Framework-Agnostic Model Inference for Intra-Thread Real-Time Tasks

Bite Ye* Filip Marković† Björn B. Brandenburg*

*Max Planck Institute for Software Systems, Germany

†University of Southampton, United Kingdom

***Abstract*—State-of-the-art *dynamic model extractors* (DMEs) for real-time systems all suffer from one of two limitations: either they require that all tasks correspond to individual OS *threads*, or they target and support only a specific framework or middleware. However, modern real-time systems rely on a diverse landscape of application frameworks, middleware libraries, OS abstraction layers, `async` runtimes, and other programming language facilities that multiplex many conceptually independent, time-sensitive tasks onto fewer OS threads. This paper presents the first DME specifically for *intra-thread tasks*—such as callbacks, message handlers, event handlers, futures, or coroutines—that is framework-agnostic and applicable to black-box threads (*i.e.*, without access to source code). To realize these capabilities, two major technical challenges are identified and solved: *intra-thread task discovery* and *release-time uncertainty*. A prototype implementation of the proposed approach for Linux is shown to achieve excellent accuracy ($>99\%$ period recovery and $<4\%$ pessimism in arrival-curve extraction) with low runtime overhead ($<2.5\%$ CPU utilization) across varied target workloads implemented with C++, Rust, C, and the ROS 2 middleware.**

I. Introduction

A *dynamic model extractor* (DME) instruments and observes a running real-time system *in situ*, discovers its constituent tasks, and maps the observed timing behavior to established models of real-time computation. For example, the recently introduced *Linux Model Extractor* (LiME) [10] derives classic *periodic* [27] and *sporadic* [29] task models, as well as generalized models described by arbitrary *arrival curves* [21, 35, 44], from thread-kernel interactions such as system calls and thread-state changes that it monitors via eBPF probes [49].

By reflecting a running system's *actual* behavior using concise, well-understood formalisms, DME tools provide strong introspection capabilities. For instance, they offer a useful perspective for validating temporal specifications, debugging timing issues, performing root-cause analysis, and, more generally, understanding a system's emergent timing behavior.

For example, LiME recently helped uncover *timer drift* in the implementation of a periodic workload [31], a subtle issue caused by relative sleeps that became apparent only through an unexpectedly poor model fit [10]. In addition, LiME can reveal system-internal real-time threads—such as high-priority kernel threads [10]—that are easily overlooked and often absent from system specifications, yet still affect timing behavior. In another recent example of models and systems diverging, Teper et al. [42] observed that callbacks scheduled under ROS 2's multithreaded executor do not satisfy the assumptions made in prior analyses of the executor. More generally, models and implementations can easily drift apart as software evolves; for instance, ROS 2's handling of timer callbacks has changed in fundamental, timing-relevant ways in recent versions without major announcements [6, 39]. Generally, task models derived from a system's *design* specify how a system *should* behave, whereas models recovered from its *implementation* reflect how it actually *is* behaving—it is therefore beneficial to regularly compare and reconcile the two views, lest they silently diverge.

From a developer's perspective, for ease of use and broad applicability, a DME should be automated, support a wide and diverse range of systems and applications, and induce only low runtime overheads to avoid disturbing the monitored system. The state-of-the-art DME tool, LiME [10], mostly achieves these goals—it is efficient, mostly automatic, and supports arbitrary *black-box* Linux threads—but it also suffers from a major limitation: each task must be a *separate* OS thread [10].

In modern software development, however, it is common to use middleware layers or frameworks such as ROS 2, Rust `async` runtimes, the C++ `std::async` API, or frequently also proprietary, in-house OS abstraction layers that multiplex many, conceptually separate tasks onto fewer *shared* threads (*i.e.*, *executors*). LiME currently does not support such workloads. While it can model the executor threads as irregularly activated sporadic tasks, it cannot "see inside" them and does not recognize the *intra-thread tasks* actually of interest to the developer.

This paper. To address this gap, we present the first *framework-agnostic DME for intra-thread tasks* such as callbacks, message and event handlers, futures, or coroutines. To this end, we:

- identify two major challenges—intra-thread task *discovery* and task-model inference with *uncertain release times*—that previously have received little attention (Sec. II);
- propose a novel methodology for the *automatic framework- and language-agnostic discovery* and efficient instrumentation of intra-thread tasks that works across a range of compiled programming languages typically encountered in embedded systems, including C++, Rust, and C (Sec. III);
- present new *uncertainty-aware* online model-inference algorithms [9, 48] for periodic and sporadic tasks (Sec. IV);
- report on an evaluation of a Linux-based implementation of our approach that demonstrates *high accuracy* ($>99\%$ period recovery and $<4\%$ pessimism in sporadic arrival curves), *low runtime overhead* ($<2.5\%$ CPU utilization), and *versatile applicability* to targets implemented using C++, Rust, C, or the ROS 2 middleware (Sec. V).

II. Example: Event- and Time-Driven Callbacks

To start, we motivate the model extraction problem for intra-thread tasks with a practical example implemented in C++. Consider a simple real-time application comprising two *control loops* operating at 200 Hz and 62.5 Hz, respectively, and a *communication endpoint* receiving new set points for the two control loops every 10 ms to 100 ms. From a modeling perspective, it is natural to represent the application as two *periodic tasks* with periods of 5 ms and 16 ms, respectively, and a *sporadic task* with a minimum inter-arrival time of 10 ms.

From a systems perspective, it can instead be convenient to realize the application's three constituent parts as *callbacks* of a single-threaded *executor*. Such an executor thread typically consists of a non-terminating main *event loop* that invokes the individual callbacks at the appropriate times or in response to newly arrived input. There are several practical advantages to such a design, including its resource efficiency (just a single thread), ease of deployment (just one binary), portability (virtually any OS), and ease of implementation (*e.g.*, avoids *pthreads*). Additionally, the callbacks can share state (*e.g.*, the controllers' set points) without any need for synchronization because they always execute sequentially, even when deployed on a multicore platform or under a preemptive scheduler that interleaves threads (*e.g.*, POSIX's SCHED_RR policy).

The problem. Developers naturally reason about timing requirements in terms of individual *intra-thread tasks* that realize operationally meaningful parts of an application (*e.g.*, "the 62.5 Hz control loop must execute once every 16 ms"). However, when implemented as part of an efficient single-threaded executor, this timing behavior is obscured and thus not directly observable from the kernel's vantage point.

This effect is illustrated in Fig. 1, which shows a trace of our example application as observed by the kernel at the thread level and the corresponding semantically meaningful intra-thread view. The executor thread is runnable whenever there is a pending callback to process. Conversely, the executor thread suspends only when there are no active callbacks. As a result, the thread's aggregate execution behavior effectively hides the underlying intra-thread task activity, as multiple intra-thread task activations can coalesce into a single thread activation.

For example, the sporadic activation of the communication endpoint at time 1 causes the thread to continue executing multiple callbacks back-to-back until time 7. From the kernel's point of view, the thread remains runnable throughout $[0, 7)$, without any indication that the 200 Hz controller was actually activated twice in that timeframe. For model extraction tools that rely on a thread's suspension pattern to infer task model parameters, such as LiME [10], the controller's true activation pattern is obscured and thus unrecoverable.

A second issue arises at time 13: a sporadic activation of the communication endpoint forces the otherwise idle thread to resume in between regular activations of the periodic callbacks. Such "stray" activations obscure the underlying periodicity of the thread's suspension pattern, again misleading any attempt at model inference using only thread-level information.

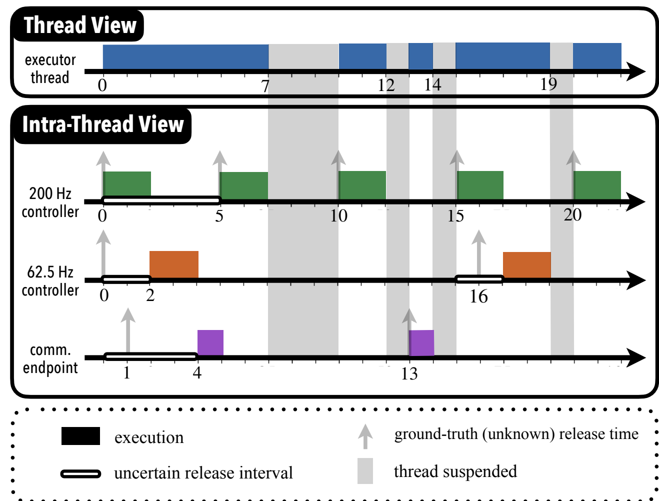

Fig. 1. Coalesced execution behavior of intra-thread tasks.

Finally, a third complication occurs at time 16: as the thread is already executing an instance of the 200 Hz controller (activated at time 15), no kernel-visible event is associated with the second periodic activation of the 62.5 Hz controller, which executes back-to-back without leaving a signal in the thread's suspension pattern. Generally, if there are multiple intra-thread tasks with non-harmonic periods, some activations will invariably fall into the execution window of other intra-thread tasks.

Thread view. Due to the above issues, model extractors relying solely on thread-level information (*i.e.*, on the kernel's vantage point) are rendered ineffective when applied to executor threads. For example, LiME is not able to extract a meaningful periodic model from the considered example application: after 30 seconds of observation, LiME's best-effort guess is a period of 4 ms, which matches neither control loop's true period. The inferred maximum release jitter is about 6 ms. As the jitter bound exceeds the inferred period, it is indicative of poor model fit. This reflects that LiME failed to recognize the application's true periodic structure, which is correct, since the executor's *aggregate* timing behavior is indeed *not* periodic.

Intra-thread view. However, to a developer seeking to validate the timing behavior of the application's three constituent intra-thread tasks, it is not helpful to learn that the executor thread as a whole is not executing periodically—that is expected, and provides little additional insight. Rather, the desired behavior is for the model extractor to "see through" the executor thread and to report timing models for the callbacks it dispatches. However, in order to report meaningful intra-thread task models, two fundamental challenges must be overcome.

Challenge 1: Task identification. Before an intra-thread task can be observed, and ultimately modeled, we must first discover it. Recall from Sec. I that we target a *framework-agnostic*, binary-only setting: we do not presume access to source code, and have no information about which middleware (if any) an executor thread uses. In this setting, it is not obvious how intra-thread tasks can be localized and targeted for observation.

However, framework-specific solutions come with significant

downsides. The large number of middleware frameworks in common use, both open-source and proprietary [23, 46], with diverse internals and ever evolving versions, makes case-by-case support impractical. Moreover, many systems rely on in-house frameworks or *ad hoc* event loops. For example, the application illustrated in Fig. 1 is implemented in C++ using a custom `std::async`-based executor, which would not be supported by an approach limited to well-known frameworks.

Motivated by these considerations, we propose a framework-agnostic intra-thread task detection heuristic in Sec. III.

Challenge 2: Uncertain release times. Nonetheless, even after all intra-thread tasks have been identified, it is still difficult to obtain the information necessary for model inference. In particular, even if we observe the start and end of each callback invocation, we still cannot infer the corresponding release time with certainty. The reason is that, as already discussed, callback activations do not necessarily correspond to observable events.

