in both situations, the system integrator would have to provision the system for bustling cities consisting of barren countryside.

In robotics, such pessimistic system dimensioning is bound to quickly run into practical limitations. Instead, to remain practical and cost-efficient, robotics systems must be provisioned for the expected peak *joint* resource demands, rather than the sum of each component's individual peak demands.

Contributions. To overcome these challenges, we propose to use an *automatic latency manager* to provision ROS systems in a timing-aware manner dynamically at runtime. Specifically, this paper introduces the ***ROS L**ive **la**tency **ma**nager* (*ROS-Llama*), which enables the use of existing real-time mechanisms to control latencies along critical cause-effect chains in a way that both is *easy to use* for non-real-time experts and that *takes little effort to configure*. Rather than asking users to provide intricate system parameters, ROS-Llama relies solely on introspection and automatically estimates all required parameters at runtime to dynamically adjust scheduling parameters as the situation changes. If not all specified latency goals can be fulfilled at the same time (*e.g.*, due to a transient overload caused by adverse environmental conditions), ROS-Llama initiates a controlled *graceful degradation* process that allows the system integrator to specify the criticality of cause-effect chains in a *purely declarative manner* (*i.e.*, without needing to understand which components a cause-effect chain traverses).

In summary, this paper makes the following contributions.

- We explore the dynamic latency-management problem in the robotics domain and document constraints and requirements that a practical solution must satisfy (Sec. III);
- present the design and implementation of ROS-Llama, the first automatic latency manager for ROS (Sec. IV); and
- report on an evaluation demonstrating that ROS-Llama is capable of successfully controlling cause-effect chain latencies of a mobile robot using standard ROS components on a stock Linux system (Sec. VI).

ROS-Llama is the result of a multi-year research and engineering effort, during which we encountered a number of challenges and limitations in the state of the art. In Sec. VII, we highlight

- opportunities for analysis improvements that would render ROS-Llama more effective and accurate and
- major limitations in the ROS and Linux platforms that stand in the way of further improving the system.

II. Background and Definitions

We briefly summarize necessary background knowledge and the main analysis concepts underlying ROS-Llama.

a) ROS: The ROS framework is a popular open-source middleware and component repository for robotics applications [1]. This paper pertains specifically to ROS 2 [2], a recent major refactoring of the first-generation ROS framework. For brevity, we omit the version number throughout this paper.

ROS is a mature and featureful middleware usually deployed on Linux; we focus on the key runtime elements essential for latency management: *topics*, *callbacks*, and *executors*.

As already mentioned, ROS is built around a publish-subscribe infrastructure that is mostly oblivious to deployment choices. ROS applications can thus span multiple hosts, cores, processes, and threads. For the purpose of automatic latency management, the natural scope is a shared-memory multicore system running Linux, to which we restrict our attention herein.

We have already discussed topics and callbacks at the conceptual level in Sec. I. At the implementation level, a ROS application consists of one or more processes, each comprising one or more threads, which in turn run executors (*i.e.*, the ROS library facility that invokes callbacks). Each callback is associated with a specific executor. When a new message is posted to a topic, the ROS middleware ensures that a copy of the message is distributed to all threads hosting executors that serve subscribers to the topic. In a standard configuration, ROS relies on the *DDS* middleware to broker messages among executors;

there exist multiple suitable DDS implementations [3–5].

When an executor is notified that a new message is available, the corresponding callback is *activated* and queued for execution. The latency with which a callback processes a message is hence determined by two major factors: **(i)** how much processor time the OS allocates to the thread hosting the respective executor, and **(ii)** any queueing delay that arises from how the executor sequences pending callback activations.

For our purposes, aspect (i)—the scheduling of executor threads—is of primary interest, as it is the main factor that an automatic latency manager can control at runtime. Aspect (ii), the queueing delay, also has a major impact on callback latency and is by no means trivial to analyze [6], but (in current ROS versions) it is not amenable to runtime management.

b) Response-time analysis: We rely on prior work by Casini et al. [6] for (ii), *i.e.*, to bound the response times of callbacks managed by ROS's default executor. Casini et al. model a ROS system as a directed acyclic graph (DAG) of callbacks, connected by activation relations, and provide executor-aware response-time bounds for *processing chains*, *i.e.*, end-to-end latency bounds for arbitrary paths in the callback DAG.[1] Casini et al.'s processing chains directly correspond to the cause-effect chains that ROS-Llama manages.

c) Reservations: More precisely, Casini et al. provide a response-time bound under the assumption that a *supply bound function* (SBF) [7] is known for each thread. An SBF characterizes aspect (i) above, *i.e.*, how much processing time a thread is guaranteed to be allocated by the OS scheduler. This allows a schedulability analysis to analyze each thread as if it ran on an isolated core that provides $sbf(\Delta)$ units of processing time in any interval of length Δ. The standard way to obtain such an SBF guarantee is to use *reservation-based scheduling*, which in Linux is realized by the SCHED_DEADLINE policy [8].

The SCHED_DEADLINE scheduler implements the *Hard Constant Bandwidth Server (H-CBS)* [9] reservation scheme in conjunction with GRUB [10] bandwidth reclamation. While the specific scheduling rules are irrelevant for now, we note that each reservation r is characterized by a budget $budget(r)$ and a period $period(r)$; the scheduler guarantees that each reservation receives at least $budget(r)$ units of processor service in each period of length $period(r)$. The guarantee is sometimes more conveniently specified as the bandwidth $bw(r) = \frac{budget(r)}{period(r)}$.

d) Arrival curves: Finally, another key assumption in Casini et al.'s analysis [6] is that an *arrival curve* is known for each *ingress callback*, *i.e.*, for callbacks triggered by an external *event source*. An event source is not itself a callback, but sends messages to one or more topics (*e.g.*, a device driver obtaining and feeding sensor values into the callback DAG). An arrival curve $\eta_c(\Delta)$ bounds the maximum number of activations of a callback c over any given interval of length Δ. Given an arrival curve and a per-invocation *worst-case execution time* (WCET) e_c, it is trivial to determine a callback's *request-bound function* (RBF) as $rbf_c(\Delta) = e_c \cdot \eta_c(\Delta)$, which bounds the maximum cumulative processor demand of all activations of callback c in an interval of length Δ. The interplay of an executor's SBF and the RBFs of all its callbacks are at the core of Casini et al.'s analysis [6], which highlights the fact that considerable real-time expertise and comprehensive knowledge of a system's internals are required to apply their analysis.

[1] In practice, ROS applications are not always acyclic. We discuss how ROS-Llama avoids cycles in the extracted callback graph in Sec. V.

III. Automatic Latency Management

As argued in Sec. I, the robotics domain comes with specific requirements and constraints that have not received much attention in the real-time systems community to date. To document the challenging nature of the *automatic latency management problem* in robotics in general, and in a ROS context in particular, we highlight the most noteworthy aspects of ROS workloads that guided our design of ROS-Llama.

a) Form does not follow function: It is common in the classic real-time literature to assume that a system's functionalities and requirements are neatly reflected in its implementation as a set of executable tasks at the OS level. Correspondingly, central notions such as response time, priority, and criticality are usually associated with specific tasks, and hence the problem of ensuring correct timing for a given functionality is implicitly understood to be equivalent to the problem of properly provisioning the corresponding task.

