LiME: The Linux Real-Time Task Model Extractor

Björn B. Brandenburg* Cédric Courtaud† Filip Marković‡ Bite Ye*

*Max Planck Institute for Software Systems, Germany
†Huawei Technologies, Paris Research Center, France
‡University of Southampton, United Kingdom

***Abstract*—We present LiME, a novel dynamic real-time task model extractor. LiME observes the temporal behavior of Linux real-time threads and automatically maps the observed activity to established real-time task models: sporadic and periodic tasks, upper and lower arrival curves, cumulative execution-time curves, and two self-suspension models (dynamic and segmented). LiME runs on unmodified Linux kernels and requires neither knowledge of real-time theory nor familiarity with Linux internals to be used effectively. An extensive evaluation shows LiME to achieve very high inference accuracy—in particular 100% accuracy for common automotive periods—with low kernel overhead, low latency impact, and low processor utilization (at best-effort priority).**

I. Introduction

The recent acceptance of the PREEMPT_RT patch into the mainline Linux kernel, after 20 years of development [58], has made official what has long been true: Linux is a major platform for hosting modern, complex real-time workloads across various industries. Notable examples feature demanding application domains such as *automotive systems* [52], *autonomous vehicles* [30, 31], *unmanned aerial vehicles* (UAVs) [19, 27, 35], and *spacecraft* [37], in particular crewed rockets [59], NASA's Mars helicopter *Ingenuity* [57], and tens of thousands of Linux systems deployed in orbit as part of *Starlink* constellations [54].

As noted recently by Erik Vallow, a representative of the RTOS vendor *LYNX Software Technologies*, in a retrospective on the evolving role of Linux in the aerospace and defense industries [56]: *"Linux has become a formidable contender in safety-critical systems due to advancements in real-time capabilities and reliability. [. . .] Linux is increasingly capable of meeting the demands of real-time applications on its own, reducing the need for a separate RTOS in some cases."* In addition, the availability of drivers for high-performance GPUs is increasingly a factor favoring the consolidation of AI-enabled or otherwise GPU-accelerated real-time workloads on Linux.

However, while the popularity of Linux as a versatile and feature-rich RTOS has soared in the real-time systems industry, there is a growing disconnect with the analytical foundations studied in the scientific literature on real-time systems. Rooted in abstract system models and high-level mathematical descriptions of workloads, state-of-the-art methods for establishing temporal guarantees are far removed from the engineering realities of a low-level embedded Linux environment.

It stands to reason, then, that only a diminishingly small fraction of the many real-time workloads deployed on Linux over the past decade have been modeled and formally evaluated using published schedulability analyses. A major contributor to this disconnect is the lack of tool support: without automated system introspection tools, engineers interested in formally characterizing the timing properties of a real-time workload running on Linux require *dual expertise* in both Linux implementation details, particularly the kernel's low-level tracing facilities and system-call interface, *and* state-of-the-art temporal modeling and analysis techniques. This, along with the associated *manual* effort that would be required (and not just once, but repeatedly as systems evolve to meet changing requirements), presents a formidable barrier to the widespread adoption of state-of-the-art real-time analysis techniques in a Linux context.

Could the schedulability analysis of Linux workloads be automated and thus made *easily* accessible to non-experts? Motivated by this question, we address the first, fundamental problem that precedes any practical analysis: the *automated* modeling of real-time tasks deployed on *unmodified* Linux kernels.

While applications developed using higher-level *model-first approaches* [9, 24, 44], or using programming languages with explicit timing semantics [7, 10, 11, 15, 26, 28, 42, 43], can certainly be *compiled down* to Linux binaries and executed efficiently (*i.e.*, model-driven engineering), the converse is far from obvious: *Is it possible to extract high-level temporal models of running Linux threads suitable for schedulability analysis simply by observing their low-level runtime behavior?* Is it possible to do this for *black-box* threads (*i.e.*, without access to source code)? Fully *automatically*, without user guidance, annotations, or specifications of intended timing? And can it be done *in situ* on a target embedded platform without unduly perturbing the timing of the real-time threads?

This paper. We show that the answer to each of these questions is 'yes' by presenting LiME, the **Li**nux real-time task **M**odel **E**xtractor (Sec. III). LiME is a dynamic introspection tool that maps sequences of low-level thread-kernel interactions (Sec. II-A), observed via the kernel's *eBPF* tracing facility, to task models from the real-time scheduling literature (Sec. II-B).