For example, in the case of a time-triggered periodic intra-thread task (such as the control loops in Fig. 1), it is simply the passage of time. Other unobservable scenarios include callbacks triggering other callbacks as part of their execution, and generally all code paths that enqueue new callbacks. If such a silent activation occurs while the executor thread is already runnable, we cannot infer with certainty when it took place, and instead can only derive an interval during which it must have occurred, called the *uncertain release window*.

In Fig. 1, for instance, we can infer from thread-level observations that the callback invoked at time 4 must have been queued for execution at some time during $[0, 4]$, but the true activation time remains unknown. Similarly, the first and second activations of the 62.5 Hz controller must have occurred during $[0, 2]$ and $[15, 17]$, respectively, but cannot be narrowed down further since activations at any times in these windows would have resulted in an identical thread suspension pattern.

Prior model-inference algorithms [10] require exact release times and are hence not applicable; in response, we develop new methods suitable for uncertain release windows in Sec. IV.

III. Intra-Thread Task Detection

To address Challenge 1, we developed a *profiling-* and *heuristic-based* methodology for the *fully automatic* detection and observation of intra-thread tasks that is applicable across a wide variety of frameworks and implementation approaches. Our implementation and the following description target Linux, but the general idea and methodology are not specific to Linux and readily transfer to other systems with similar capabilities.

A. Scope and Assumptions

We target programs implemented in an ahead-of-time compiled language that makes normal use of the thread's function call stack, which includes most programming languages commonly used in embedded real-time systems (such as C, C++, Ada, or Rust). We do not require access to source code. When available, debugging symbols improve detection accuracy, but they are not required. Binaries that have been intentionally obfuscated or otherwise hardened against analysis are beyond the scope of our methodology. We make no assumptions on whether programs are statically or dynamically linked.

As our methodology is rooted in dynamic profiling, it applies only to intra-thread tasks that execute frequently enough to be repeatedly observable. Intra-thread tasks that are not activated during profiling will not be detected. In other words, developers must drive the system under analysis with a representative workload that exercises all intra-thread tasks of interest.

The core idea behind the notion of "intra-thread tasks" is that developers use language features or programming frameworks to *compose* programs from *conceptually distinct, concurrent activities* that *recur* over time. Common examples of such *structured concurrency* include:

- **C function pointers** invoked from an event loop (*e.g.*, realized with the `epoll` API) in response to I/O events.
- **ROS 2 callbacks** for messages and timers.
- **Rust futures**, commonly created with the `async` keyword.
- **C++ closures** created with the `std::async` API.

This list is necessarily incomplete: there exist countless libraries, frameworks, and design patterns that all revolve around the idea of separating *what* code to run (*i.e.*, an intra-thread task's *individual* logic) from the mechanics of managing *when* to run it (*i.e.*, the *shared* event and timer management code).

We call any thread that primarily invokes such units of process-internal concurrency an *executor*. In this paper, we restrict our attention to single-threaded executors, meaning that each intra-thread task is bound to a specific executor and does not migrate between threads. Nonetheless, applications may still employ multiple executor threads, as long as each thread forms its own single-threaded executor, as demonstrated by the case study in Sec. V-A. In contrast, we do not yet handle multithreaded executors that manage a thread pool.

B. Approach and Overview

How can we handle a diverse landscape of varied implementation languages and programming abstractions in a generic, framework-agnostic way? The key insight underlying our methodology is that *all of these high-level abstractions leave a similar telltale signature on the function-call stack.*

Irrespective of whether an intra-thread task is realized as a plain C function, an instance of Rust's `Future` trait, or a C++ class implementing ROS 2's callback interface, after compilation, an invocation of any of these abstractions ultimately corresponds to a stack frame associated with a compiler-generated symbol that uniquely identifies the intra-thread task. Thus they all map to a common pattern that can be readily observed with standard profiling tools.

Based on this insight, our methodology consists of four steps: **(i)** first, we profile the target process with Linux's *perf* utility [13] to understand a thread's activity in terms of the symbols corresponding to addresses observed in function-call stack samples (Sec. III-C); **(ii)** then we find the stack frame corresponding to the thread's main event loop (Sec. III-D); **(iii)** from the main event loop, we identify the stack frames of intra-thread tasks (Sec. III-E); and finally, **(iv)** we inject

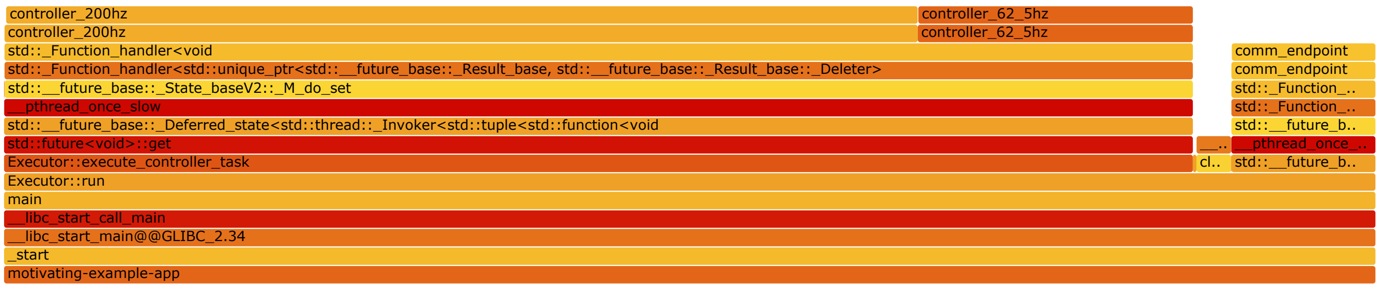

Fig. 2. The profiled folded stack ("flame graph") [19] of the example application from Sec. II. Call frames corresponding to the three intra-thread tasks, `controller_200hz`, `controller_62_5hz`, and `comm_endpoint`, appear at the top of the stack. Box width indicates the relative frequency of observation. The callback symbols each appear twice at the top of the folded stack due to the C++ compiler's implementation of the `std::async` API.

user-space eBPF probes (*uprobes*) [49] that record the start and finish times of each intra-thread task invocation (Sec. III-F).

C. Candidate Discovery via Profiling

After a configurable delay (by default, 1 second) intended to give the executor a chance to finish initializing and reach steady-state operation, we launch *perf* [13] to collect per-thread stack samples (by default, for 30 seconds). We use sampling-based profiling because it imposes low overhead, requires no code changes, and applies uniformly across frameworks.

The stack samples are then aggregated into *folded stacks* (also known as "flame graphs") [19]. Folded stacks summarize a corpus of call stacks into a hierarchical distribution that exposes calling relationships and how execution time accumulates along branches, which reveals where executors hand off to application code (*i.e.*, the intra-thread tasks we seek to discover).

For example, Fig. 2 shows a folded stack obtained from the application discussed in Sec. II. Each box corresponds to an observed stack frame. The folded stack reveals that the application's `main` function calls `Executor::run`, from which all further code is called. The box corresponding to the `Executor::run` stack frame spans the entire width of the figure, which indicates that it was present in *all* sampled stack traces. Conversely, the box for `Executor::execute_controller_task` reaches only partially across, as it was observed in many but not all of the sampled stack traces. The fraction of samples in which a stack frame was observed is proportional to the width of its box.

Boxes corresponding to the three intra-thread tasks can be seen at the top of the folded stack. Additionally, in between the `Executor::run` box and the actual callbacks, there are several stack frames corresponding to "glue code" found in the executor's implementation and the libraries that it uses.

A folded stack as shown in Fig. 2 offers a lot of useful, but also quite "messy" information. The challenge is to *automatically* make sense of the collected data. Specifically, we need to identify stack frames belonging to intra-thread tasks among a typically large number of other observed stack frames.

D. Event-Loop Detection Heuristic

A common characteristic shared across many languages and frameworks is that executor threads, after initialization, execute some kind of "main event loop" *ad infinitum* from which all other activity is initiated. For example, in Fig. 2, the box labeled `Executor::run` represents the stack frame of such a loop.

The first detection step is thus to identify the stack frame corresponding to the main event loop. Fortunately, it can be detected with a very simple heuristic, based on the following observation: it is inherent in the nature of an event loop that it will invoke *different* callbacks in response to different events, which implies that the folded stack necessarily *branches* on top of the event loop's stack frame. Conversely, the main event loop—which by assumption *always* executes during steady-state operation—is present in *all* stack samples. Therefore, the main event loop is uniquely identified by *the top-most stack frame* that occurs before any branches in the folded stack.

E. Intra-Thread Task Detection Heuristic

With the event loop in hand, we next identify stack frames corresponding to intra-thread tasks as follows. For ease of explanation, refer in the following to Fig. 3, which shows an abstracted view of a typical, generic folded stack.

We iteratively classify stack frames according to five rules into four different categories (as illustrated in Fig. 3):

1) **Intra-thread task entry points**: the bottom-most stack frame that uniquely identifies the intra-thread task. Finding these stack frames is the goal of the detection heuristic.
2) **Functions within an intra-thread task**: any stack frames corresponding to function calls within an intra-thread task, including library symbols and helper functions.
3) **Common-path frames**: stack frames above the main event loop that are not intra-thread tasks themselves, but from which multiple intra-thread tasks are reachable further up the stack. Concretely, in Fig. 2, `Executor::execute_controller_task` is a common-path frame that leads to `controller_200hz` and `controller_62_5hz`. Similarly, the intervening frames corresponding to low-level library code such as `__pthread_once_slow` are also common-path frames.
4) **Miscellaneous frames**: stack frames of library functions and other infrastructure code that should be disregarded.

Stack frames are classified according to the following rules.

Rule 1 (repeated symbols). Any stack frames corresponding to symbols that appear in multiple branches of the folded stack are classified as miscellaneous frames. The rationale is that, in callback-based executors, each intra-thread task usually has a single entry point, and thus any symbols that appear in multiple branches must correspond to shared infrastructure code.

In Fig. 3, for example, a "helper" frame occurs in different branches, above both intra-thread task entries 1 and 3, and thus

is discarded according to Rule 1. In practice, this rule filters out commonly called code such as memory management routines.

Rule 2 (source path). When the sampled target binary contains debug information, we use the embedded *compilation-unit paths* to classify symbols in well-known standard libraries as miscellaneous frames. For example, if a symbol's source location includes directory names that suggest that it resides in a common system or standard-library location (such as `libc`, `glibc`, `libstdc++`, `libc++`, `musl`, or `/usr/include`), then it is not application-specific code and hence not an intra-thread task.

This rule is optional. In particular, the remaining rules still permit targeting *stripped* binaries (without debug information), albeit with a potentially higher false-positive rate.

Rule 3 (binary path). Similarly, if a frame's symbol is not found in the target binary, it must instead reside in a shared library and thus likely indicates a miscellaneous frame. For example, in Fig. 3, the "stdlib" frame's symbol is not found in the target binary, so it is disregarded due to Rule 3.