As should be evident by now from Secs. I and II, this is far from the case in ROS: latency-critical functionality is rarely contained to a single component, cause-effect chains usually extend across many executors (and hence threads), and shared executors frequently serve multiple chains with vastly different latency needs and activation patterns. A latency manager must hence consider ROS systems holistically and cannot provision individual tasks, threads, or other OS entities in isolation.

b) Do no harm: ROS is popular because, empirically, it works well (enough) for many workloads. By default, ROS relies on Linux's best-effort CFS scheduler, which requires no configuration whatsoever. To state the obvious: active latency management should *not* result in *worse* compliance with latency goals than observed under CFS. This, however, is far less trivial than it sounds since a poorly configured real-time scheduler easily performs much worse than the default CFS scheduler. A latency manager should hence be self-aware and refrain from enacting configuration changes of uncertain benefit.

c) No exotic kernel patches: Robotics engineers are generally not willing to switch away from officially supported, "battle-tested" platforms just because of a promise of better real-time support. The gain in predictability rarely outweighs the lack of tooling, the difficulty in obtaining support, or the (perceived) lack of code maturity with its implied risks of rare bugs and untested corner cases. This rules out the use of bespoke patches augmenting a kernel's real-time capabilities. A practical solution is hence limited to the facilities found in a stock Linux kernel (and its widely used PREEMPT_RT variant).

d) No universal buy-in: As discussed in Sec. I, the ROS ecosystem is inherently heterogeneous, and development proceeds in an asynchronous, agile fashion, marked by frequent component updates. It is hence not realistic to expect all

(or even any) component developers to invest effort into supporting any particular latency management approach, nor is it reasonable to expect system integrators to fill in such support where it is lacking. In particular, this means a practical solution cannot rely on source-level annotations, presuppose the use of custom APIs, or change how ROS works in fundamental ways.

e) Ease of adoption: In a similar vein, a latency manager must minimize the upfront configuration and continuous maintenance burden incurred by system integrators. This is especially true given that the baseline choice—the default CFS scheduler—requires no setup at all. A system integrator usually has a high-level understanding of robot- and mission-specific end-to-end latency requirements but cannot reasonably be expected to know low-level system internals such as how the various ROS components interact precisely, how frequently they do so, how many executors there are, how callbacks are scheduled by the ROS executors, or how Linux's real-time scheduling facilities work in detail. To minimize the barrier to adoption, a practical latency manager should thus rely as much as possible on dynamic introspection rather than on upfront configuration (or costly static analysis) and favor configuration by means of declarative goals over mechanism-specific options.

f) Unpredictable environments: A dynamic, introspection-based approach is also advisable due to the inherently uncertain and shifting resource needs in dynamic environments, as already explained in Sec. I. Furthermore, latency goals may change as mission profiles evolve and robots adapt their behavior, which reinforces the need for a high-level, goal-oriented approach.

g) Nice-to-have payloads: Closely related to the prior point, it would be naïve and misguided to assume that a robot is actually equipped with sufficient compute resources to sustain all software functions in all situations. To the contrary, especially in mobile robots subject to space, weight, and power (SWaP) constraints, it is not uncommon to include "nice-to-have" functionalities that should work "most of the time," but which are not strictly essential and fully expected to operate at a degraded level (or not at all) when conditions become challenging (*e.g.*, mission- but not safety-critical payloads). A practical latency manager must be cognizant of such intentional under-provisioning of non-critical functionalities.

h) Unsurprising overload behavior: Conversely, it is not out of the ordinary for robots to experience periods of transient overload. By their very nature, such periods are the most challenging situations for an automatic latency manager and necessitate hard choices as not all latency goals can be satisfied simultaneously. A practical latency manager must not devolve to erratic decision making or otherwise unstable behavior under overload. Rather, it should avoid "surprises" by degrading the latency of cause-effect chains predictably and gracefully.

i) Earn your keep: Last but not least, it is worth emphasizing that every processor cycle spent on the latency manager is a cycle not spent on the workload, especially during periods of overload. Since, pragmatically speaking, latency issues under CFS can often be attenuated simply by making additional resources available, it is not a given that the presence of an active latency manager is actually beneficial in terms of latency goal compliance. In other words, a latency manager must yield sufficient benefits to compensate for the cost of running it in the first place. Implementation efficiency, as well as the runtimes of any employed analyses, are hence crucial.

IV. Design of ROS-Llama

Guided by the just-discussed observations and considerations, we designed ROS-Llama to operate largely automatically, following a *purely declarative* configuration approach. Specifically, in terms of necessary setup, ROS-Llama requires only a *latency goal* for each cause-effect chain the system integrator deems latency-sensitive (*i.e.*, in need of active latency management), and a *degradation order* among latency-sensitive cause-effect chains that is consulted in case of transient overload.

Latency goals are stated for cause-effect chains, which are identified solely by their start- and endpoints. For example, a user might specify that at most 200 ms may pass between the arrival of a new measurement at the laser scanner callback and the completion of the callback registering the detected obstacle in the map. Motivated by Req. (a) and Req. (e), it is ROS-Llama's responsibility to determine how these callbacks are connected, how frequently the chain is triggered, how much processor time each callback requires, and ultimately how the involved threads must be scheduled to achieve the latency goal.

Guided by Reqs. (f)–(h), ROS-Llama allows system integrators to configure a policy for controlled degradation, which is also defined in terms of processing chains. If ROS-Llama determines that it cannot guarantee all latency goals simultaneously, it degrades some chains to *best-effort mode*. The provided degradation policy determines in which order ROS-Llama will provision chains, which ensures predictable behavior under overload and allows system integrators to ensure that critical chains are never degraded to best-effort status in favor of lower-importance chains (Req. (h)).

To realize the configured goals, ROS-Llama must solve three main problems: **(1)** *extract* a model of the running ROS system (all topics, callbacks, executors, their resource needs, *etc.*), **(2)** *provision* all threads such that the configured latency goals are satisfied to the extent possible and *decide* if any chains need to be degraded to best-effort mode, and **(3)** *schedule* all threads in accordance to how they were provisioned in (2).

As illustrated in Fig. 1, ROS-Llama consists of a *model extractor* and a *budget manager* to address (1) and (2), respectively, and uses Linux's SCHED_DEADLINE scheduler for (3). To keep up with changing demands (Req. (f)), the model extractor continuously updates the model at runtime. Periodically, the budget manager takes a snapshot of this model and computes a new set of scheduling parameters, which are then enacted by SCHED_DEADLINE. In our case study, this was done every six seconds. Effectively, ROS-Llama provisions a dynamic ROS system by treating it as a series of static systems over time, always based on the latest model. We now discuss each part in turn, beginning with the model extractor.

A. The Model Extractor

The model extractor computes an empirical timing model of the running ROS system by observing its behavior over time.

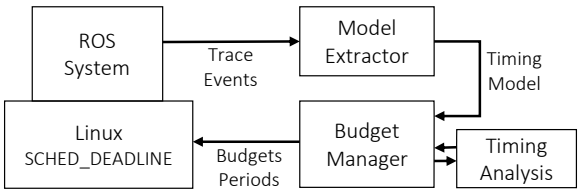

Fig. 1. Overview of ROS-Llama. The model extractor derives a timing model of the ROS system in a continuous manner and makes it available to the budget manager, which provisions the executor threads based on a response-time analysis of each cause-effect chain. Finally, the new budgets are enacted.

Specifically, the model extractor must identify all relevant threads in the system, recover the callback graph structure, and measure arrival- and execution times. To this end, the ROS-Llama model extractor transparently instruments the ROS core libraries `rclcpp` and `rcl`. Since these libraries are used by all (C++-based) ROS nodes, they can be instrumented locally without requiring any changes to third-party ROS components, which is essential to comply with Req. (d).

The instrumentation comprises static tracepoints that report when callbacks are registered, are invoked, publish, or complete. There are further tracepoints for thread identification, which report when a thread becomes an executor by starting the callback processing loop, and special-purpose tracepoints to monitor the use of certain APIs (*e.g.*, to detect "services," see below).

The model extractor incrementally constructs a callback-activation graph based on the stream of trace events. Each trace event comprises its type, the originating thread, a pair of timestamps, and additional event-specific data. The two timestamps measure time according to different clocks: the system-wide *monotonic clock* that indicates wall-clock time, and the per-thread *CPU-time clock* that tracks the amount of processor service received by the thread. The extractor uses the monotonic timestamp to infer arrival times and the CPU-time clock to measure execution times.

When a new callback is registered, the corresponding trace event reports the callback's type and its associated topic. The event record further contains an identifier derived from the virtual address of the callback object, which is used to uniquely identify the callback. These three properties — type, topic, and identifier — are determined at registration time and do not change during the lifetime of the callback.

Additional dynamic properties are derived and continuously updated from events that are emitted when a callback starts running, completes, or publishes. Each such trace event carries the unique callback identifier and, in the case of publication events, the topic being published to. Whenever a callback publishes, the extractor adds an edge between the callback and the reported topic if one does not exist yet. Whenever a callback completes, the extractor updates the callback's execution-time curve. Similarly, arrival curves are updated incrementally based on the occurrence of publication and start events.