Fig. 1 illustrates the entire pipeline: Given an arbitrary black-box workload (in Fig. 1, a thread implementing periodic activations with `clock_nanosleep`), LiME uses eBPF to observe key scheduling events and system calls throughout the whole system (Inset 1). After reconstructing the timeline for each target thread (Inset 2), LiME identifies job boundaries based on its builtin understanding of Linux system call

*†‡ Authors are listed in alphabetical order. †‡ This work was carried out while affiliated with the Max Planck Institute for Software Systems, Germany.

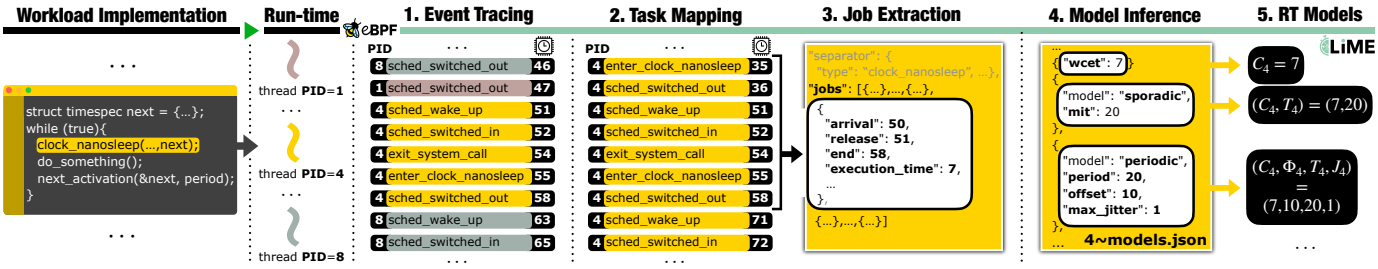


Fig. 1: LiME's model-inference pipeline, illustrated for a periodic thread with PID = 4 (colored yellow).

semantics (Inset 3). From the stream of jobs, LiME extracts the building blocks of real-time task models (Inset 4). Finally, the user is presented with instances of classic task models (such as a periodic task with jitter) that *conservatively* describe all observed thread behavior (Inset 5). Importantly, LiME can run the entire pipeline *fully automatically* (without annotations or user guidance) and *in situ* (online, with bounded memory).

LiME is highly accurate (Sec. V-A), easy to deploy, runs on unmodified Linux kernels, and requires no knowledge of real-time theory, Linux internals, or eBPF to be used effectively. The models it extracts are useful in a number of ways: **1)** for *behavior validation*, wherein LiME confirms that a thread exhibits the intended timing (Secs. V-A and VI-B); **2)** for *system exploration*, wherein LiME discovers previously unknown or unexpected kernel and application threads and characterizes their impact (Sec. VI-A); **3)** for *debugging*, wherein peculiarities of the extracted models point to subtle bugs in application code (Sec. VI-B); and **4)** for *continuous monitoring*, wherein LiME runs alongside an application for prolonged times with low overheads (Secs. V-B to V-D) to check that the target's timing does not drift over time (Sec. VI-B).

Thanks to the low overhead of eBPF, LiME's online model-inference algorithms (Sec. IV), and its efficient implementation in the *Rust* programming language, LiME is lightweight enough to run alongside the monitored target (Sec. V-B) without placing a significant burden on application timing (Sec. V-C).

We will revisit Fig. 1 throughout the rest of this paper as we explain each stage of LiME's model-inference pipeline in detail. To begin, we take a closer look at the fundamental disconnect that makes model inference challenging in Linux.

II. Linux vs. Real-Time Task Models

Conceptually, the core challenge addressed by LiME is the large *semantic gap* between how Linux processes operate compared to established models of real-time computation. To illustrate the difficulties involved in bridging this gap, we begin with a high-level overview of Linux's process abstraction and then review the real-time task models most relevant to LiME.

A. Linux Thread Behavior

The principal schedulable entity in Linux is a *thread*,[1] representing an ongoing computation defined by a register context and a call stack. A *process* contains one or more threads that share the process's resources (*e.g.*, the address space, open file descriptors, security capabilities, *etc.*).