This rule focuses the search on application-defined intra-thread tasks and helps to reduce false positives from shared libraries. However, particularly complex applications may choose to load functionality dynamically; to account for this possibility, Rule 3 is optional and can be disabled.

Rule 4 (child coverage). Starting from the event loop, the remaining stack frames are tested in a bottom-up traversal along all branches to detect common-path frames. The idea is the following: common-path frames are necessarily on the stack whenever one of the intra-thread tasks that they lead to is sampled, so a common-path frame should be sampled roughly just as often as its direct, not-yet-discarded children.

More precisely, for a stack frame p, let $S(p)$ denote the number of times it was sampled, and let $S_{\text{children}}(p)$ denote the *total* sample count of its direct children *from which stack frames not excluded by Rules 1–3 are reachable*. Stack frame p is classified as a common-path frame if $(S(p) - S_{\text{children}}(p))/S(p) \leq u$, where u is a small threshold to allow for sampling noise and some execution within p itself (by default, $u = 1\%$).

If p is classified as a common-path frame, then Rule 4 is recursively applied to each of its children (not yet excluded by Rules 1–3), and so forth. Otherwise, if p is not a common-path frame, it is marked as an *intra-thread task candidate* and the traversal of the current branch ends; any frames further up the stack are classified as functions within the intra-thread task.

Rule 4 can "overshoot" if the true intra-thread task entry point contains only a single child. For example, in Fig. 3, after applying Rules 1–3, Rule 4 marks the following stack frames as candidates: "ITT entry" 1, 3, and 4, but also "ITT func." 1, which is actually a function within an intra-thread task.

Rule 5 (parent promotion). To correct such cases, we apply a final top-down pass over the set of candidates. For each candidate frame c, we recursively inspect its parent p on the same branch. If p has exactly one child that is also a candidate (namely c), then we promote p to be the candidate for that branch, demote c to be a function within an intra-thread task, and recursively continue the check at p's parent. Promotion stops when a parent has multiple candidate children, when there is no parent (*i.e.*, if the executor loop is reached), or when the parent is ineligible due to Rules 1–3. After Rule 5, all remaining candidates are marked as intra-thread task entry points.

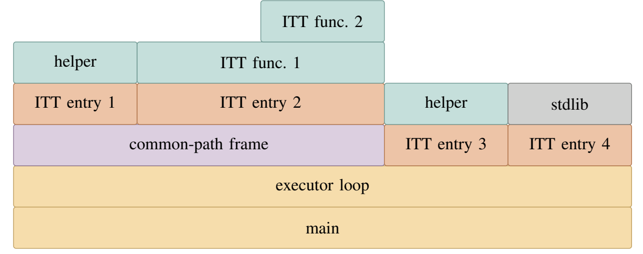

Fig. 3. Example folded stack exhibiting all four types of frames. The goal of the heuristic is to identify (only) the intra-thread task (ITT) entry points. Rule 1 eliminates the "helper" frames; Rule 3 eliminates the "stdlib" frame, Rule 4 identifies ITT entries 1, 3, and 4, and Rule 5 identifies ITT entry 2.

For example, applied to Fig. 3, this final pass transfers the candidate status from "ITT func." 1 to "ITT entry" 2, thus finally yielding the desired stack frame classification.

In our experience, and as demonstrated in Sec. V, Rules 1–5 are effective at surfacing all relevant intra-thread tasks. False negatives are rare (in fact, we did not observe any). In contrast, false positives do occur regularly, since Rules 1–3 are not guaranteed to detect all miscellaneous frames. However, as developers typically have little trouble excluding false positives, we consider it a good tradeoff to err on the side of reporting spurious tasks (rather than risk missing real intra-thread tasks).

F. Intra-Thread Task Instrumentation

After all intra-thread task candidates have been detected, we switch from sampling with *perf* to complete (*i.e.*, non-lossy) observation with eBPF. To this end, we instrument all intra-thread task entry points using eBPF user-space probes: *uprobes* at task entry and *uretprobes* at return [49]. Since eBPF instrumentation is injected transparently at runtime via binary rewriting, it does not require any modifications to the traced application by the user. It thus aligns well with our goal of framework-agnostic model inference.

The injected probes allow us to record, with low overhead, for *every* intra-thread task activation **(i)** the start of execution, **(ii)** the end of execution, and **(iii)** the total amount of processor time consumed in between (i) and (ii). We next discuss how to extract suitable task models from this information.

IV. Model Inference With Uncertain Release Times

Once every intra-thread task's entry frame has been instrumented (Sec. III-F), we observe the start and finish times of every invocation by recording a timestamp when the call occurs that pushes the frame onto the stack, and also when the return occurs that pops the frame off the stack. Additionally, using established techniques [10], we observe when the executor thread suspends, resumes, and executes. From this information, we seek to derive models that explain the intra-thread task's observed timing behavior (*i.e.*, activation pattern).

In general, there can be many executor threads in a system, each serving multiple intra-thread tasks. For ease of explanation,

we focus on a single intra-thread task in the following since model inference proceeds independently for each observed task.

As we observe precisely when intra-thread tasks start and cease to execute, including any suspensions that may occur in between (*e.g.*, due to blocking I/O), it is straightforward to obtain execution-time models using the approach followed by LIME [10]; we hence will not dwell on this point any further.

Rather, as motivated in Sec. II, the problem is to estimate release times, which we do not directly observe (Challenge 2). Each time an intra-thread task is activated, the *ground-truth release time* describes the earliest moment at which the executor could possibly execute the intra-thread task in question. For example, in an executor that maintains a ready queue of callbacks to execute, a callback's release time denotes the point in time at which both the callback has been added to the ready queue and, if necessary, the executor is resumed.

In general, there can be a considerable delay between an intra-thread task's release (which we cannot observe) and when it actually starts executing (which we trace) if the executor first attends to other pending work (*e.g.*, such as during $[0, 2)$ in Fig. 1). Additionally, it is possible for a release to occur while the executor is already busy executing (*e.g.*, at time 16 in Fig. 1). As elaborated in Sec. II, both effects obscure the ground-truth release time. We make no assumptions on what scheduling policy, if any, is used, but do expect executors to be *work-conserving* (*i.e.*, an idle executor that suspended will resume immediately when an intra-thread task is released).

In the following, we first introduce models that describe the activation patterns of *periodic* and *sporadic* intra-thread tasks in the face of uncertain release times, and then present algorithms for inferring such models from a stream of timestamps.

A. Model Definitions

We assume discrete time (*e.g.*, processor cycles) and let r_i denote the ground-truth *release time*, s_i the corresponding *start of execution*, and f_i the *finish time* of an intra-thread task's i-th activation. Whereas r_i is unknown in general, s_i and f_i are timestamps recorded by our instrumentation (Sec. III-F). For example, in Fig. 1, for the 62.5 Hz controller's second activation, we have $r_2 = 16$, $s_2 = 17$, and $f_2 = 19$. We call $[s_i, f_i)$ the intra-thread task's i-th *execution window*.

Additionally, let $W(t)$ denote the *latest wake-up time* of the executor before time t that does not fall within an execution window of any intra-thread task. That is, $W(t)$ is the latest time the executor thread resumes from a self-suspension while not executing any intra-thread task. (If no such wake-up occurred before t, $W(t)$ is defined to be the start of observation.)

We let $R_i = [r_i^-, r_i^+]$ denote the *uncertain release window* of the intra-thread task's i-th activation. To ensure that it includes the actual release ($r_i \in R_i$), we estimate the release window's lower and upper bounds as follows:

- $r_i^- \triangleq W(s_i)$, as a work-conserving executor cannot idle while the activation is pending (*i.e.*, $\forall i,\ W(s_i) \leq r_i$); and
- $r_i^+ \triangleq s_i$, since an intra-thread task cannot start execution prior to its release (*i.e.*, $\forall i,\ r_i \leq s_i$).

For example, consider again the 62.5 Hz controller in Fig. 1: for the first activation, $r_1 = 0 \in [W(s_1), s_1] = [0, 2]$, and for the second activation, $r_2 = 16 \in [W(s_2), s_2] = [15, 17]$.

Periodic models. The classic *periodic task model* [3, 27, 33] describes a sequence of releases with three parameters: an offset Φ, the period T, and a maximum jitter bound J. Based on these parameters, the i-th (ideal) *arrival time* is $a_i = \Phi + (i-1) \times T$, and the acceptable *arrival window* is $A(i) \triangleq [a_i, a_i + J]$. The periodic model constrains releases to occur within the acceptable arrival-time window: $\forall i,\ r_i \in A(i)$.

LIME implements a model-inference algorithm that, given a release sequence $r_1, r_2, r_3, \ldots$, recovers suitable parameters Φ, T, and J such that the arrival-window constraint is satisfied [10]. We call such a model a *point-release model*, as it is inherently based on the sequence of exact release times.

Point-release models are not suitable in our setting as we do not observe release times. We therefore introduce two alternative periodic models applicable to uncertain release windows. First, given a parameter tuple (J, T, Φ) as above, the *possible-fit periodic model* requires that each uncertain release window *intersects* with the corresponding arrival window: $\forall i,\ R_i \cap A(i) \neq \emptyset$. Second, the more conservative *certain-fit periodic model* requires uncertain release windows to be *contained* within arrival windows: $\forall i,\ R_i \subseteq A(i)$.

Intuitively, the certain-fit model ensures that *any possible* realization of the ground-truth release time $r_i \in R_i$ satisfies the classic periodic model's arrival-window constraint, which makes this model well-suited for worst-case analysis. In contrast, the possible-fit model requires only that there *exists* a realization $r_i \in R_i$ that satisfies the classic arrival-window constraint; because this is weaker, it can be seen as a *plausibility check*, but it is not suitable for the derivation of guarantees.

Sporadic models. Sporadic tasks are generally described by an *upper arrival curve* $\alpha^+(\Delta)$ and a *lower arrival curve* $\alpha^-(\Delta)$ that bound the number of releases in any interval from above and below [21, 24, 35, 44]. Formally, $\forall\, \Delta \geq 0,\ \forall\, t : \alpha^-(\Delta) \leq \left|\{\, i \mid r_i \in I_t^\Delta \,\}\right| \leq \alpha^+(\Delta)$, where $I_t^\Delta = [t, t + \Delta)$. Arrival curves generalize the classic scalar sporadic model [29] by allowing for bursts while remaining bounded for larger Δ.

As the arrival curves $\alpha^+(\Delta)$ and $\alpha^-(\Delta)$ are point-release models, we instead use four *over-* and *under-approximations* $\alpha_{\text{lo}}^-(\Delta)$, $\alpha_{\text{hi}}^-(\Delta)$, $\alpha_{\text{lo}}^+(\Delta)$, and $\alpha_{\text{hi}}^+(\Delta)$, where $\forall\, \Delta \geq 0,\ \forall\, t :$

- $\alpha_{\text{lo}}^-(\Delta) \leq \left|\{\, i \mid R_i \subseteq I_t^\Delta \,\}\right| \leq \alpha_{\text{lo}}^+(\Delta)$ and
- $\alpha_{\text{hi}}^-(\Delta) \leq \left|\{\, i \mid R_i \cap I_t^\Delta \neq \emptyset \,\}\right| \leq \alpha_{\text{hi}}^+(\Delta)$.