In addition to observing executors, the model extractor also needs to identify all event sources, that is, threads that interact with the ROS system but are not executors themselves. Such threads are easily recognizable because they publish without ever starting a callback. A typical example is a driver that reads from a device file in a blocking I/O loop and relays each new sample encapsulated in a ROS message. ROS-Llama needs to manage such threads to control the latency of cause-effect chains that start with data acquisition.

It is worth mentioning that, in addition to callbacks, the ROS API also offers a notion of *services*, which realize call-return semantics: callbacks can invoke a service in a seemingly blocking manner and receive a reply in response. Under the hood, however, services are implemented as regular callbacks using a continuation-passing approach (clients invoke a service by posting a message to a special request topic and indicate on which topic they would like to receive the response). The model extractor can hence detect and trace services, and represent them in a way that is transparent to the rest of ROS-Llama.

The latest inferred callback graph and timing model is periodically provided to the budget manager, which it then uses to make up-to-date provisioning decisions.

B. Predictable Thread Scheduling

Before taking a detailed look at the budget manager, however, it is necessary to understand how provisioned threads are actually scheduled by ROS-Llama. Due to Req. (c), there are only two viable choices: either use a classic POSIX fixed-priority scheduler (*i.e.*, the SCHED_FIFO or SCHED_RR policies), or rely on Linux's more recent reservation-based SCHED_DEADLINE scheduler. We chose the latter.

Compared to the fixed-priority option, SCHED_DEADLINE provides two main advantages: analyzability and containment. Most importantly, its convenient analyzability stems from the fact that Casini et al.'s response-time analysis for ROS processing chains [6] directly applies to executor threads scheduled in resource reservations (recall Sec. II). By building on this analysis, ROS-Llama's budget manager can predict the effect of its provisioning decisions on the worst-case latency of the associated chains in a safe manner that avoids "missteps." That is, in accordance with Req. (b), it enables ROS-Llama to decide with confidence whether a cause-effect chain can be guaranteed, or else needs to be degraded to best-effort mode.

Containment means that a thread experiencing an unexpected surge in processor demand cannot prevent other threads from receiving their provisioned budget in a timely manner. This property is beneficial in case of transient overload or when resource needs increase due to changes in a robot's dynamic environment, and is hence well-aligned with Req. (f) and Req. (h). Any such surges in demand will be detected by the model extractor and reflected in the next timing model update, and hence are taken into account the next time budgets are recomputed.

On multicore platforms, Linux provides two ways to instantiate SCHED_DEADLINE: *global scheduling*, where the scheduler migrates threads freely among cores depending on current availability, and *partitioned scheduling*, where each thread is assigned to a specific core on which it remains even when other cores are idle. Prior work has shown that, empirically, partitioned scheduling achieves higher schedulability for most workloads (*i.e.*, it is much more effective at admitting

and guaranteeing reservations) [11]. Its effectiveness, though, depends heavily on the mapping from tasks to cores, which places an additional burden onto the user (especially in dynamic environments). Linux avoids this burden by defaulting to global scheduling. ROS-Llama, on the other hand, has sufficient information to determine a suitable mapping automatically and therefore uses partitioned scheduling without imposing any additional burden on the system integrator (recall Req. (e)).

Partitioned scheduling also allows ROS-Llama to conveniently isolate itself and miscellaneous system infrastructure such as various kernel and DDS middleware threads on a reserved *system core*. The remaining cores are made available to the budget manager for the provisioning of ROS threads.

Executors that serve only chains in best-effort mode are not provisioned by ROS-Llama and left to be scheduled by CFS instead. In particular, to comply with Req. (b), ROS-Llama never partially provisions a chain, letting degraded chains be served in a best-effort manner rather than risk providing a thread with insufficient budget. A degraded chain hence continues to operate without interruption and can still complete in time, but ROS-Llama cannot guarantee that it does.

C. The Budgeting Algorithm

Based on the extracted model, the budget manager must find a scheduler configuration—a budget and period for each reservation, and a feasible mapping of reservations to cores—that ensures the timely completion of the configured chains.

In the literature on reservation-based scheduling, a reservation's period is often derived from the underlying periodicity of the task [12–15]. This is not possible in our case since executors often influence multiple chains with different periods, as explained in Req. (a). Lacking a clear choice, ROS-Llama simply assigns all reservations a uniform period that is chosen to be significantly shorter than the tightest latency goal, but sufficiently long to avoid undue context-switching overheads. In our case study, the uniform reservation period was 10 ms.

How to assign a sufficient budget to each reservation is also far from obvious. In fact, reservation budgets cannot be chosen in isolation, but require solving a global co-optimization problem due to the interconnected nature of the callback graph. While we omit a formal description of the problem due to space constraints, the underlying intuition is easy to see. In a callback chain, the response-time bound of an "upstream" callback determines the activation jitter of any "downstream" callbacks. Changing the budget of one executor affects the response-time bounds of *all* callbacks that it handles and can thus induce changes in demand in any number of executors serving "downstream" callbacks. This in turn affects response-time bounds in those executors, at which point the propagation effect repeats. Worse, influence *cycles* are possible: though each callback chain is cycle-free, it is possible for two executors to be connected by multiple callback chains in opposite directions.

Arbitrary budget dependencies may thus exist among executors. Optimally solving such a complex optimization problem would be much too expensive at runtime, especially given Req. (i). ROS-Llama therefore attempts to find a (non-optimal) solution with an iterative heuristic search that is performed for each cause-effect chain in reverse degradation order.

Algorithm 1: The initial budget estimate.

```
1  for all influencing executors e without a budget do
2      needed ← Σ_{c served by e} rbf_c(horizon)
3      bw(e) ← needed / horizon
4  for any callback c with unbounded response times do
5      e ← executor serving c
6      needed ← rbf(c, horizon) − sbf_e(horizon)
7      if bw(e) = 1 then
8          degrade chain
9      else
10         bw(e) ← min(bw(e) + needed / horizon, 1)
11 if no partitioning for budget found then degrade chain
```

a) Finding a Starting Point: Algorithm 1 is used to find an initial budget assignment that reflects the maximum *longterm* processor demand of each executor. Since later steps will only add bandwidth, but never remove it, it is desirable to start with the least estimate that still ensures finite response-time bounds.

To estimate the longterm demand, first the cumulative demand over a long interval called the *horizon* is determined (Line 2). In our case study discussed in Sec. VI, we arbitrarily chose a horizon of 10 seconds, which exceeded all latency goals and typical busy-window lengths. To break the aforementioned dependency cycle—actual RBFs depend on the budget of other executors—Line 2 is computed under the simplifying assumption that all *other* executors receive 100% budget. Line 3 configures the resultant initial budget estimate.

Having assigned an initial budget, we now drop the assumption that other executors have 100% bandwidth. As a result, some callbacks likely become unschedulable due to increased jitter effects. To cope, Lines 4 to 10 iteratively increase the bandwidth of the corresponding executors until either the longterm demand at the horizon is met, or the executor would have to exceed 100% bandwidth, in which case it is impossible to guarantee bounded response times and the degradation process is started. The degradation process is also started in Line 11 if no feasible reservation-to-processor mapping can be found with common bin-packing heuristics. Following [11], ROS-Llama tries worst- and first-fit-decreasing, in that order.

b) Refining the Budget: Based on the initial budget assignment that (barely) achieves finite response-time bounds, Algorithm 2 refines executor budgets until the processing chain's response-time bound no longer exceeds the chain's configured latency goal. To this end, ROS-Llama relies on what we refer to as the *budget-shortage delay* heuristic $d(e)$, which is the total increase in response time attributable to executors having less than 100% bandwidth (Line 4). Here, $RT(c)$ denotes the actual current response-time bound [6], whereas $RT^{100\%}(c)$ denotes the response-time bound obtained by assuming 100% bandwidth. A large budget-shortage delay indicates that increasing the budget of this executor is likely to have large positive effects on response times in the system.