At any point in time, each thread in the system is either *runnable* (*i.e.*, either currently executing or ready to execute) or *blocked* (*i.e.*, in some state in which the thread's computation cannot proceed until some thread-external event takes place). A thread is said to *block* (or *suspend*) when it ceases to be runnable, and to *activate* (or *resume*) when it becomes runnable again. Conceptually, the scheduler always selects a subset of the runnable threads to dispatch on the available processors.

Linux provides a sophisticated and continuously evolving hierarchy of thread schedulers that implement several real-time (SCHED_RR, SCHED_FIFO, SCHED_DEADLINE) and best-effort policies (SCHED_OTHER, SCHED_BATCH); however, as we focus on intrinsic thread behavior, which is largely independent of the scheduling policy a thread is managed by, we will not delve further into Linux scheduling details.

In Linux, thread behavior is *fundamentally unconstrained*: threads execute arbitrary code and may call arbitrary system calls in arbitrary order at arbitrary points in their execution. Regular or repeating thread behavior emerges by choice of the programmer, not due to kernel-enforced constraints or thread-model properties. Any formal model of "real-time thread behavior" in Linux suitable for schedulability analysis thus inherently targets only a narrow subset of permissible thread behavior.

Fundamentally, at any point in time, a thread can be involved in one of three activities: it may **(i)** compute in user space using only the processor and memory already mapped into the thread's address space; **(ii)** invoke system calls (or trigger exceptions) that cause some computation to take place in kernel space *without blocking the thread* (*e.g.*, the `getpid()` system call has no timing implications other than taking a few cycles to complete); and **(iii)** invoke (potentially) *blocking* system calls (or trigger exceptions) that change a thread's status to "blocked" and prevent it from being scheduled for some time. Activities in categories (i) and (ii) affect only a thread's execution time, but otherwise do not affect its timing behavior. Category (iii), however, is critical to understanding thread behavior.

Blocking system call semantics arise for many reasons. For example, attempting to read data from a socket when the socket's buffer is empty will (by default) block the calling thread until data becomes available (*e.g.*, in the case of a datagram socket, until the next packet arrives and its payload is placed into the socket's buffer). Similarly, a thread programming a timer to be activated later (*e.g.*, via `clock_nanosleep()`)

[1] Linux documentation interchangeably refers to threads and processes also as "tasks." To avoid confusion, we exclusively use "thread" when discussing Linux's OS abstraction and reserve "task" to refer to real-time model entities.

will block until the timer expires. Thread synchronization APIs (*e.g.*, *pthreads*, POSIX, or SYSTEM V) also come naturally with blocking semantics (*e.g.*, semaphores, barriers, *etc.*). A thread might also become involuntarily blocked if it triggers a page fault that requires some time to be serviced (*e.g.*, demand paging, copy-on-write). Of particular interest to real-time systems are blocking system calls with *timeout semantics* (*e.g.*, `sig_timed_wait()`, which cause a thread to block until it either receives a signal or a specified timeout expires).

A comprehensive review of all blocking system calls in Linux is necessarily beyond the scope of this paper. Suffice to say, there are *many* ways in which Linux threads can block, and many more reasons for why they might do so.

Abstracting from such details for now, we observe an important high-level pattern: any well-behaved Linux thread awaiting a *future event* or the *passage of time* will block in *some* way. Furthermore, this behavior is unambiguous and readily observable from the kernel's vantage point. However, it is also important to realize that threads do not *necessarily* block when invoking system calls with blocking semantics, precisely when the event in question has already occurred (*e.g.*, when reading from a socket with a non-empty buffer, when acquiring a semaphore that is available, or when requesting a timer for a point in time already in the past).

In summary, a Linux thread alternates between intervals in which it is runnable and intervals in which it is blocked. This sequence of intervals may follow any pattern, or no pattern at all. The transition from *runnable* to *blocked* is always marked by a blocking system call (or exception handler). Intervals in which a thread is runnable may contain any number of system call invocations, including system calls with blocking semantics if the particular invocation did not block the thread.

B. Models of Recurrent Real-Time Computation

In the five decades since Liu and Layland's pioneering work [39], a plethora of increasingly nuanced and detailed real-time task models have been proposed and analyzed. In the following, we restrict our attention to widely used, fundamental models of *sequential* tasks (since we seek to model sequential Linux threads) and focus on those for which we have implemented support in LITMUS^{RT}.