The corresponding inference algorithms ensure that, $\forall \Delta$, $\alpha_{\text{lo}}^+(\Delta) \leq \alpha^+(\Delta) \leq \alpha_{\text{hi}}^+(\Delta)$ and $\alpha_{\text{lo}}^-(\Delta) \leq \alpha^-(\Delta) \leq \alpha_{\text{hi}}^-(\Delta)$. Therefore, $\alpha_{\text{hi}}^+(\Delta)$ safely over-approximates $\alpha^+(\Delta)$, and $\alpha_{\text{lo}}^-(\Delta)$ safely under-approximates $\alpha^-(\Delta)$. The other two curves are useful to assess the tightness of the approximation.

B. Sporadic Model Inference

The objective of model inference is to recover task parameters from a stream of release-window observations $R_1, R_2, \ldots$ such that the model's constraints are satisfied as tightly as possible. We begin with sporadic models, which are easier to infer.

```
1 Input: $R_1, R_2, R_3, \ldots,$ where $R_i = [r_i^-, r_i^+]$
2 Initialize: $\delta^{hi}_{min}[0] \leftarrow 0$; $\delta^{hi}_{min}[1] \leftarrow 1$; $\forall n \geq 2,\ \delta^{hi}_{min}[n] \leftarrow \infty$
3 for each $R_i$ do:
4   for each $n$ such that $2 \leq n \leq i$ do:
5     $\delta^{hi}_{min}[n] \leftarrow \min(\delta^{hi}_{min}[n],\ \max(1,\ r_i^- - r_{i-n+1}^+ + 1))$
6 $\alpha^+_{hi}(\Delta) \triangleq \min\{n \mid \delta^{hi}_{min}[n+1] > \Delta\}$ if $0 \leq \Delta < \max_n\{\delta^{hi}_{min}[n]\}$
```

Listing 1. $\alpha^+_{hi}(\Delta)$ inference

```
1 Input: $R_1, R_2, R_3, \ldots,$ where $R_i = [r_i^-, r_i^+]$
2 Initialize: $\forall n \geq 0,\ \delta^{lo}_{max}[n] \leftarrow 0$
3 for each $R_i$ do:
4   for each $n$ such that $0 \leq n < i-1$ do:
5     $\delta^{lo}_{max}[n] \leftarrow \max(\delta^{lo}_{max}[n],\ \max(0,\ r_i^+ - r_{i-n-1}^- - 1))$
6 $\alpha^-_{lo}(\Delta) \triangleq \max\{n \mid \delta^{lo}_{max}[n] \leq \Delta\}$ if $0 \leq \Delta \leq \max_n\{\delta^{lo}_{max}[n]\}$
```

Listing 2. $\alpha^-_{lo}(\Delta)$ inference

Listing 1 provides the algorithm we use to infer $\alpha^+_{hi}(\Delta)$. The central data structure is the vector $\delta^{hi}_{min}[n]$, which stores an *under*-approximation of the length of the *shortest* interval that contains *at least* n job releases. Initially, in line 2, $\delta^{hi}_{min}[n]$ is set to ∞ for $n \geq 2$; by default, $\delta^{hi}_{min}[0] = 0$ and $\delta^{hi}_{min}[1] = 1$.

Lines 3–5 then process all observed release windows to populate $\delta^{hi}_{min}[n]$. For each observed R_i and $2 \leq n \leq i$, the (unknown) interval $[r_{i-n+1}, r_i]$ contains (at least) the n job releases $r_{i-n+1}, \ldots, r_i$. (The interval may contain more than n job releases if multiple jobs are released simultaneously at either endpoint.) The two release times r_{i-n+1} and r_i are unknown, but we do know that $r_{i-n+1} \in R_{i-n+1}$ and $r_i \in R_i$. We hence under-approximate the length of the interval $[r_{i-n+1}, r_i]$ as $\max(1, r_i^- - r_{i-n+1}^+ + 1)$ based on the upper bound $r_{i-n+1}^+ \geq r_{i-n+1}$, the lower bound $r_i^- \leq r_i$, and because the interval cannot be empty. Line 5 updates $\delta^{hi}_{min}[n]$ accordingly.

Finally, line 6 defines $\alpha^+_{hi}(\Delta)$ using the final vector δ^{hi}_{min}. Recall that $\alpha^+_{hi}(\Delta)$ must upper-bound the maximum number of job releases in any interval of length Δ. Intuitively, if we have evidence that the shortest interval containing $n+1$ job releases is *longer* than Δ, then there are at most n job releases in an interval of length Δ. Formally, for a given Δ such that $0 \leq \Delta < \max_n\{\delta^{hi}_{min}[n]\}$, the maximum number of job releases in any interval of length Δ is upper-bounded by the smallest n such that $\delta^{hi}_{min}[n+1] > \Delta$. The curve $\alpha^+_{hi}(\Delta)$ is undefined for $\Delta \geq \max_n\{\delta^{hi}_{min}[n]\}$ due to a lack of observations.

The extraction of $\alpha^-_{lo}(\Delta)$, given in Listing 2, follows the same scheme, but in "reversed" manner. Here, the vector $\delta^{lo}_{max}[n]$ tracks an *over*-approximation of the *longest* interval in which *at most* n releases have been seen. Initially, line 2 of Listing 2 sets $\delta^{lo}_{max}[n]$ to 0 for all $n \geq 0$. In contrast to Listing 1, $n = 0$ and $n = 1$ are not special cases in the inference of $\alpha^-_{lo}(\Delta)$.

For each R_i and $0 \leq n < i-1$, the *open* interval (r_{i-n-1}, r_i) contains (at most) the n releases $r_{i-n}, r_{i-n+1}, \ldots, r_{i-1}$ (or fewer, if some of the releases coincide with either endpoint). Line 5 thus over-approximates the length of the interval (r_{i-n-1}, r_i) as $\max(0, r_i^+ - r_{i-n-1}^- - 1)$ since $r_{i-n-1}^- \leq r_{i-n-1}$ and $r_i \leq r_i^+$, and updates $\delta^{lo}_{max}[n]$ accordingly. Line 6 defines $\alpha^-_{lo}(\Delta)$, which lower-bounds the minimum number of jobs released in any interval of length Δ, to be the largest n such that Δ does not exceed $\delta^{lo}_{max}[n]$. As before, $\alpha^-_{lo}(\Delta)$ is undefined for $\Delta > \max_n\{\delta^{lo}_{max}[n]\}$ due to a lack of observations.

Due to space constraints, we omit the inference algorithms for $\alpha^+_{lo}(\Delta)$ and $\alpha^-_{hi}(\Delta)$, which follow the same patterns as those for $\alpha^+_{hi}(\Delta)$ and $\alpha^-_{lo}(\Delta)$, respectively. In practice, it is too costly to store and update $\delta_{max}[n]$ and $\delta_{min}[n]$ for arbitrarily large n. Instead, we track only up to a fixed number of releases n^{limit} by replacing i with n^{limit} in line 4 of both Listings 1 and 2, using $n^{limit} = 128$ by default. For larger n, the curves can be safely extrapolated from the inferred prefix [21, 35, 39].

C. Periodic Model Inference

We adapted LIME's model-inference algorithm for point-release models [10] to support uncertain release windows. Listing 3 shows the resulting algorithm for possible-fit models.

The algorithm processes release windows in *batches* of a configurable batch size b (line 1). The batches are chosen to overlap in o release windows for continuity: the k-th batch consists of the release windows $\mathbf{B}_k = \langle R_{\beta(k)+1}, \ldots, R_{\beta(k)+b} \rangle$, where $\beta(k) = (k-1) \cdot (b-o)$. We use $b = 4096$ and $o = 1$ by default.

Each batch $\mathbf{B}_k$ is consumed in sequence by lines 6–37. The first batch $\mathbf{B}_1$ is handled specially to determine the initial set of *model candidates* $\mathcal{C}$ (lines 10–12). Any subsequent batches are used only to update and prune $\mathcal{C}$ (lines 13–37). Finally, lines 38–43 select the final model estimate from $\mathcal{C}$.

In the following, we use $\mathcal{M}^{poss}(B, T') \triangleq (J', T', \Phi')$ to denote the possible-fit model implied by the release windows in batch B for a given candidate period T', where

- $\Phi' = \min_{R_i \in B}\{r_i^+ - T' \cdot (i-1)\}$, and
- $J' = \max_{R_i \in B}\{r_i^- - T' \cdot (i-1) - \Phi'\}$.

The underlying intuition is as follows: A periodic model with parameters Φ' and J' constrains the i-th job release to occur at some time on or after $(i-1)T' + \Phi'$ and no later than $(i-1)T' + \Phi' + J'$. Thus, for the inferred model to intersect R_i, $(i-1)T' + \Phi'$ must be no larger than r_i^+ and $(i-1)T' + \Phi' + J'$ no less than r_i^-. As J' and Φ' are uniquely defined for a given T', the main challenge that Listing 3 must solve is to identify the "right" period T' among many possible candidates.

Let us now consider the main loop in lines 6–37 step by step. First, in order to prevent any exceptionally noisy observations at the endpoints of the current batch $\mathbf{B}_k$ from biasing the period estimation, line 7 derives an *outlier-truncated* version $\mathbf{B}'_k$ of the current batch. To this end, the algorithm considers the gaps between consecutive release-window upper bounds (*i.e.*, $r_{i+1}^+ - r_i^+$ for $R_i, R_{i+1} \in \mathbf{B}_k$), and using a lightweight *Hampel identifier* [*e.g.*, 32, Ch. 3.2.2], flags any gaps that appear to be outliers (for brevity, this is not shown in Listing 3). Let R_x be the *first* release window in $\mathbf{B}_k$ such that $r_{x+1}^+ - r_x^+$ is not an outlier gap, and let R_y be the *last* release window in $\mathbf{B}_k$ such that $r_y^+ - r_{y-1}^+$ is not an outlier gap: the outlier-truncated version of batch $\mathbf{B}_k$ is then given by $\mathbf{B}'_k = \langle R_x, \ldots, R_y \rangle$.