Following this heuristic, ROS-Llama considers the influencing executors in order of their shortage delay (Lines 5 to 10).

Algorithm 2: The assignment improvement heuristic.

```
1  res ← set of "upstream" executors that affect response times
2  while chain latency > goal do
3  |   for e in res do
4  |   |   d(e) ← Σ_{c served by e} (RT(c) − RT^100%(c))
5  |   for e in res by decreasing d(e) do
6  |   |   if bw(e) = 1 then remove e from res and continue
7  |   |   bw(e) ← min(bw(e) + 5%, 1)
8  |   |   if partitioning for budget found then
9  |   |   |   candidate found, break inner loop
10 |   |   remove e from res and restore old value of bw(e)
11 if no candidate found then degrade chain
```

For each executor, the algorithm tries to increase the bandwidth by a fixed step size (Line 7) until the chain's latency goal is reached. We chose 5% as a compromise between the speed of convergence and the quality of the result. If no candidate for a budget increase can be found while the latency goal remains unmet, the degradation process is started (Line 11).

V. Practical Considerations

While implementing ROS-Llama, we encountered several unanticipated challenges that required some extensions to its analytical foundations and special considerations in the model extractor and budget manager, which we sketch in following.

a) Activation cycles: The response-time analysis used by ROS-Llama [6] models the callbacks of a ROS system as an acyclic graph. This is an obvious choice for a topic-based pub-sub system like ROS, as a cycle in the graph would imply that some callback directly or indirectly activates itself, which one expects to result in an infinite cycle of activations.

To our surprise, we nonetheless encountered a cycle in the self-localization component AMCL (a part of the ROS navigation stack). This component publishes the robot's estimated position through the widely used TF coordinate transform library [16], a core ROS library for managing the various coordinate transformations encountereds in robotics applications. In short, the AMCL component not only publishes an estimated location of the robot, it also consumes odometry updates to produce its estimate—hence it both subscribes and publishes to the `/tf` topic, resulting in an apparent cycle.

Why does this not trigger an infinite loop? The answer lies in the *content* of the messages published on the `/tf` topic: AMCL's position update is triggered only by messages updating the odometry coordinate frame. Translations involving any other coordinate frames, like the position update published by AMCL itself, are ignored and can therefore not trigger a cycle.

As this construct effectively implements "topics within topics," it is hardly an idiomatic instance of ROS's design philosophy. Alas, it is widespread and hence, in the spirit of Req. (d), ROS-Llama needs to generally address such activation cycles without relying on case-by-case reasoning. To this end, we assume that developers guard against infinite loops, which implies that a callback publishing to its own topic activates only other subscribers to the topic, not itself. To cope, the model extractor introduces a *virtual cycle-breaker topic* in the extracted model: the cycle-inducing component is modeled as if it published to this virtual topic instead, which is given the same set of subscribers as the real topic except for the cycle-inducing component itself. This modeling tweak restores ROS-Llama's ability to apply Casini et al.'s response-time analysis [6] in our case study. More complex cycles would require more sophisticated model adjustments or manual intervention.

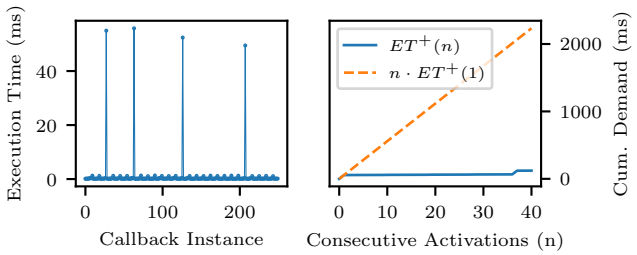

Fig. 2. Left: 250 observed per-invocation execution times of the `/tf` callback in the AMCL component. Right: Cumulative demand assumed by the analysis for n consecutive activations under different execution-time models.

b) Scalar WCETs are pessimistic: A common modeling choice in the real-time literature is to describe the maximum execution time of executable entities like callbacks or jobs with a *scalar* WCET parameter, meaning that the maximum cost per *single* activation is known, and the joint cost of n consecutive activations is extrapolated simply as the product of n and the scalar WCET.

In the context of ROS, this turned out to be prohibitively pessimistic. Fig. 2 shows observed execution times of the AMCL node's `/tf` callback, which as discussed above handles diverse messages. In the depicted trace, the maximum observed cost of a *single* activation was roughly 56 ms. However, it is apparent that the trace follows a pattern where such expensive activations are rare and far apart—any two peaks are separated by many "cheap" activations. Thus, assuming that *every* instance of the callback requires 56 ms is horrendously pessimistic.

To describe the execution-time requirements of such callbacks more accurately, ROS-Llama uses *cumulative execution-time curves* [17], a more expressive execution-time model that describes the *cumulative* execution-time requirements of multiple consecutive invocations. More precisely, an execution time curve $ET^+(n)$ bounds the maximum cumulative execution time of any n consecutive activations. The classic scalar WCET is hence equivalent to $ET^+(1)$. The resulting gain in precision can be seen in the right-hand inset of Fig. 2. The $ET^+(n)$ curve correctly represents that any *single* activation might take up to 56 ms, but also shows that the cumulative execution time of any 40 consecutive executions never exceeded $ET^+(40) = 122$ ms. For the same number activations, the scalar WCET model would predict a cost of $40 \times ET^+(1) = 2240$ ms.

We therefore modified Casini et al.'s response-time analysis [6] to use $rbf_c(\Delta) = ET^+(\eta_c(\Delta))$ as a callback's RBF.

c) Rare events: Extracted arrival curves can be highly imprecise for rare aperiodic events. For example, consider an interactive operator input that arrives twice in short succession after minutes of operation. A naïve arrival curve estimator would account *only* for the two observed events, pessimistically mischaracterizing them as following a strictly periodic arrival pattern with a short period. To incorporate that the event has

TABLE I
CONFIGURED PROCESSING CHAINS IN REVERSE DEGRADATION ORDER

Name	Purpose	Deg. Order	Length	Goal (ms)
heartbeat	Keep-alive signal	(last) 7	1	100
pilot	Navigation & control	6	2	125
odometry-nav	Odometry (navigation)	5	2	75
laser-scanner	Sensor data acquisition	4	2	150
localization	Self-localization	3	1	450
odometry-loc	Odometry (localization)	2	2	100
tracker	Track objects	(first) 1	2	990

been observed only twice since system startup, we additionally treat the system startup as a virtual activation of each callback. In the above example, this allows the model extractor to conclude that, although two events may occur in rapid succession, it takes much longer to observe three or more events.

VI. EVALUATION

We evaluated ROS-Llama on a Turtlebot 3 "Burger" controlled by a Raspberry Pi 4B. The Raspberry Pi features an ARM A72 CPU with four cores clocked at 600 MHz.[2] The system ran a standard Linux kernel with the PREEMPT_RT patch (version *4.19.71-rt24-raspi2*) hosting ROS 2 "Dashing Diademata" using Eclipse's Cyclone DDS (version 0.5.1-1).

a) ROS components: On top, we deployed three ROS components: **(1)** drivers for the Turtlebot, which provide a ROS interface to the robot's hardware (a laser scanner, odometers, and wheels powered by an electric motor), **(2)** the ROS navigation stack, and **(3)** an object tracker payload.

The ROS navigation stack [18] implements generic navigation primitives for wheeled mobile robots, including navigation planning, path following, and self-localization. There are various tuning parameters to adapt the components to the available computational resources, physical characteristics of the robot, and navigation precision demands. We used the default configuration provided with the Turtlebot, except for the period of the local planner, which we increased slightly from 100 ms to 125 ms as this proved sufficient for our scenario and induces less load. The object tracker [19] tracks designated objects through a video sequence and serves as a typical example of a computationally demanding mission- but not safety-critical payload. In our setup, we simulated a camera by repeatedly playing the *car1* video taken from the VOT 2018 challenge [20, 21]; in this scenario, the object tracker is tasked with following cars through the scene. To compensate for the large performance difference between the Raspberry Pi and the hardware recommended by the package developers (an Intel *i7-6700HQ* with four cores clocked at 2.6 Ghz), we downsampled the video accordingly.

b) Latency goals: We configured latency goals for the callback chains shown in Table I. The *heartbeat* chain simply manages a watchdog timer with a period of 100 ms that prevents the hardware from resetting. The first two functional chains are concerned with the movement of the robot's wheels. The *pilot* chain consists of a computation-intensive local planner callback responsible for computing the next motor command, followed by a shorter callback that encodes the command for transmission to the electric motor. The 125 ms latency goal ensures the motor receives the local planner's command once per period. The *odometry-nav* chain reports the measured wheel movements to the planner every 50 ms. We set a latency goal of 75 ms to ensure that an odometry update arrives every period, *i.e.*, that the gap between two measurements, including the sampling delay of up to 50 ms, remains below 125 ms.