Under all considered models, the system is comprised of n sequential real-time *tasks* $\tau_1, \ldots, \tau_n$, which matches the fact that a Linux system consists of a finite number of sequential threads. Each task τ_i is repeatedly activated, and each activation is called a *job*, so that each task is simply a stream of jobs. We let $\tau_{i,j}$ denote the j^{th} job of task τ_i. Each job $\tau_{i,j}$ is defined by (at least) two parameters $(r_{i,j}, c_{i,j})$: its *release time* $r_{i,j}$, which is the earliest time it becomes available for execution, and its *execution time* (or *cost*) $c_{i,j}$. Task models differ in how they constrain each task's stream of jobs, and for some models additional job parameters will be introduced in the following.

Sporadic. The *sporadic task model* [41], wherein each task τ_i is characterized by two parameters (C_i, T_i), is the most simple task model applicable to Linux. The *worst-case execution time* (WCET) bound C_i simply limits the maximum cost of any job: $\forall j,\ c_{i,j} \leq C_i$. Conversely, the *minimum job separation* T_i lower-bounds the minimum distance between any two job releases: $\forall j,\ r_{i,j} + T_i \leq r_{i,j+1}$. The sporadic task model is best suited to describing tasks that are infrequently triggered.

Arrival curves. If tasks can be subject to *bursty* activations, then the classic sporadic model's reliance on a scalar separation parameter T_i is often too limiting. In particular, if multiple jobs can arrive simultaneously, then the sporadic task model degenerates to $T_i = 0$, which prevents meaningful analysis.

Arrival curves [29, 50], inspired by network calculus [34, 53], are a more general way of describing arrival processes that can express bursts and simultaneous arrivals without loss of analysis accuracy. Specifically, an *upper arrival curve* $\alpha_i^+(\Delta)$ bounds the maximum number of jobs of task τ_i released in any given interval: $\forall \Delta,\ \forall t,\ |\{\tau_{i,j} \mid t \leq r_{i,j} < t + \Delta\}| \leq \alpha_i^+(\Delta)$.

Conversely, a *lower arrival curve* $\alpha_i^-(\Delta)$ bounds the minimum number of jobs of task τ_i released in any given interval: $\forall \Delta,\ \forall t,\ \alpha_i^-(\Delta) \leq |\{\tau_{i,j} \mid t \leq r_{i,j} < t + \Delta\}|$. Together, α_i^+ and α_i^- provide a good characterization of how frequently job releases must be expected, irrespective of how bursty a task is.

Periodic. The best-known real-time task model is Liu and Layland's original *periodic task model* [39], in which each task is characterized by a tuple (C_i, T_i) as in the (later) sporadic task model. However, in the periodic model, all release times are pre-determined and the separation is required to be *exact*: $\forall j,\ r_{i,j} = (j-1) \cdot T_i$, which implies $\forall j,\ r_{i,j} + T_i = r_{i,j+1}$. This, unfortunately, is unobtainable in Linux and similar systems. As the processing of activations invariably involves kernel code, there is always some amount of variably-timed overhead involved that affects the separation of job releases. Furthermore, there is no guarantee that the first release occurs at some integer multiple of T_i. The Liu and Layland model is thus inapplicable.

Nonetheless, periodic workloads are common in practice and, for reasons of analytical precision, it is undesirable to over-approximate periodic tasks as sporadic tasks. We thus turn to a more expressive variant of the periodic task model in which each task τ_i is described by four parameters (C_i, Φ_i, T_i, J_i), where the *offset* $\Phi_i \geq 0$ [46] describes the time of the task's first activation and the *maximum release jitter* $J_i \geq 0$ [3] bounds the maximum deviation from ideal periodic behavior.

To state the model's constraints precisely, we first need to introduce two new job parameters: a job $\tau_{i,j}$'s *arrival time* $a_{i,j}$ describes the *intended* release time (*i.e.*, when the job should have been released in a hypothetical overhead-free system), and its *jitter* $\lambda_{i,j} = r_{i,j} - a_{i,j}$ is the deviation from this ideal.

The *jitter-aware periodic model* requires exactly spaced arrivals, but allows releases to deviate by a bounded amount:

$$\forall j,\ a_{i,j} = \Phi_i + (j-1) \cdot T_i \ \wedge\ 0 \leq \lambda_{i,j} \leq J_i \qquad (1)$$

As shown later (Sec. V-A) the flexibility afforded by the offset and jitter parameters makes the model practical in our setting.