Based on the truncated batch $\mathbf{B}'_k$, line 8 computes the *mean gap* $\hat{g}_k = (\sum_{R_i, R_{i+1} \in \mathbf{B}'_k} r_{i+1}^+ - r_i^+)/(|\mathbf{B}'_k| - 1)$. Line 9 uses the mean gap to search for the periodic model that *minimizes*

1 Input: $\mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3, \ldots$ where $\mathbf{B}_k = \langle R_{\beta(k)+1}, \ldots, R_{\beta(k)+b} \rangle$
2 Parameters:
3 $\quad X_1$: negligible-jitter threshold (default: 1 ms)
4 $\quad X_2$: candidate-pruning factor (default: 5)
5 $\quad X_3$: candidate-selection factor (default: 1.25)
6 for each $\mathbf{B}_k$ do:
7 $\quad \mathbf{B}'_k \leftarrow$ outlier-truncated version of $\mathbf{B}_k$
8 $\quad \hat{g}_k \leftarrow$ mean gap of $\mathbf{B}'_k = (\sum_{R_i, R_{i+1} \in \mathbf{B}'_k} r^+_{i+1} - r^+_i)/(|\mathbf{B}'_k| - 1)$
9 $\quad (J^{\text{min}}_k, T^{\text{min}}_k, \Phi^{\text{min}}_k) \leftarrow \min(\{\mathcal{M}^{\text{poss}}(\mathbf{B}'_k, T) \mid T \in [0.5\hat{g}_k, 2\hat{g}_k]\})$
10 $\quad$ if $k = 1$ then:
11 $\quad\quad \overline{T} \leftarrow T^{\text{min}}_k$
12 $\quad\quad \mathcal{C} \leftarrow \{\mathcal{M}^{\text{poss}}(\mathbf{B}_1, T) \mid T \in \mathcal{T}_1(T^{\text{min}}_k) \cup \mathcal{T}_2(T^{\text{min}}_k)\}$
13 $\quad$ else:
14 $\quad\quad \overline{T} \leftarrow \overline{T} + (T^{\text{min}}_k - \overline{T})/k$
15 $\quad\quad \mathcal{D} \leftarrow \emptyset$
16 $\quad\quad$ for each $T^{\text{ref}} \in \{\overline{T}, T^{\text{min}}_k\}$ do:
17 $\quad\quad\quad$ choose $(J^*, T^*, \Phi^*) \in \mathcal{C}$ s.th. $|T^* - T^{\text{ref}}|$ is minimal
18 $\quad\quad\quad \ell \leftarrow \beta(k-1) + b - 1$
19 $\quad\quad\quad$ if $T^* < T^{\text{ref}}$ then:
20 $\quad\quad\quad\quad \Phi'' \leftarrow \Phi^* + \ell \cdot (T^* - T^{\text{ref}})$
21 $\quad\quad\quad\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(\Phi^* + J^* - \Phi'',\ T^{\text{ref}},\ \Phi'')\}$
22 $\quad\quad\quad$ else:
23 $\quad\quad\quad\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(J^* + \ell \cdot (T^* - T^{\text{ref}}),\ T^{\text{ref}},\ \Phi^*)\}$
24 $\quad\quad \mathcal{C}' \leftarrow \emptyset$
25 $\quad\quad$ for each $(J, T, \Phi) \in \mathcal{C} \cup \mathcal{D}$ do:
26 $\quad\quad\quad (J', T', \Phi') \leftarrow (J, T, \Phi)$
27 $\quad\quad\quad$ for each $R_i \in \mathbf{B}_k$ do:
28 $\quad\quad\quad\quad \delta \leftarrow r^+_i - (i-1) \cdot T' - \Phi'$
29 $\quad\quad\quad\quad$ if $\delta < 0$ then:
30 $\quad\quad\quad\quad\quad \Phi' \leftarrow \Phi' + \delta$
31 $\quad\quad\quad\quad\quad J' \leftarrow J' - \delta$
32 $\quad\quad\quad\quad J' \leftarrow \max(J', r^-_i - (i-1) \cdot T' - \Phi')$
33 $\quad\quad\quad \mathcal{C}' \leftarrow \mathcal{C}' \cup \{(J', T', \Phi')\}$
34 $\quad\quad L \leftarrow \min(\{\infty\} \cup \{J \mid (J, T, \Phi) \in \mathcal{C}' \wedge J > 0\})$
35 $\quad\quad n^{\mathcal{C}} \leftarrow |\mathcal{C}|$
36 $\quad\quad \mathcal{C} \leftarrow \{(J, T, \Phi) \mid (J, T, \Phi) \in \mathcal{C}' \wedge (J \leq X_1 \vee J \leq X_2 \cdot L)\}$
37 $\quad\quad \mathcal{C} \leftarrow$ top $n^{\mathcal{C}}$ models in $\mathcal{C}$ sorted by inc. jitter
38 $L \leftarrow \min(\{J \mid (J, T, \Phi) \in \mathcal{C}\})$
39 $\mathcal{A} \leftarrow \{(J, T, \Phi) \mid (J, T, \Phi) \in \mathcal{C} \wedge (J \leq X_1 \vee J \leq X_3 \cdot L)\}$
40 choose $(J, T, \Phi) \in \mathcal{A}$ such that:
41 $\quad$ 1. T maximizes the number of trailing zeroes and
42 $\quad$ 2. J is minimal (if there are multiple such T).
43 return $(\max(0, J), T, \Phi)$ as the final model estimate

Listing 3. Heuristic for possible-fit periodic model inference.

the jitter bound (w.r.t. the observations in $\mathbf{B}'_k$), based on the intuition that the best-fitting model will require the least release jitter to explain all observations. The period T^{min}_k that minimizes J^{min}_k across all periods in the search space $[0.5\hat{g}_k, 2\hat{g}_k]$ can be found efficiently via ternary search [10]. Note that J^{min}_k can be negative due to the underlying uncertainty in observations.

The algorithm maintains a running estimate $\overline{T}$ of the mean of all $T^{\text{min}}_1, \ldots, T^{\text{min}}_k$. In the special case of the first batch $\mathbf{B}_1$, $\overline{T}$ is simply initialized to T^{min}_1 (line 11). In subsequent batches, $\overline{T}$ is updated using *Welford's algorithm* (line 14).

When processing the first batch, the jitter-minimizing period T^{min}_1 is further used to populate the set of candidate models $\mathcal{C}$ (line 12), based on two sets of period estimates: $\mathcal{T}_1(T^{\text{min}}_1)$ contains N periods evenly spaced throughout $[T^{\text{min}}_1 - \sigma \cdot |J^{\text{min}}_1|,\ T^{\text{min}}_1 + \sigma \cdot |J^{\text{min}}_1|]$, with $N = 50$ and $\sigma = 3$ by default; and $\mathcal{T}_2(T^{\text{min}}_1)$ contains variations of T^{min}_1 rounded to nearby integral values that human designers are likely to select (*i.e.*, $\left(\lfloor T^{\text{min}}_1 / 10^x \rceil \pm y\right) \cdot 10^x$ for $y \in \{0, 1, 2\}$ and $x \in \{1, 2, \ldots\}$).

Every subsequent batch $\mathbf{B}_k$ ($k > 1$) is processed in three steps: first, lines 15–23 derive two additional model candidates $\mathcal{D}$; second, lines 24–33 update all model estimates to account for the observations in $\mathbf{B}_k$; and finally, lines 34–37 discard any diverging and hence now irrelevant models.

To obtain $\mathcal{D}$, the algorithm constructs two models that implicitly reflect all prior observations for $T^{\text{ref}} \in \{\overline{T}, T^{\text{min}}_k\}$. Line 17 selects a candidate $(J^*, T^*, \Phi^*) \in \mathcal{C}$ that minimizes $|T^* - T^{\text{ref}}|$ (*i.e.*, the one closest to T^{ref}). If $T^* < T^{\text{ref}}$, then lines 20–21 add the model $(J'', T^{\text{ref}}, \Phi'')$ with $\Phi'' = \Phi^* + \ell \cdot (T^* - T^{\text{ref}})$ and $J'' = \Phi^* + J^* - \Phi''$ for $\ell = \beta(k-1) + b - 1$, which corrects for the larger reference period using the last already-processed observation $R_{\beta(k-1)+b}$ as a reference point. Otherwise, line 23 adds $(J'', T^{\text{ref}}, \Phi'')$ with $\Phi'' = \Phi^*$ and $J'' = J^* + \ell \cdot (T^* - T^{\text{ref}})$, which analogously corrects for a shorter period $T^{\text{ref}} \leq T^*$.

Lines 24–33 then update all candidates, reducing the offset Φ' (lines 29–31) and increasing the jitter bound J' (line 32) as necessary to intersect all release windows in $\mathbf{B}_k$. The set of candidate models $\mathcal{C}$ is populated (line 12) and updated (line 27) based on the full batch $\mathbf{B}_k$, and *not* the truncated version $\mathbf{B}'_k$, to ensure that the inferred models reflect *all* observations.

Finally, line 34 computes the least positive jitter bound L (if any), which line 36 uses to drop any model candidates with a "non-negligible" jitter bound (> 1 ms by default) larger than L by a factor exceeding the pruning threshold (5 by default). Lines 35 and 37 prevent the candidate set from growing in size, which could occur if line 36 prunes fewer than $|\mathcal{D}|$ candidates.

After all batches have been processed, lines 38–43 choose the final model estimate from the remaining candidates in $\mathcal{C}$. Line 38 again computes the least jitter bound L. Line 39 derives the set of *acceptable* models $\mathcal{A}$, which includes all models that do not exceed L by more than the candidate-selection factor (1.25 by default) and any model with "negligible" jitter. From $\mathcal{A}$, the model with the largest-granularity period is chosen (*i.e.*, one that maximizes the number of trailing zeroes in decimal notation), which biases the selection towards "human-friendly" periods at the expense of a moderate increase in jitter. If there are multiple candidates with the same period granularity, then the jitter bound serves as a tiebreaker (line 42).

Certain-fit inference works analogously to Listing 3, with the following changes to ensure full coverage of each R_i:

- in lines 9 and 12, $\mathcal{M}^{\text{poss}}(B, T')$ is replaced with $\mathcal{M}^{\text{cert}}(B, T') \triangleq (J', T', \Phi')$, where $\Phi' = \min_{R_i \in B}\{r^-_i - T' \cdot (i-1)\}$ and $J' = \max_{R_i \in B}\{r^+_i - T' \cdot (i-1) - \Phi'\}$;
- line 28 uses r^-_i instead of r^+_i when computing δ; and
- line 32 uses r^+_i instead of r^-_i when adjusting J'.

For further study and to encourage reuse, we provide a reference implementation of all proposed model-inference algorithms as an open-source Python library [9, 48].

V. Implementation and Evaluation

We implemented the proposed approach as an extension of the existing LIME codebase [10] on Linux using Rust. In the following, we refer to our extended version as LIME$^{\text{IT}}$. LIME$^{\text{IT}}$

consists of two parts: the first performs intra-thread task *discovery* (Secs. III-C to III-E) using *perf*, while the second realizes the actual *model extraction* (Secs. III-F and IV) using eBPF.

We evaluated LIME^{IT} with both a case study (Sec. V-A) to assess its ability to discover and model real-world intra-thread tasks, and synthetic workloads (Sec. V-B) to systematically test its performance characteristics and to demonstrate its applicability to different programming languages and abstractions.

All experiments were conducted on a *Raspberry Pi 5* (RP5), equipped with a quad-core ARM Cortex A76 processor and 8 GiB RAM, running Ubuntu Server 24.04 LTS with Linux kernel 6.12 (compiled with PREEMPT_RT and eBPF support). In general, *perf* can run at different sampling rates: higher rates increase overhead, but can also improve discovery accuracy. By default, LIME^{IT} uses 10 kHz, which provided a good balance between accuracy and overhead in our experiments.