The next three chains cover the self-localization of the robot. The localization component relies on laser scans and an internal map to narrow down plausible estimates of the robot's current location. Since the laser moves with the robot, interpreting these scans also requires information about the robot's movement and orientation, *i.e.*, odometry. The two inputs are provided by the *laser-scanner* and *odometry-loc* chains. The *localization* chain involves the merging of the inputs and the computation of a position estimate. We will revisit this chain in Sec. VII.

The localization estimate expires after one second, counting from the time the underlying laser scan was taken. This imposes a timing constraint on the localization component: once the localization estimate expires, the robot cannot navigate and performs an emergency stop. We thus need to arrange for an end-to-end latency of at most one second between the laser scanner and the final localization callback. The laser scanner rotates at 5 Hz, allowing it to produce one scan every 200 ms. In practice, we found that individual scans are occasionally transmitted incompletely by the hardware and cannot be interpreted. Accounting for such skipped scans yields a worst-case sampling delay of 400 ms, leaving 600 ms for processing. We assigned 150 ms to the *laser scanner* chain and 450 ms to the *localization* chain. For the odometry, we have to account for an additional 50 ms of sampling delay, leaving 100 ms for the *odometry-loc* chain.

Finally, the *tracker* chain covers the image tracker. The chain covers the (simulated) camera, which periodically acquires a frame (from disk) and sends it to the `/rgb` topic, and the tracker, which follows marked objects and outputs their position in the latest frame. The assigned latency goal ensures that every frame is processed before the next frame arrives, ensuring that the tracker does not fall behind under normal conditions. However, the *tracker* chain is also first in the degradation order, which reflects that its output is "nice to have" but not essential for the correct operation of the robot.

We stress that all latency goals derive purely from high-level functional considerations and hardware properties that would be known to a system integrator.

c) Baselines: We compare ROS-Llama against two baselines. The first is a standard Linux setup, where threads are scheduled globally across all four cores using CFS without any of the ROS-Llama infrastructure present. This is the default ROS setup and arguably the only other available choice that does not require real-time expertise on behalf of the system integrator and component developers. It provides a fair baseline with regard to Req. (i) (*i.e.*, the question of whether the cost

[2]The processor also supports a 1.2 GHz setting. However, this frequency cannot be sustained in continuous operation due to overheating, quickly leading to unpredictable thermal throttling of the cores. As dynamic frequency scaling is beyond the scope of this paper, we thus focus on the stable 600 MHz setting.

TABLE II
NUMBER OF GOAL VIOLATIONS PER CHAIN **(TODO: ADD COMMAS, REORDER COLUMNS)**

	fp		llama		cfs	
	violations	count	violations	count	violations	count
heartbeat	0	6908	0	7324	0	6907
pilot	2	4857	0	4826	3	4869
odometry-nav	0	13814	0	14648	0	13812
laser-scanner	15	3432	0	3636	0	3428
localization	0	3433	0	3636	0	3429
odometry-loc	1	13814	0	14648	0	13812
tracker	219	700	208	584	238	700

of running ROS-Llama outweighs its benefits) since it does not incur any of ROS-Llama's overhead.

The second is a primitive variant of ROS-Llama that does not use response-time analysis. Instead, it schedules latency-critical executors with the SCHED_RR fixed-priority scheduler. It aims for graceful degradation by assigning priorities according to the degradation order, prioritizing executor threads serving chains later in the degradation order over executors serving exclusively chains earlier in the order. This is arguably the most straightforward approach to implement controlled degradation without analysis, but still cumbersome and error-prone to realize manually[3] as it requires correct identification of all callbacks, chains, and threads. We use this baseline to evaluate to what extent the use of response-time analysis improves the decisions of ROS-Llama. We use SCHED_RR rather than SCHED_DEADLINE for this baseline, because without analysis, we have no way of assigning sensible reservation budgets.

d) Scenario: The robot patrols between two fixed locations while following a number of objects in the video. The experiment is divided into three phases: *no load*, *normal load*, and *high load*. In the first phase, the object tracker does not follow any objects. In the following phases, the number of objects to track increases to simulate the effects of an increasingly crowded environment. This increases the execution time of the tracker from a barely noticeable to an unsustainable load that forces video frames to be discarded. We observe the impact of this demand increase on the other chains.

A. Evaluation Results: Latency Goal Compliance

We evaluated how effective ROS-Llama is at meeting latency goals. Table II shows the number of observed chain completions with end-to-end latency exceeding the configured goal. The *tracker* chain exceeds its bound frequently (between 208 and 238 times out of about 600–700). In the case of ROS-Llama, the *tracker* chain is even forced to skip entire periods, resulting in fewer chain instances during the experiments. This indicates that the camera simulation does not receive enough processor time to transmit a frame during some of the periods. Such effects are to be expected as a result of the unsustainable load in the last phase. Controlled degradation should ensure that this overload does not affect the other chains. Still, under both baselines other chains exceed their goal latency, namely the *pilot* chain under both CFS and SCHED_RR, and additionally

[3]That is, without a tool providing automatic introspection capabilities such as those provided by ROS-Llama's automatic model extractor.

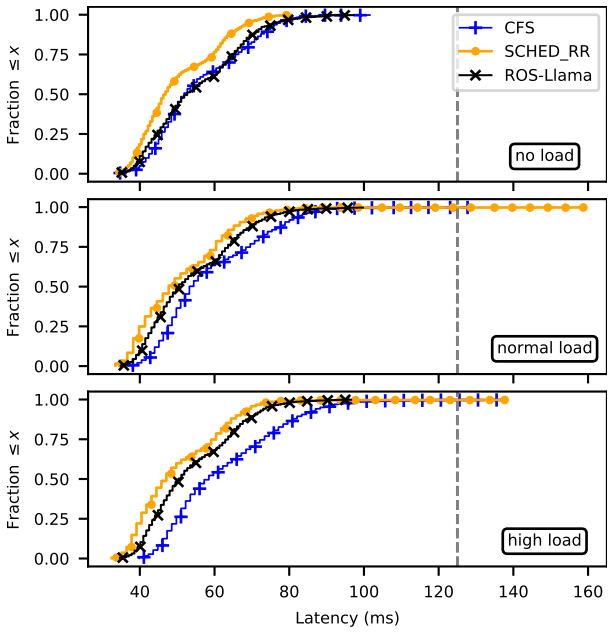

Fig. 3. CDFs of the latency of the *pilot* chains, separated by phase. ROS-Llama keeps the tail latency below the 125 ms goal (dashed line), even under heavy load. Both baselines exhibit tail latencies in excess of the goal.

the *odometry-loc* chain under SCHED_RR. In contrast, ROS-Llama successfully protects the more critical chains from undue interference by the *tracker* chain, degrading the system gracefully in face of the surge.

To understand the reasons for the observed latency goal violations, we investigated the affected chains in more detail. Figure 3 shows a CDF of the observed end-to-end latencies in the *pilot* chain, separated by phase. The dashed vertical line marks the latency goal: the goal is always met if a curve's points are all to the left of this line (*i.e.*, if the observed maximum end-to-end latency does not exceed the chain's latency goal).

In the first phase, all three approaches ensure a margin of at least 25 ms to the latency goal. As the load increases, CFS's curve grows wider, indicating that high-latency results become more prevalent. While only 4% of activations exceed 80 ms in the first phase, 10% do under normal load, and 16% under high load. It is also evident how the tail grows longer, finally exceeding the latency goal.