Self-suspension. So far, we have not discussed whether jobs execute as a whole from start to finish or whether they can intermittently *suspend* their execution. The models described so far (implicitly) assume the absence of self-suspension, meaning

once released a job can execute whenever selected by the scheduler. As already discussed, in a complex OS such as Linux, this is not always a practical assumption: threads can call blocking system calls at any time (*e.g.*, to wait for I/O).

Self-suspension has major implications for schedulability analysis [18], and thus must be modeled when it occurs. The simplest approach is the *dynamic self-suspension model*, which can be integrated with all models considered so far. It simply adds a scalar task parameter S_i that bounds the cumulative self-suspension duration exhibited by any job of task τ_i.

To state the semantics of the parameter S_i precisely, we once more must refine our job model. Let $m_{i,j} \geq 0$ denote the number of self-suspensions exhibited by a job $\tau_{i,j}$, and let the vector $\langle (s_{i,j,0}, c_{i,j,0}); (s_{i,j,1}, c_{i,j,1}); \ldots; (s_{i,j,m_j}, c_{i,j,m_{i,j}}) \rangle$ describe $\tau_{i,j}$'s sequence of alternating suspensions and execution *segments*, such that $c_{i,j} = \sum_{k=0}^{m_{i,j}} c_{i,j,k}$. That is, each tuple $(s_{i,j,k}, c_{i,j,k})$ describes a segment of $\tau_{i,j}$ comprising $c_{i,j,k} \geq 0$ time units of execution preceded by a self-suspension lasting $s_{i,j,k} \geq 0$ time units. (The "suspension" before the first computation segment $c_{i,j,0}$ is equivalent to the release jitter $\lambda_{i,j}$ [17].) Given this elaborated job model, S_i simply bounds the total self-suspension time: $\forall j,\ \sum_{k=0}^{m_{i,j}} s_{i,j,k} \leq S_i$.

The dynamic self-suspension model is flexible, but limits analysis accuracy [18]. A more accurate model is the *segmented self-suspension model*, wherein each job's segment structure is fixed at the task level [18]. However, a single permissible segment vector per task is too restrictive in a Linux environment, where (some) tasks tend to exhibit more dynamic behavior.

Inspired by an earlier hybrid model [60], we therefore consider a *bag of segment vectors* (BOS) model, in which each task is associated with a set of possible segment vectors. Let V_i denote a set of segment vectors for task τ_i, and let $\vec{S_l} \in V_i$ denote a vector of per-segment upper bounds of the form $\vec{S_l} = \langle (S_{i,l,0}, C_{i,l,0}); (S_{i,l,1}, C_{i,l,1}); \ldots \rangle$. The BOS model constraint can then be precisely stated as follows: $\forall j,\ \exists \vec{S_l} \in V_i$ s.th. $|\vec{S_l}| = m_{i,j} + 1 \wedge \forall k \in \{0, \ldots, m_{i,j}\}, s_{i,j,k} \leq S_{i,l,k} \wedge c_{i,j,k} \leq C_{i,l,k}$, where $|\vec{S_l}|$ denotes the length of vector $\vec{S_l}$.

WCET(ι). Finally, let us refine the (scalar) WCET bound C_i. As stated above for the sporadic task model, C_i applies equally to all jobs. In practice, however, it is often not the case that all jobs are of similar cost. For example, the first job in a burst may be more expensive to compute than subsequent jobs in the same burst due to warm-up effects. Furthermore, some tasks are deliberately programmed to carry out certain particularly expensive operations only infrequently.

For such workloads, it is beneficial to adopt a *cumulative execution-time curve* $\mathrm{WCET}_i(\iota)$ to express bounds on the joint execution time of multiple consecutive jobs [49]. Similarly to arrival curves, $\mathrm{WCET}_i(\iota)$ bounds the total cost of any ι consecutive jobs: $\forall \iota, \forall j, \sum_{k=j}^{j+\iota} c_{i,k} \leq \mathrm{WCET}_i(\iota)$. Clearly, $C_i = \mathrm{WCET}_i(1)$, but usually $\iota \cdot C_i > \mathrm{WCET}_i(\iota)$. This allows for a more accurate characterization of the joint impact of a task's jobs across some longer interval (*e.g.*, in response-time analysis), as previously observed in the context of ROS [8].