A. Case Study: The ROS 2 Reference System

We applied LIME^{IT} to the ROS 2 reference system maintained by the ROS 2 Real-Time Working Group [36]. We use its `prioritized` configuration, which is structurally representative of an autonomous driving (AD) pipeline. The configuration contains 24 *nodes*, which are ROS 2 software components that encapsulate one or more callbacks and communicate via publisher-subscriber topics. These 24 nodes comprise six sensor nodes, nine transform nodes, five fusion nodes, one cyclic node, two intersection nodes, and one command node, and are spread across five single-threaded executors: the *front*, *rear*, and *fusion* executors run at SCHED_RR priority 1, the *planner* executor at priority 30, and the remaining nodes are handled by an executor without real-time priority (*i.e.*, a best-effort background thread under Linux's default EEVDF [40] scheduler).

At the intra-thread level, however, the system exposes 29 tasks because each fusion node encapsulates two callbacks, as indicated with subscripts in Table I. Seven of the nodes are timer-driven and thus have explicit periods configured by the application's initialization code; the remaining nodes are message-driven (*i.e.*, triggered when new messages appear on topics they subscribe to) and hence do not have explicit periods. The reference system does not include actual AD logic, but emulates each callback's execution with a `number_cruncher` function; we added a C++ template *tag* to this function's type to ensure that each callback has a unique entry point, as real AD logic does.

LIME^{IT} successfully discovered all 29 intra-thread tasks and extracted periodic models as reported in Table I. For each timer-driven node, LIME^{IT} detected a second intra-thread task not listed in Table I that reflects ROS 2's internal timer management, which takes place before the actual callback runs. Additionally, LIME^{IT} detected 23 further intra-thread tasks. These are not strictly false positives, but rather code regularly called by the ROS 2 executor, and can thus be considered *system-internal* intra-thread tasks (*e.g.*, 17 of these are related to message propagation and hence inherent to the workload). For brevity, we also omit these from Table I, but note that a schedulability analysis of the system would have to take such system-internal tasks into account, which highlights the value of *discovering*

TABLE I
PERIODIC MODELS EXTRACTED FROM THE ROS 2 REFERENCE SYSTEM

Timer Callbacks

Callback	Executor	Prio	Period [ms] Configured	Period [ms] Inferred	Jitter [ms] PF	Jitter [ms] CF
BP	planner	RR 30	100	100	0.06	0.11
FLidar	front	RR 1	100	100	0	0.08
RLidar	rear	RR 1	100	100	0	0.09
PCMap	other	EEVDF	120	120	0.12	36.88
Vis	other	EEVDF	60	60	0.35	43.73
L2Map	other	EEVDF	100	100	0.50	40.62
ECSet	other	EEVDF	25	25	0.73	44.29

Message Callbacks

Callback	Trigger(s)	Executor	Prio	Inferred Period [ms]	Jitter [ms] PF	Jitter [ms] CF
PTF	FLidar	front	RR 1	100	0	0.34
PTR	RLidar	rear	RR 1	100	0	0.42
RGF	PCF	fusion	RR 1	100	0	41.26
OCE	ECD	fusion	RR 1	100	0	11.24
PCF$_1$	PTR	fusion	RR 1	100	0.17	0.37
PCF$_2$	PTR	fusion	RR 1	100	0.52	0.79
ECD	RGF, ECSet	fusion	RR 1	20	15.62	29.08
VGD	PCF	other	EEVDF	100	9.93	47.71
PCML	PCMap	other	EEVDF	120	1.10	44.72
MPC	BP	other	EEVDF	100	0.68	43.38
Park	L2ML	other	EEVDF	120	0	69.59
LPlan	L2ML	other	EEVDF	120	0	66.16
ITO	ECD	other	EEVDF	25	7.36	56.26
NDT$_1$	VGD	other	EEVDF	100	0	53.28
NDT$_2$	PCML	other	EEVDF	120	0	48.57
VI$_1$	MPC	other	EEVDF	100	0	47.60
VI$_2$	BP	other	EEVDF	100	0	32.60
L2GP$_1$	Vis	other	EEVDF	60	0	72.03
L2GP$_2$	NDT	other	EEVDF	120	0	55.40
L2ML$_1$	L2Map	other	EEVDF	100	4.85	61.93
L2ML$_2$	L2GP	other	EEVDF	120	0	54.94
VDBW	VI	other	EEVDF	100	0	62.30

Abbrev.: FLidar:FrontLidarDriver, RLidar:RearLidarDriver, PCF:PointCloudFusion, PCMap:PointCloudMap, RGF:RayGroundFilter, ECD:EuclideanClusterDetector, MPC:MPCController, BP:BehaviorPlanner, L2ML:Lanelet2MapLoader, L2GP:Lanelet2GlobalPlanner, L2Map:Lanelet2Map, VGD:VoxelGridDownsampler, PCML:PointCloudMapLoader, NDT:NDTLocalizer, Vis:Visualizer, VI:VehicleInterface, PTF:PointsTransformerFront, PTR:PointsTransformerRear, OCE:ObjectCollisionEstimator, Park:ParkingPlanner, LPlan:LanePlanner, ECSet:EuclideanClusterSettings, VDBW:VehicleDBWSystem, ITO: IntersectionOutput.

what *actually* executes in a system (rather than relying on a specification listing only the *intended* payload tasks).

For each detected intra-thread task, LIME^{IT} extracted possible-fit (PF) and certain-fit (CF) periodic models as reported in Table I, based on one minute of tracing. For timer-driven callbacks, the inferred periods exactly match the configured periods, validating that the desired operating frequency is reached. Likewise, message-driven callbacks with a single predecessor exhibit periods that exactly match the input rate, demonstrating that periodicity propagates along processing chains as anticipated by schedulability analysis [7, 11, 41, 43].

Most of the PF models exhibit little or even zero jitter, which indicates excellent model fit and thus confirms that the callbacks are executing regularly, without much deviation from the expected periodic pattern. The CF jitter bounds are necessarily larger, given the conservative nature of the model, but still sufficiently low to confirm the intended timing behavior.

The executor partitioning is clearly reflected in the inferred jitter bounds. The front and rear LiDAR nodes each run in their own dedicated executor, and both their PF and CF models exhibit very low jitter, well below 1 ms. The same holds for

the BehaviorPlanner (BP) node, which is also isolated in a dedicated executor. The remaining nodes share the other two executors, and correspondingly exhibit larger jitter bounds because they experience more interference from co-located work. In particular, the jitter bounds of callbacks executed by the best-effort executor (*e.g.*, L2Map, ECSet) reflect the interference experienced by the executor thread itself.

A notable exception is the ECD callback: despite being served by a real-time executor, it exhibits a relatively large jitter bound even in the PF model, indicating weak periodicity. The reason is that it receives input from *two* callbacks (RGF and ECSet) with different periods (100 ms and 25 ms, respectively). Furthermore, the ECSet callback runs in the best-effort executor and thus is itself subject to significant jitter, as revealed by its CF model. This uncertainty propagates downstream to ECD, highlighting how a real-time callback can inherit poor timing regularity from an upstream best-effort callback.

Overall, LIME[IT] successfully recovered the periods of all periodic tasks and exposed timing consequences arising from the system's dataflow and executor configuration. While tracing the ROS 2 reference system, LIME[IT] exhibited low overhead: taken together, the injected eBPF probes and online model inference consumed less than 1% utilization of a single core.

B. Systematic Evaluation with Synthetic Workloads

To evaluate LIME[IT]'s accuracy against known ground-truth behavior, we ran experiments with Rust, C++, and C executors that were serving synthetic workloads composed of futures and callback functions. We chose this mix of languages and abstractions to demonstrate language- and framework-agnostic applicability. The workloads were generated as follows.

Workloads. Given a *target total utilization* $U^{\max}$, a *mean scaling factor* α, and a *target task-set size* n, we first randomly generated a maximum utilization vector $u_1^{\max}, \ldots, u_n^{\max}$ using the *ConvolutionalFixedSum* method [20] such that $\sum_i u_i^{\max} = U^{\max}$, and set each task's average utilization to $u_i^{\text{avg}} \triangleq \alpha \cdot u_i^{\max}$.

Next, each task's minimum inter-arrival time (or period) $T_i^{\min}$ was drawn log-uniformly at random from $[1\,\text{ms}, 100\,\text{ms}]$. The max. execution time $C_i^{\max}$ was then set to $C_i^{\max} \triangleq T_i^{\min} \cdot u_i^{\max}$.

For periodic tasks, the mean execution time is simply set to $C_i^{\text{avg}} \triangleq T_i^{\min} \cdot u_i^{\text{avg}}$. For sporadic tasks, we first selected a mean inter-arrival delay T_i^{avg} uniformly at random from $\left[T_i^{\min}, \min\left(\frac{0.9 C_i^{\max}}{u_i^{\text{avg}}}, 1000\,\text{ms}\right)\right]$, and then set $C_i^{\text{avg}} \triangleq T_i^{\text{avg}} \cdot u_i^{\text{avg}}$.

To vary execution times at runtime, we selected for each task a dispersion control parameter x_i uniformly at random from $[0.5, 5.5]$. Per-invocation execution times are then drawn from a *Normal* distribution with mean C_i^{avg} and standard deviation $(C_i^{\max} - C_i^{\text{avg}})/x_i$. Smaller x_i cause more variable execution times, whereas larger values result in a tighter clustering around C_i^{avg}. To ensure that execution times fall into $(0, C_i^{\max}]$, we discard and redraw any out-of-range samples (which introduces a slight distribution shift that is irrelevant for our purposes).

For sporadic tasks, we further randomized inter-arrival times by setting the maximum inter-arrival time to $T_i^{\max} \triangleq T_i^{\text{avg}} + (T_i^{\text{avg}} - T_i^{\min})$ and then drawing inter-arrival delays with rejection sampling from one of the following distributions:

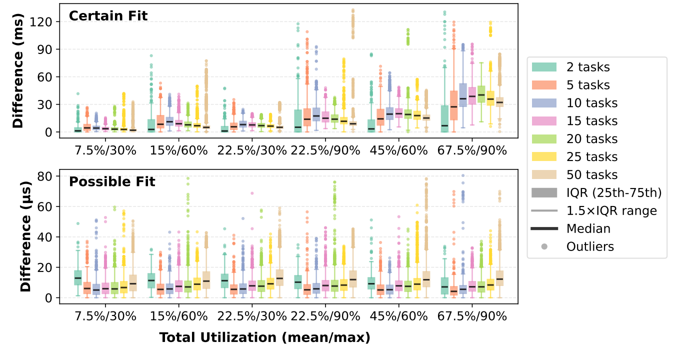

Fig. 4. Inter-quartile range (IQR) plots of the absolute difference in jitter bounds. Note the difference in scale of the two plots (ms vs. μs).