These observations demonstrate the risk posed by CFS's complete lack of temporal isolation: the non-essential *tracker* chain is *functionally* completely unrelated to the critical *pilot* chain, but still the overload experienced in *tracker* chain heavily impacts the observed end-to-end latency of the critical chain, increasing it by over 25 ms in the most extreme cases.

The root cause is that both chains contain computationally intensive callbacks, and hence both are entitled to an equal share of the available resources under the default CFS timesharing policy. Oblivious to their respective latency requirements and their relative importance to the robot's overall correct operation, CFS has no way of inferring which of the two components it should prioritize and, as a result, *both* chains exhibit latency goal violations. Fair sharing of resources, the core principle underlying CFS, is obviously and demonstrably

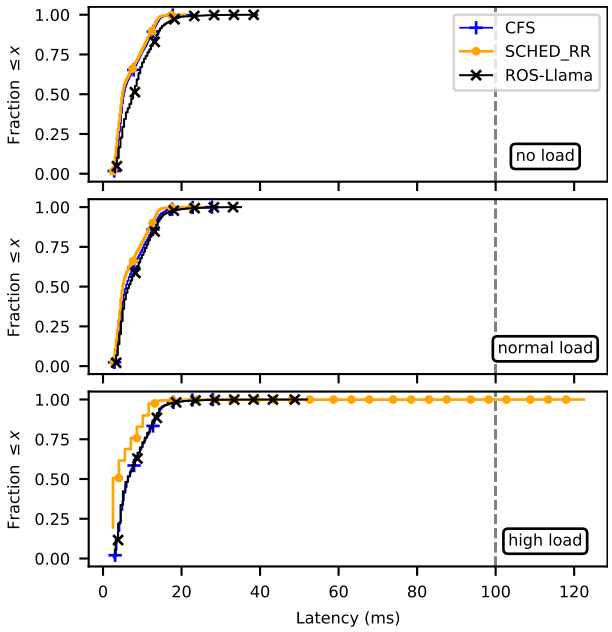

Fig. 4. CDFs of the latency of the *odometry-loc* chains, separated by phase. ROS-Llama and the CFS baseline comply with the 100 ms latency goal, whereas the SCHED_RR baseline exhibits a large tail-latency spike.

not the appropriate policy under transient overload conditions.

As one might hope, the SCHED_RR baseline keeps latency more stable *most* of the time. However, we also observed a few large outliers in the *pilot* chain that exceeded the latency goal by at least 15 ms, which is surprising given that, according to the degradation order, the *pilot* chain clearly should have higher priority than the overloaded *tracker* chain (the two chains do not share any executors). In fact, the effect here is subtle and can be understood as a kind of "collateral damage": because the *tracker* chain is overloaded *at real-time priority*, it induces a bursty overload also into the DDS layer, which in turn negatively affects the intra-chain communication delays of the *pilot* chain. This shows that graceful degradation is essential—ROS-Llama actively degrades the overloaded *tracker* chain to best-effort mode, which implicitly prevents the DDS overload. We revisit the DDS issue in Sec. VII-A.

We observed an even more severe latency spike under the SCHED_RR baseline in the *odometry-loc* chain, with the high-load tail latency exceeding the maximum observed latency in the no-load scenario by almost 100 ms, as shown in Fig. 4. The reason here is that the *odometry-loc* chain is early in the degradation order, and the executors serving its callbacks are thus assigned a low scheduling priority. It is therefore at risk of being starved by less urgent chains that happen to occur later in the degradation order, which is a typical problem when assigning priorities in a *criticality-monotonic* fashion rather than according to urgency [22]. Of course, *not* assigning criticality-monotonic priorities would risk (even more) unpredictable degradation behavior, and hence is also not a viable approach.

The complex interaction of the various third-party components and the sheer number of moving parts makes it difficult to draw more detailed conclusions. Our case study illustrates a clear trend, though: Neither avoiding real-time scheduling altogether (the default CFS baseline) nor assigning real-time priorities in a "blind", purely heuristic-driven manner without a backing analysis (the SCHED_RR baseline) leads to satisfactory results. ROS-Llama in contrast exploits the existing capabilities offered by Linux's SCHED_DEADLINE scheduler *without* burdening the system integrator with any real-time scheduling details by provisioning the system dynamically, based on introspection and guided by response-time analysis.

TABLE III
AVERAGE ROS-LLAMA OVERHEAD BY COMPONENT PER PHASE **(TODO: ADD PHANTOM/MATHLLAP TO ALIGN PARENS)**

Phase	No load	Medium load	High load
Model Extractor (Go)	1.17 (70%)	1.38 (62%)	1.39 (61%)
Model Processing (Python)	0.28 (17%)	0.32 (14%)	0.32 (14%)
Budget Selection (Python)	0.05 (3%)	0.12 (5%)	0.13 (6%)
Timing Analysis (Rust)	0.18 (11%)	0.41 (19%)	0.43 (19%)
Total	1.68 (100%)	2.23 (100%)	2.27 (100%)

B. Evaluation Results: ROS-Llama Runtime Costs

We conclude the evaluation with an investigation of the costs involved in running ROS-Llama. ROS-Llama's memory footprint is negligible relative to the footprint of ROS; we hence focus on processor time. Table III reports the average per-invocation cost of the main parts of ROS-Llama. Recall that we configured ROS-Llama to recompute the budget every six seconds; in total, ROS-Llama as a whole thus consumes 30–40% of one CPU. The lower analysis- and budgeting overhead in the first phase results from ROS-Llama opportunistically reusing cached budget assignments in a low-load scenario. The model extractor, which runs continuously alongside the system, accounts for about 60% of the total cost of ROS-Llama. The cost of preparing the timing model for analysis causes about 15% of the overhead. The remainder of the runtime costs are due to the budget selection process, separated into the budgeting heuristics (≈5%) and the response-time analysis (≈20%).

The results show that ROS-Llama introduces a noticeable overhead. However, despite this overhead, ROS-Llama ensures better latency goal compliance and more graceful degradation behavior than the CFS baseline. In other words, ROS-Llama satisfies Req. (i): it comes with significant costs, but provides sufficient benefits to realize a favorable trade-off between performance and predictability.

In the current prototype, the model extractor causes most of the overhead due to our unoptimized tracing implementation. Integrating ROS-Llama with a mature tracing system like LTTng [23] would likely lower the cost of running ROS-Llama.

VII. Open Problems and Pragmatic Workarounds

In developing ROS-Llama, we have become aware of a number of opportunities for future work as well as limitations in ROS and Linux that were not obvious to us initially.

A. Open Analysis Problems

ROS-Llama stands to benefit from improvements in the analytical foundations upon which it rests along several directions.

a) Complex activations: In addition to regular callbacks, which are unconditionally activated upon publication of a message, ROS also provides advanced activation semantics in the form of *message filters*. Of particular interest to us are the TF-related message filters, which are used in the navigation stack. Conceptually, filters *select* and *combine* multiple incoming data items (from separate messages) into a single message for *joint* downstream processing based on complex rules. In terms of expressiveness, they go far beyond classic "and" activation semantics, and may in fact also depend on the *contents* of to-be-combined messages.

For instance, in the navigation stack, the localization component uses message filters to ensure that the laser scanner callback is only triggered once an odometry measurement is available that is no older than the latest available laser scan sample. Such complex activation semantics cannot be represented with current modeling approaches. In our case study, we circumvented this problem by declaring the parts before and after the message filter as separate chains (*i.e.*, the *laser-scanner*, *odometry-loc*, and *localization* chains), and by manually distributing the latency goal among the chains before and after the message filter. With a more expressive model, such manual configuration tuning could be avoided, benefitting both ROS-Llama's usability and efficiency.

b) In want of stochasticity: As discussed in Sec. V, the use of execution-time curves (rather than a scalar WCET model) is essential to obtaining useful analysis accuracy. Nonetheless, in our experiments, we sometimes still noticed large gaps between observed latencies and predicted bounds.

The reason is obvious: an execution-time curve, while less pessimistic, is still a model of *worst-case* demand, but in practice ROS systems are exceedingly unlikely to consistently exhibit worst-case behavior. Conceptually, a *stochastic* timing analysis might thus be a much better match for ROS systems, which are virtually always deployed on commodity platforms and certainly not amenable to static WCET analysis.