C. Semantic Gap

There is a large conceptual gulf between Linux thread behavior and established real-time task models. On the one hand, Linux threads are unconstrained and can behave in arbitrary, irregular, and unrepeating ways. On the other hand, real-time task models fundamentally assume recurrent task activations and seek to expose as much regularity as possible in the interest of analysis accuracy. It is thus not feasible to capture *any possible* Linux thread behavior with these models.

Nevertheless, *useful* software is usually *not* arbitrary in its behavior. In particular, well-engineered real-time workloads tend to exhibit highly regular behavior even on Linux. It is thus feasible to describe *well-behaved* Linux real-time threads using the models discussed in Sec. II-B.

However, even well-behaved Linux threads are challenging to model automatically, due to some fundamental obstacles. First, *Linux is jobless*: All task models discussed in this section inherently revolve around the notion of *jobs*, but thread execution under Linux is unstructured and often involves many different potentially blocking system calls. Even if tasks behave in a recurrent fashion, it is not always obvious where one job "ends" and another one "starts." Defining and efficiently detecting reliable *job separators* is thus a core challenge when modeling Linux threads as real-time tasks.

Second, *thread intention is unknown*: In the absence of annotations and without resorting to a static analysis of a thread's machine code (or its source code if available), an observer is generally unaware of a thread's intended timing behavior. This creates considerable ambiguity. For example, even humans struggle to reliably distinguish the trace of a jitter-affected periodic task from a truly sporadic one.

Third, *many threads are event-driven*: Even a static analysis cannot tell us the intended timing behavior of a thread that is activated by incoming UDP packets—the sender and the network determine the timing behavior, not the recipient's source code. Models of event-driven tasks thus always need to be understood in the context of the observed environment.

III. LiME: Design and Implementation

We designed and built LiME to *automatically* bridge the semantic gap between low-level thread behavior and high-level task models *without* relying on user guidance, annotations, or access to the source code of running threads.

As shown in Fig. 1, LiME extracts task models from observed thread behavior via a four-stage pipeline. First, to observe thread behavior, LiME's kernel-level tracing component records events at predefined tracepoints in the Linux kernel (Inset 1). The resulting stream of thread events is then mapped to individual tasks (Inset 2). After the stream of events has been split into individual per-task streams, each such per-task stream of events is distilled into an intermediate representation that provides a more succinct summary of thread behavior, including separators marking probable job boundaries (Inset 3). In the fourth and last stage, LiME's model inference algorithms continuously update the parameters of *each* model reviewed in Sec. II-B based on the stream of jobs identified in the

prior stage (Inset 4). Finally, when LIME terminates, instances of each applicable real-time task model are emitted for each observed task (Inset 5). We discuss the first three stages next and dedicate Sec. IV to the last stage.

It is important to acknowledge that LIME is inherently *measurement-based* and *heuristic-driven*, as is common in the context of Linux. As such, it cannot provide hard guarantees on the correctness of the models that it extracts. Rather, it should be seen as making a best-effort attempt at *interpreting* real-world thread behavior through the lens of real-time scheduling theory, and it may fail to offer a reasonable interpretation if confronted with pathological workloads. That said, LIME goes to great lengths to ensure that it extracts only task models that *conservatively* explain *all* observed activity (as we revisit in greater detail in Sec. IV-A) and as a result achieves remarkable accuracy for well-behaved threads (Sec. V-A).

Operating modes. LIME can be used in three separate modes. In *offline mode*, the pipeline stops after the second stage (Inset 2 in Fig. 1). In this mode, LIME simply records the per-task stream of events for all monitored threads for later processing. Conversely, in *online mode*, LIME runs the full pipeline, processes all events *in situ*, and outputs *only* the extracted task models (Inset 5). In particular, in online mode, LIME consumes only a constant amount of memory per task, irrespective of the monitoring duration. Finally, in *replay mode*, LIME runs the third and fourth stages on a trace previously recorded by its offline mode. In replay mode, LIME can optionally also output the lists of jobs corresponding to Inset 3 of Fig. 1.

The main advantage of LIME's offline mode is that later offline analyses can be arbitrarily complex and resource-consuming (as they can be carried out on a different machine). Its major limitation is the large amount of disk space required to store longer traces (Sec. V-A), which can quickly exhaust the storage capacities of typical embedded platforms.