- *Uniform* across the range $[T_i^{\min}, T_i^{\max}]$,
- *Normal,* with mean T_i^{avg} and standard deviation $(T_i^{\max} - T_i^{\text{avg}})/y_i$, where y_i is again a dispersion control parameter drawn uniformly at random from $[0.5, 5.5]$,
- *Poisson* with mean T_i^{avg}, and
- *Mixed*, where each task randomly selects one of the above.

We varied $U^{\max} \in \{0.3, 0.6, 0.9\}$, $\alpha \in \{0.25, 0.75\}$, and $n \in \{2, 5, 10, 15, 20, 25, 50\}$, for a total of 42 diverse scenarios, from lightly to heavily loaded executors, with both few and many tasks. The ROS 2 reference system's number of payload callbacks (29) falls squarely into the considered range.

Periodic models (Rust). To assess LIME[IT]'s model inference accuracy for periodic workloads, we implemented a host program for periodic intra-thread tasks in Rust. The program consists of two threads: one runs a widely used executor from Rust's `futures` library [37], and the other periodically adds `async` functions (that realize the workload behavior described above) to the `futures` executor's queue of pending futures and records the ground-truth release times for later analysis.

For each of the 42 considered workload parameter combinations, we generated 100 workloads and traced each for 30 seconds with LIME[IT], for a total of 35 hours of traced execution. As a baseline, we inferred a point-release periodic model for each intra-thread task using the algorithm employed by LIME [10] based on the recorded ground-truth release times.

Across all 76,200 intra-thread tasks, LIME[IT] inferred the correct ground-truth period $T_i^{\min}$ for 100% of the possible-fit models, and for 99.73% of certain-fit models. In cases where the reported period was inexact, the median error was 0.006% of the ground-truth period, and the maximum was 0.033%. These results were obtained with the default candidate-selection factor of 1.25 (*i.e.*, X_3 in Listing 3); increasing it to $X_3 = 1.65$ raises the certain-fit period recovery rate to 100% for these workloads.

To assess the jitter bounds, we compared the possible-fit and certain-fit model parameters reported by LIME[IT] against the jitter bound of the point-release baseline model. Figure 4 summarizes the observed absolute differences in jitter bounds, grouped by the target total utilization and task count n.

For the certain-fit model, the jitter-bound difference increases with higher total utilization, but shows no clear dependence on n. This trend reflects the increased release-time uncertainty under higher load (*i.e.*, the executor suspends less often),

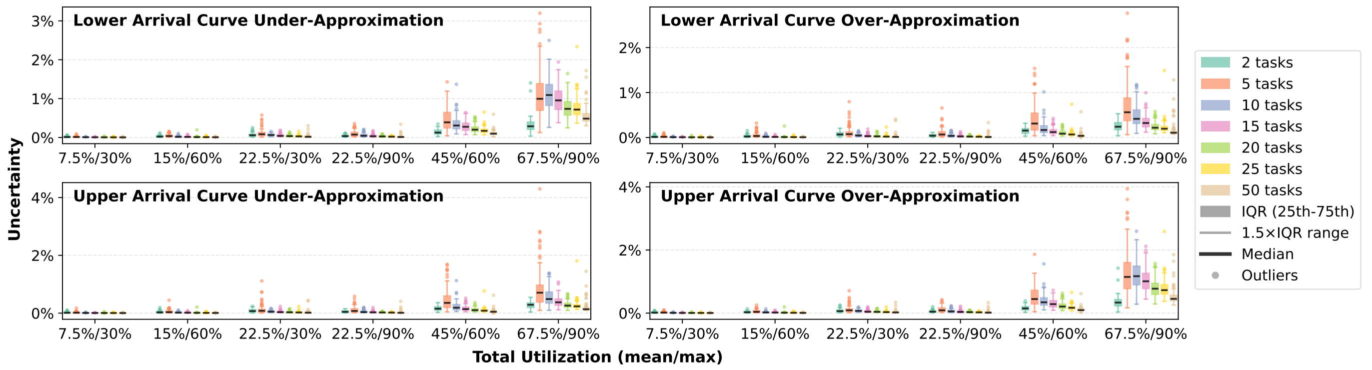

Fig. 5. IQR plot of arrival-curve uncertainty for the mixed arrival delay distribution. Plots for the other distributions exhibit similar trends and have been omitted.

which widens the conservative bound that certain-fit requires. In contrast, the possible-fit model does not exhibit a strong correlation: its jitter-bound difference w.r.t. the point-release baseline stays below 80.4 μs across all evaluated utilizations.

Overall, LIME$^{\text{IT}}$'s periodic model inference is highly accurate. The possible-fit model, in particular, exhibits excellent accuracy across the board, whereas the certain-fit model is inherently more conservative and hence works best for lightly-to-moderately loaded executors that frequently suspend.

Arrival curves (C++). Next, we evaluated LIME$^{\text{IT}}$'s lower- and upper-arrival curve approximation accuracy. To this end, we implemented another simple single-threaded executor in C++ that executes *closures* created with the C++ standard library's `std::async` API in its `std::launch::deferred` mode.

We implemented the described sporadic arrival behavior as self-triggered closures: during initialization, each intra-thread task is activated once, and then whenever a task executes, it determines the next time it should be activated (using one of the four inter-arrival delay distributions) and enqueues a copy of itself (*i.e.*, another closure) in the executor for execution at the appropriate time in the future. As before, the executor recorded the ground-truth release times as a baseline.

We generated 100 workloads for each of the 42 parameter combinations and each of the four inter-arrival distributions considered, for a total of 16,800 workloads. Each workload was traced for 30 seconds, for a total of 140 hours of runtime.

For each intra-thread task, as a baseline for comparison, we derived point-release upper and lower arrival curves based on the recorded exact release times using the algorithms employed by LIME [10]. Recall from Sec. IV that LIME$^{\text{IT}}$ extracts over- and under-approximations of both the lower and upper arrival curves. For all 304,800 evaluated sporadic tasks, the baseline lower and upper arrival curves were correctly bounded by the corresponding over- and under-approximations, confirming that LIME$^{\text{IT}}$'s over- and under-approximations are sound.

Next, we assessed the tightness of the approximations. To aggregate a notion of tightness across the large number of workloads, we numerically integrated the baselines and each of the over- and under-approximations, and compared the *areas under the curve*. The results are summarized in Fig. 5 for the mixed inter-arrival time distribution. The *uncertainty percentage* given in the plot is the difference in covered area, normalized by the area under the baseline curve. For example, 2% uncertainty in the plot entitled "lower arrival curve under-approximation" means that LIME$^{\text{IT}}$'s under-approximation of the lower arrival curve leaves 2% of the baseline's area under the curve uncovered (which is sound, but slightly pessimistic).

The general trends in Fig. 5 reveal that LIME$^{\text{IT}}$'s over- and under-approximations track the ground-truth baselines very closely in lightly-to-moderately loaded executors, and that more pessimism appears as executors become increasingly loaded. Again, the explanation is that LIME$^{\text{IT}}$'s approximations decrease in accuracy as executor suspensions become rarer since the underlying uncertain release windows derive in part from executor suspensions. Fig. 5 further shows that, for our workload generation method, pessimism peaks around 5 tasks. Overall, the results confirm that LIME$^{\text{IT}}$ tracks sporadic activity well, with only a moderate amount of uncertainty-induced pessimism.

Runtime overhead (C). Finally, to quantify the runtime costs of LIME$^{\text{IT}}$'s eBPF instrumentation and online model inference, we traced *Rössl* [5], a verified real-time executor implemented in C that invokes callbacks in response to UDP network packets.

We implemented ten distinct callbacks, each performing a deterministic 500 μs busy-wait, and varied the number of triggered callbacks from 1 to 10, and the period from 10 ms to 100 ms in steps of 10 ms. For each combination of callback count and period, we ran 10 repetitions, each lasting 30 seconds, for a total of 10 $\times$ 10 $\times$ 10 test runs lasting a combined 500 minutes. We measured both the eBPF overhead (induced in the executor) and the processor time used by LIME$^{\text{IT}}$ itself.

The results are shown in Fig. 6. The main factor affecting the magnitude of both the eBPF and LIME$^{\text{IT}}$ runtime overheads is the rate of eBPF events generated by the workload: for a fixed period, the overhead increases linearly with the number of intra-thread tasks, and for a fixed number of intra-thread tasks, the overhead increases proportionally to the rate at which intra-thread tasks are triggered. Both trends implicate the underlying rate of eBPF events as the main driver of overhead.

The overall runtime overhead is low: at the highest tested rate of 1,000 intra-thread task activations per second (10 tasks $\times$ 10 ms period), the eBPF overhead accounts for less than 1.1% processor utilization, and the runtime of LIME$^{\text{IT}}$ itself remains below about 1% utilization (both w.r.t. one core). LIME$^{\text{IT}}$ is

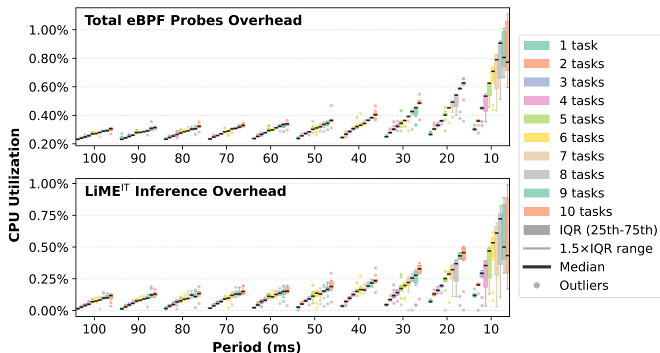

Fig. 6. Runtime overhead due to eBPF instrumentation and model inference.

efficient enough to monitor even highly loaded systems.

VI. Related Work

Real-time system *tracing* and *task-model inference* are two complementary techniques that have been extensively explored. We narrow our focus to two lines of work that are particularly relevant for comparison with our approach: 1) *framework-specific* techniques tailored to particular software stacks and 2) broadly applicable *framework-agnostic* techniques.

Within the *framework-specific* line of work, ROS 2 provides a representative example of an ecosystem with rich tracing and timing-analysis support. Core tracing facilities include *ros2_tracing* for low-overhead callback observation [4] and *Autoware_Perf* for performance analysis [26]. For ROS 2 processing chains, Blaß et al. [7] developed introspection support that automatically estimates timing parameters at runtime to adjust Linux's scheduling policies dynamically. As part of this effort, Stark proposed inference mechanisms [39] for *execution-time curves* [34], which generalize the notion of "maximum observed execution time" to multiple consecutive jobs. In more recent work, Abaza et al. [1] proposed a method for inferring *Directed-Acyclic Graph* (DAG) tasks by attaching eBPF's `uprobe` and `uretprobes` to several ROS 2 functions. Finally, Li et al. [25] proposed a unified trace analysis framework for the discovery and reporting of rare performance issues.