Case in point, given that ROS-Llama follows a dynamic, introspection-based approach anyway, we would gladly trade some analysis certainty for much tighter response-time bounds—for instance, with a hypothetical analysis rooted in the concept of a traced "probabilistic worst-case execution-time curve" $pWCET(n)$, in analogy to the probabilistic WCET concept at the center of much recent attention [24–30]. Given the inherent uncertainty in ROS systems and their dynamic environments, we believe there to be much promise in this direction.

c) DDS: The DDS middleware heavily affects the transmission delay between ROS components, as all communication must pass through it. To tune this delay, DDS provides numerous QoS options; the documentation of one implementation [31] lists alone 53 parameters affecting, among other things, the number of threads created, the scheduling priority of several DDS support threads, or the message transmission order. While there is prior work on analyzing implementation-independent QoS options [32, 33], the scheduling of DDS threads is intentionally left aside by the DDS standard as "implementation-defined" and remains little-understood to date. To our knowledge, there is as of now no principled way to predict the impact of scheduling decisions on DDS latency.

In this work, we therefore treated the DDS infrastructure as a black box and reduced its latency impact by isolating all DDS threads on the separate system core at a high scheduling priority. Future work illuminating the effects of thread scheduling on DDS latency might enable ROS-Llama to automatically select suitable DDS QoS options.

B. Linux Platform Limitations

ROS-Llama benefits from Linux's real-time capabilities, and particularly from SCHED_DEADLINE [8], to a great extent. Nonetheless, certain issues also posed surprising challenges.

a) High-latency I/O: In our experiments, we found that laser-scanner and odometry data would sometimes arrive at the Turtlebot driver threads only after excessive delays. It turned out that data arriving on USB serial ports traverses the TTY layer, which involves CFS-scheduled kernel threads (even in a PREEMPT_RT kernel) that are easily starved by real-time processes. Although we were ultimately able to sidestep this problem by forcing Linux's "unbound *kworker* threads" onto the system processor, it serves as a reminder that real-time I/O remains a frequently overlooked and understudied problem.

b) Scheduler inversion: SCHED_DEADLINE threads *always* take priority over SCHED_FIFO and SCHED_RR threads, which simplifies the analysis but also causes many practical issues. Although the assigned reservation bandwidths limit this delay somewhat, it can still be substantial (*i.e.*, the maximum EDF busy-window length). This design is particularly unfortunate since many system-critical kernel threads (*e.g.*, disk drivers) are scheduled with SCHED_FIFO or SCHED_RR priorities. Assigning "too much" bandwidth to reservations can thus starve critical kernel threads and actually lead to kernel panics. A principled solution might be to introduce "scheduling-class reservations" that explicitly reserve processor time for SCHED_FIFO, SCHED_RR, and CFS in the SCHED_DEADLINE schedule.

c) Threads vs. reservations: SCHED_DEADLINE ties reservation parameters to individual threads. It is thus not possible to share a budget among multiple threads, making it exceedingly inefficient to apply reservations to multi-threaded applications that distribute work dynamically among threads (*e.g.*, virtually all DDS middlewares). Popular libraries for asynchronous programming like *boost::asio* or the C++ *std::async* API are also impossible to provision. First-class reservations supporting multiple client threads would be a relief.

d) Soft reservations needed: Mainline Linux presently supports only *hard* reservations, which unconditionally cut off a thread that exhausts its reservation's budget from processor service until the next replenishment time [34]. This rate-limiting behavior can be highly problematic: *underestimating* the required budget even by a minuscule amount results in a massive latency *increase*, since once a thread's under-dimensioned budget runs out, it must wait *even if it could complete if it were a* CFS *thread.* In contrast, *soft* reservations [34] would allow under-provisioned threads to receive at least *some* guaranteed

bandwidth and then progress on a best-effort basis, which would allow ROS-Llama to partially provision degraded chains.

C. ROS Idiosyncrasies

Although ROS-Llama seeks to accommodate the existing ROS ecosystem as much as possible, it cannot work miracles if the managed ROS components are already susceptible to latency spikes even in the absence of interference. Case in point, consider the TF system again, which internally queues any messages it cannot process immediately. *All* such queued messages are reconsidered when a new translation appears on the TF topic—to the effect that a single `/tf` callback instance may *non-preemptively* process a large backlog of messages in one fell swoop, which inevitably induces latency spikes in other callbacks served by the same executor.

Another noteworthy implementation choice is a domain-specific language in the navigation stack. Specifically, the navigator's strategy is described using XML *behavior trees*, which are queried for changes in intended behavior in a polling manner by a C++-based interpreter embedded in a timer callback that is triggered every 10 ms. Such constructs obfuscate the system's true communication structure and timing behavior, and thus impede the model extractor's efforts. We resolved this problem in our case study by rewriting the behavior tree as a normal, message-triggered C++ callback.

Overall, ROS-Llama shows that automatic latency management for robotics is practical and has great potential, but these discoveries also illustrate that there are limits to the degree of automation that can be realistically attained if developers do not favor principled, *idiomatic* solutions whenever possible.

VIII. Related Work

The real-time needs of robotics workloads have long been a subject of intense study, with classic papers in this area defining suitable APIs and runtime systems; better-known examples are HARTIK [35, 36], which introduced a similar approach to controlled degradation as used in ROS-Llama, ETHNOS [37, 38], and XO/2 [39]. Our focus is the nowadays popular ROS, which we cannot alter due to Req. (d).

The general ideas of adaptation and graceful degradation in the face of transient overload have also been explored from many angles in prior work. A substantially different, very well-explored approach is the concept of *service levels* [15, 40–45], which relies on application developers to explicitly expose different operating modes and a notion of utility associated with each mode. Compared to ROS-Llama's much more narrow but simpler mechanism, such approaches are more difficult to adopt due to the required developer buy-in. ROS components generally do not expose multiple operating modes.

Another common adaptation strategy is to rely on feedback-control theory to adjust scheduling parameters. Prior work has explored approaches to directly control periods [46, 47], QoS-levels [42, 45, 46, 48], and reservation budgets [43, 49–54], to name a few. Such approaches could in principle be transferred to ROS, but have not yet been systematically studied in that context. ROS-Llama differs from these prior approaches in that it realizes *predictive* provisioning guided by response-time analysis [6] based on an explicit extracted model of the workload, rather than feedback-guided *reactive* provisioning.

While there is earlier work on the real-time capabilities of ROS 1 [56, 57], it is unclear to what extent these findings still apply since improved real-time support is a key differentiator of ROS 2. Park *et al.* [55] evaluated the predictability of ROS 2 in comparison to ROS 1 and found ROS 2's improved timing precision to result in better path-following precision.

Finally, the problem of finding optimal budgets and periods for resource reservations has been studied extensively, for both individual tasks [12–15, 58–60] and DAGs [61]. However, these approaches are not directly applicable to ROS due to the complex interdependencies in the ROS callback graph (recall Sec. IV-C). Nonetheless, future work adapting and extending such approaches to cope with ROS-specific challenges could supplant ROS-Llama's current budgeting heuristic.

In summary, ROS-Llama builds on well-established techniques and ideas that have been studied individually in prior contexts. The primary contributions of ROS-Llama are the integration of these concepts in a practical, easy-to-use system and their adaptation to overcome ROS-specific challenges. ROS-Llama is the first automatic latency manager that copes with the complexities posed by real-world robotics frameworks such as ROS. Its distinguishing feature is that can be applied to *existing* ROS applications running on unmodified stock Linux kernels hosted on commodity platforms (recall Sec. III).

IX. Conclusion

We have presented ROS-Llama (Sec. IV), the first automatic latency manager for ROS 2. Its design has been shaped by a careful analysis of the requirements and constraints of the ROS ecosystem (Sec. III). Our evaluation has shown ROS-Llama to be practical and beneficial, achieving better latency control under load than either the default CFS baseline or a criticality-monotonic SCHED_RR baseline. Finally, our work on ROS-Llama has given us a better appreciation of a number of open analysis questions of practical relevance and current limitations in ROS and Linux, which we have shared in Sec. VII in the interest furthering advances in this promising area.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[2] "ROS 2," https://github.com/ros2.

[3] "Eclipse Cyclone DDS," https://projects.eclipse.org/projects/iot.cyclonedds.

[4] "eProsima FastRTPS," https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps.

[5] "RTI Connext DDS," https://www.rti.com/products/connext-dds-professional.

[6] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[7] A. Mok, X. Feng, and D. Chen, "Resource Partition for Real-Time Systems," in *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2001.

[8] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An

EDF scheduling class for the Linux kernel," in *Proceedings of the 11th Real-Time Linux Workshop*, 2009.

[9] A. Biondi, A. Melani, and M. Bertogna, "Hard Constant Bandwidth Server: Comprehensive formulation and critical scenarios," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2014.

[10] G. Lipari and S. Baruah, "Greedy Reclamation of Unused Bandwidth in Constant-Bandwidth Servers," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

[11] B. B. Brandenburg and M. Gül, "Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations," in *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[12] G. Buttazzo and E. Bini, "Optimal Dimensioning of a Constant Bandwidth Server," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, 2006.

[13] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based Real-Time Scheduling in the Linux Kernel," *ACM SIGBED Review*, 2019.

[14] L. Palopoli and L. Abeni, "Legacy Real-Time Applications in a Reservation-Based System," *IEEE Transactions on Industrial Informatics*, 2009.

[15] S. Groesbrink, L. Almeida, M. de Sousa, and S. M. Petters, "Towards Certifiable Adaptive Reservations for Hypervisor-Based Virtualization," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[16] T. Foote, "Tf: The Transform Library," in *Proceedings of the 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, 2013.

[17] S. Quinton, M. Hanke, and R. Ernst, "Formal Analysis of Sporadic Overload in Real-Time Systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.

[18] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A Navigation System," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[19] "Ros2_object_analytics," https://github.com/intel/ros2_object_analytics.

[20] M. Kristan, A. Leonardis, J. Matas, M. Felsberg, R. Pflugfelder, L. Č. Zajc, T. Vojir, G. Bhat, A. Lukezic, A. Eldesokey, and G. Fernandez, "The sixth visual object tracking VOT2018 challenge results," in *VOT2018 Workshop*, 2018.

[21] M. Kristan, J. Matas, A. Leonardis, T. Vojir, R. Pflugfelder, G. Fernandez, G. Nebehay, F. Porikli, and L. Cehovin, "A Novel Performance Evaluation Methodology for Single-Target Trackers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.

[22] S. Baruah, A. Burns, and R. Davis, "Response-Time Analysis for Mixed Criticality Systems," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[23] M. Desnoyers and M. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Linux Symposium*, 2006.

[24] K. P. Silva, L. F. Arcaro, and R. S. De Oliveira, "On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[25] J. Abella, D. Hardy, I. Puaut, E. Quinones, and F. J. Cazorla, "On the Comparison of Deterministic and Probabilistic WCET Estimation Techniques," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[26] K. P. Silva, L. F. Arcaro, D. B. de Oliveira, and R. S. de Oliveira, "An Empirical Study on the Adequacy of MBPTA for Tasks Executed on a Complex Computer Architecture with Linux," in *Proceedings of the 23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

[27] L. Santinelli, F. Guet, and J. Morio, "Revising Measurement-Based Probabilistic Timing Analysis," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[28] N. Gobillot, F. Guet, D. Doose, C. Grand, C. Lesire, and L. Santinelli, "Measurement-Based Real-Time Analysis of Robotic Software Architectures," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[29] R. I. Davis and L. Cucu-Grosjean, "A Survey of Probabilistic Schedulability Analysis Techniques for Real-Time Systems," *LITES: Leibniz Transactions on Embedded Systems*, 2019.

[30] ——, "A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems," *LITES: Leibniz Transactions on Embedded Systems*, 2019.

[31] "RTI Connext DDS – Comprehensive Summary of QoS Policies," https://community.rti.com/static/documentation/connext-dds/5.2.0/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_QoS_Reference_Guide.pdf.

[32] H. Pérez and J. J. Gutiérrez, "Modeling the QoS parameters of DDS for event-driven real-time applications," *Journal of Systems and Software*, 2015.

[33] A. Hakiri, "Supporting end-to-end quality of service properties in OMG data distribution service publish/subscribe middleware over wide area networks," *Journal of Systems and Software*, 2013.

[34] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," in *Multimedia Computing and Networking*, K. Jeffay, D. D. Kandlur, and T. Roscoe, Eds., 1998.

[35] G. C. Buttazzo, "HARTIK: A Real-Time Kernel for Robotics Applications," in *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS)*, 1993.

[36] G. Buttazzo, "Real-time Issues in Advanced Robotics Applications," in *Proceedings of the 8th Euromicro Workshop on Real-Time Systems (EMWRTS)*, 1996.

[37] M. Piaggio, A. Sgorbissa, and R. Zaccaria, "A Programming Environment for Real-Time Control of Distributed Multiple Robotic Systems," *Advanced Robotics*, 2000.

[38] M. Piaggio and R. Zaccaria, "Distributing a Robotic System on a Network: The ETHNOS Approach," *Advanced Robotics*, 1996.

[39] R. Brega, N. Tomatis, and K. Arras, "The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2000.

[40] H. Hassan, J. Simó, and A. Crespo, "Flexible Real-Time Mobile Robotic Architecture based on Behavioural Models," *Engineering Applications of Artificial Intelligence*, 2001.

[41] G. Beccari, S. Caselli, and F. Zanichelli, "A Technique for Adaptive Scheduling of Soft Real-Time Tasks," *Real-Time Systems*, 2005.

[42] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint, "An Adaptive Framework for Multiprocessor Real-Time Systems," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[43] L. Abeni and G. Buttazzo, "Adaptive Bandwidth Reservation for Multimedia Computing," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999.

[44] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley, "Performance Specifications and Metrics for Adaptive Real-Time Systems," in *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS)*, 2000.

[45] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms," *Real-Time Systems*, 2002.

[46] A. Cervin and J. Eker, "Feedback Scheduling of Control Tasks," in *Proceedings of the 39th IEEE Conference on Decision and Control*, 2000.
[47] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback–Feedforward Scheduling of Control Tasks," *Real-Time Systems*, 2002.
[48] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, "The Case for Feedback Control Real-Time Scheduling," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS)*, 1999.
[49] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a Reservation-Based Feedback Scheduler," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.
[50] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA—Adaptive Quality of Service Architecture," *Software: Practice and Experience*, 2009.
[51] L. Palopoli, L. Abeni, and G. Lipari, "On the Application of Hybrid Control to CPU Reservations," in *International Workshop on Hybrid Systems: Computation and Control*, 2003.
[52] E. Eide, T. Stack, J. Regehr, and J. Lepreau, "Dynamic CPU Management for Real-Time, Middleware-Based Systems," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.
[53] N. Shankaran, X. D. Koutsoukos, D. C. Schmidt, Y. Xue, and C. Lu, "Hierarchical Control of Multiple Resources in Distributed Real-Time and Embedded Systems," *Real-Time Systems*, 2008.
[54] D. Fontanelli, L. Palopoli, and L. Greco, "Deterministic and Stochastic QoS Provision for Real-Time Control Systems," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.
[55] J. Park, R. Delgado, and B. W. Choi, "Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study," *IEEE Access*, 2020.
[56] Y. Saito, T. Azumi, S. Kato, and N. Nishio, "Priority and Synchronization Support for ROS," in *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016.
[57] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio, "Real-Time ROS Extension on Transparent CPU/GPU Coordination Mechanism," in *Proceedings of the 21st IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2018.
[58] G. Lipari and E. Bini, "Resource Partitioning among Real-Time Applications," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.
[59] ——, "A Methodology for Designing Hierarchical Scheduling Systems," *Journal of Embedded Computing*, 2005.
[60] ——, "A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation," in *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.
[61] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning Parallel Applications on Multiprocessor Reservations," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.