In the automotive domain, AUTOSAR Adaptive platforms such as EB corbos AdaptiveCore [16] and RTA-VRTE [17] provide tracing and monitoring for Linux-based high-performance ECUs. DDS-based middleware implementations, including RTI Connext and Eclipse Cyclone DDS, offer built-in monitoring facilities for distributed real-time applications. Similarly, industrial-automation frameworks such as OPC UA and PROFINET incorporate diagnostic and monitoring tools [2, 30].

Taken together, these framework-specific techniques offer deep, semantics-aware instrumentation tailored to particular middleware architectures. However, in contrast to LiME^IT, they typically exploit substantial framework-specific expertise, are specific to particular versions of the targeted middleware frameworks, and, in many cases, require access to proprietary tooling or source-code modifications. More importantly, apart from a few notable exceptions [1, 6, 7], tools in this space do not produce classic models of real-time execution that have been independently studied (*e.g.*, in schedulability analyses).

Within the framework-agnostic line, much work targets execution-time bounds, in particular, measurement-based *worst-case execution time* (WCET) analysis [12, 47]. As this line of work typically relies on manual annotations and does not consider arrival processes, it is largely orthogonal to the problems studied herein. In the following, we thus focus on model extractors that infer arrival-timing behavior.

Existing model extractors include *offline* learning- and regression-based techniques that infer periods and periodicity from traces without explicit release times [45], signal-processing approaches using quasi-maximum-likelihood estimators on incomplete and noisy measurements [38], and methods that mine periodic task sets by identifying jobs and estimating periods and response-time profiles [22]. Feng et al. [18] construct black-box models for real-time systems, and Brand et al. [8] employed hardware-assisted tracing for non-intrusive task detection and periodic task inference. Maggio et al. [28] proposed *rt-muse*, combining tracing and modeling of synthetic workloads on Linux to derive supply-bound functions, while de Oliveira et al. proposed automata-based analyses of the Linux scheduler [14] and its synchronization facilities [15].

The key difference to our work is that the above techniques provide only a thread-level view (Sec. II), whereas our approach automatically detects intra-thread tasks and thus provides an explicit *intra-thread view* of the system's timing behavior.

VII. Conclusion

We have introduced LiME^IT, the first framework-agnostic dynamic model extractor for intra-thread tasks such as callbacks, event handlers, coroutines, and futures. The primary technical innovations underlying LiME^IT are **(i)** a novel profiling-based method for automatic intra-thread task discovery and **(ii)** new online model-inference algorithms for uncertain release times, of which we provide a reference implementation [9, 48].

We applied LiME^IT to a ROS 2 reference system, where it successfully detected and modeled all timer- and message-driven callbacks. In an extensive evaluation with synthetic workloads implemented with Rust, C++, and C, LiME^IT achieved near-perfect accuracy for periodic tasks (over 99% period recovery) and closely approximated the point-release upper and lower arrival curves of sporadic tasks (within 2% pessimism in most cases). Jitter bounds remained within 100 μs of the baseline for possible-fit models and within tens of milliseconds for the more conservative certain-fit models. Runtime overhead stayed below 2.5% of a single core, even for 1,000 intra-thread task activations per second. Taken together, our results underline the broad applicability, high degree of automation, and accuracy of LiME^IT, which provides introspection capabilities not previously available for intra-thread tasks.

LiME^IT's main limitation is that it currently supports only single-threaded executors (with multiple executors per process possible), but not yet multithreaded executors (*i.e.*, thread pools) or parallel workloads (*e.g.*, OpenMP). Lifting these restrictions poses several interesting challenges for future work.

LiME^IT is freely available under an open-source license at:

https://lime.mpi-sws.org

REFERENCES

[1] H. Abaza, D. Roy, S. Fan, S. Saidi, and A. Motakis, "Trace-enabled timing model synthesis for ROS 2-based autonomous applications," in *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[2] P. N. America, "PROFINET system description: Technology and application," PROFIBUS & PROFINET International (PI), Tech. Rep., 2021, overview of PROFINET diagnostics, alarms, and maintenance information.

[3] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[4] C. Bédard, I. Lütkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.

[5] K. Bedarkar, L. Elbeheiry, M. Sammler, L. Gäher, B. B. Brandenburg, D. Dreyer, and D. Garg, "RefinedProsa: Connecting response-time analysis with C verification for interrupt-free schedulers," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 73–97, 2025.

[6] T. Blaß, D. Casini, I. Lütkebohle, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *Proceedings of the 42nd IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 41–53.

[7] T. Blaß, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic latency management for ROS 2: Benefits, challenges, and open problems," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 264–277.

[8] M. Brand, A. Mayer, and F. Slomka, "NITRO: Non-intrusive task detection and monitoring in hard real-time systems," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS '20)*, 2020, pp. 78–88.

[9] B. B. Brandenburg, "`rt-model-inference`: Real-time task model inference," Python library available on PyPI, https://pypi.org/project/rt-model-inference/, 2026.

[10] B. B. Brandenburg, C. Courtaud, F. Marković, and B. Ye, "LiME: The Linux real-time task model extractor," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 255–269.

[11] 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)*, vol. 133, 2019, pp. 6:1–6:23.

[12] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic timing analysis techniques for real-time systems," *Leibniz Transactions on Embedded Systems (LITES)*, pp. 1–60, 2019.

[13] A. C. De Melo, "The new Linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, no. 1, 2010.

[14] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Modeling the behavior of threads in the PREEMPT_RT Linux kernel using automata," *ACM SIGBED Review*, vol. 16, no. 3, pp. 63–68, 2019.

[15] D. B. de Oliveira, R. S. de Oliveira, and T. Cucinotta, "A thread synchronization model for the PREEMPT_RT linux kernel," *Journal of Systems Architecture*, vol. 107, p. 101729, 2020.

[16] Elektrobit Automotive GmbH, "EB corbos AdaptiveCore: Adaptive AUTOSAR middleware," https://www.elektrobit.com/products/ecu/eb-corbos/adaptivecore/, 2024, proprietary AUTOSAR Adaptive middleware for Linux-based automotive high-performance ECUs.

[17] ETAS GmbH, "RTA-VRTE: AUTOSAR adaptive microprocessor platform," https://www.etas.com/ww/en/products-services/vehicle-software-platform/rta-vrte/, 2024, commercial AUTOSAR Adaptive middleware for Linux-based automotive microprocessor platforms.

[18] T. H. Feng, L. Wang, W. Zheng, S. Kanajan, and S. A. Seshia, "Automatic model generation for black box real-time systems," in *Proceedings of the 2007 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2007, pp. 1–6.

[19] B. Gregg, "Flame graphs," https://www.brendangregg.com/flamegraphs.html, 2016, Visualization tool for profiled software, allowing the most frequent code-paths to be identified quickly.

[20] D. Griffin and R. I. Davis, "ConvolutionalFixedSum: Uniformly generating random values with a fixed sum subject to arbitrary constraints," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025.

[21] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.

[22] O. Iegorov, R. Torres, and S. Fischmeister, "Periodic task mining in embedded system traces," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 331–340.

[23] KUKA AG, "KUKA unveils iiQKA.OS2 operating system," https://www.kuka.com/, 2025, announcement of KUKA's proprietary iiQKA.OS2 operating system with virtual robot controller.

[24] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001.

[25] Y. Li, Y. Matsubara, H. Takada, S. Funahashi, and H. Kawashima, "Monitor and analyze rare ROS 2 performance issues with a unified tracing framework," in *Proceedings of the 6th World Symposium on Software Engineering (WSSE)*, 2024, pp. 95–104.

[26] Z. Li, A. Hasegawa, and T. Azumi, "Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications," *Journal of Systems Architecture*, vol. 123, 2022.

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[28] M. Maggio, J. Lelli, and E. Bini, "rt-muse: measuring real-time characteristics of execution platforms," *Real-Time Systems*, vol. 53, pp. 857–885, 2017.

[29] A. K.-L. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

[30] OPC Foundation, "OPC unified architecture, part 5: Information model (OPC 10000-5)," OPC Foundation, Tech. Rep., 2023, defines diagnostics nodes and monitoring constructs in the standard information model.

[31] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The ROSACE case study: From Simulink specification to multi/many-core execution," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 309–318.

[32] R. K. Pearson, *Mining imperfect data: Dealing with contamination and incomplete records*. SIAM, 2005.

[33] R. Pellizzoni and G. Lipari, "Feasibility analysis of real-time periodic tasks with offsets," *Real-Time Systems*, vol. 30, pp. 105–128, 2005.

[34] S. Quinton, M. Hanke, and R. Ernst, "Formal analysis of sporadic overload in real-time systems," in *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 515–520.

[35] K. R. Richter, "Compositional scheduling analysis using standard event models," Ph.D. dissertation, TU Braunschweig, Germany, 2005.

[36] ROS 2 Real-Time Working Group, "ROS 2 reference system,"

https://github.com/ros-realtime/reference-system, reference implementation of an autonomous driving perception pipeline for benchmarking real-time performance.

[37] Rust Networking Working Group, "Zero-cost asynchronous programming in Rust," https://rust-lang.github.io/futures-rs/, 2018.

[38] N. D. Sidiropoulos, A. Swami, and B. M. Sadler, "Quasi-ML period estimation from incomplete timing data," *IEEE Transactions on Signal Processing*, vol. 53, no. 2, pp. 733–739, 2005.

[39] T. Stark, "Real-time execution management in the ROS 2 framework," Ph.D. dissertation, Saarland University, Germany, 2022.

[40] I. Stoica, H. M. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of the 17th Real-Time Systems Symposium (RTSS)*, 1996, pp. 288–299.

[41] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ROS 2 executors," in *Proceedings of the 41st Real-Time Systems Symposium (RTSS)*, 2020, pp. 231–243.

[42] H. Teper, D. Kuhse, M. Günzel, G. von der Brüggen, F. Howar, and J.-J. Chen, "Thread carefully: Preventing starvation in the ROS 2 multithreaded executor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3588–3599, 2024.

[43] H. Teper, O. Bell, M. Günzel, C. Gill, and J. Chen, "Reconciling ROS 2 with classical real-time scheduling of periodic tasks," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 177–189.

[44] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2000, pp. 101–104.

[45] Ş. Vădineanu and M. Nasri, "Robust and accurate regression-based techniques for period inference in real-time systems," *Real-Time Systems*, vol. 58, no. 3, pp. 313–357, 2022.

[46] Waymo, "Behind the innovation: AI & ML at Waymo," https://waymo.com/blog/2024/10/ai-and-ml-at-waymo, October 2024, blog post describing Waymo's proprietary AI middleware and foundation model architecture.

[47] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[48] B. Ye, F. Marković, and B. Brandenburg, "Python reference implementation of LiME's model inference algorithms," software artifact, Zenodo, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19355033

[49] B. Zhou, J. Xu, X. Liu, Z. Zhu, X. Chen, Z. Liu, J. Ye, C. Jiang, T. Guo, Y. Zhang, Y. Zhang, L. Liu, K. Sun, Y. Wu, and X. Zhang, "The eBPF runtime in the Linux kernel," *arXiv preprint arXiv:2410.00026*, 2024.