LIME's online mode is specifically designed so that LIME can be deployed for arbitrary durations alongside the monitored system, even on storage- and memory-constrained platforms. Its downside is that model-extraction overheads are incurred on the target platform, but fortunately these overheads are low (Sec. V-B). In the following, we focus mainly on LIME's online mode, as it is the more challenging use case to support.

Event tracing. LIME relies on Linux's eBPF tracing facility, which allows (privileged) user processes to upload sandboxed programs (or *probes*) to kernel space without endangering the kernel's integrity. First introduced more than 10 years ago, eBPF is by now a mature technology with a rich feature set. We focus here only on the parts most relevant to LIME.

Depending on the kernel version, LIME installs up to 65 eBPF probes, implemented in about 1,600 lines of eBPF's C dialect, to monitor context switches, thread-state changes (*e.g.*, threads becoming runnable), and specific system calls. Each time a probe is triggered, it checks whether the event is relevant to LIME (*e.g.*, if the triggering thread is being traced), and if so, places the event in a shared ring buffer accessible to user space (a standard component provided by the eBPF library).

LIME consumes events from this ring buffer. Each event has a timestamp, a thread ID, a type, and a body with additional attributes depending on the type of event. Timestamps are provided in nanosecond resolution based on the kernel's internal clock (known as `CLOCK_MONOTONIC` in user space). The first step in processing an event is to associate it with a task.

Task mapping. The thread-to-task mapping is somewhat more nuanced than one might initially assume. In particular, it is not a 1-to-1 mapping: a thread can be mapped to multiple tasks throughout its lifetime to reflect changing behavior and/or gaps in observation. We collectively refer to changes in the thread-to-task mapping as *task transition events*.

By default, a task transition event occurs if major scheduling-related thread parameters change (*e.g.*, processor affinity, scheduling policy, priority). The rationale is that such a change typically indicates a major shift in operating mode, which is more accurately modeled as one task leaving and another task joining the system. A practical benefit is that it also tends to automatically separate execution phases that are uninteresting from the point of view of longterm behavioral modeling.

For example, programs tend to parse flags, initialize their state, load additional resources, set up logging facilities, *etc.* before switching to "real-time mode" by setting their scheduling policy to a real-time policy. Conversely, a switch from a real-time priority to SCHED_OTHER typically indicates the end of "real-time mode." By default, LIME will record such initialization and termination phases as separate tasks, which can then be trivially discarded during subsequent processing.

Importantly, it is possible to override LIME's default heuristic so that it does not count priority and affinity changes as task transition events, which is necessary when faced with a user-space scheduling framework (as we revisit in Sec. VI-B).

Job separation. For each task, the incoming stream of events is reduced to an intermediate representation that tracks the intervals in which the corresponding thread is suspended, running, or preempted, in addition to other information useful to the model extractors. Most importantly, it identifies *job separators* that demarcate the end of one job and the beginning of the next. As motivated in Sec. II, splitting a thread's execution history into a sequence of logically distinct jobs is a key step in LIME's extraction pipeline and a major challenge.

LIME identifies probable job boundaries based on two insights: **(i)** a job corresponds conceptually to one *activation* of a task, and such an activation is virtually always preceded by a *potentially* blocking system call since the task must wait for the next activation if none has occurred yet; and **(ii)** proper, well-behaved real-time threads tend to execute the same system calls over and over since they react to activations of a particular kind (*e.g.*, it is unlikely that one job is triggered by timer, the next by a UDP packet, and the third by a signal).

Therefore, the first step is to efficiently pick out potentially blocking system calls from the stream of events. To this end, LIME uses regular expressions (implemented as simple finite automata) to match *signatures* that uniquely identify particular system calls in the per-task streams of events.

For example, consider the periodic task shown in Fig. 1. In this example, the useful work is done by the `do_something()` function and the desired period is enacted by invoking `clock_nanosleep()`. In every iteration of the `while` loop, the thread generates the characteristic sequence of events highlighted in Inset 1 of Fig. 1, which is the signature specific to the *job separator* `clock_nanosleep()`. Whenever the thread has no more work to do, it enters the system call, which causes it to block in state `TASK_INTERRUPTIBLE` (*i.e.*, the thread is no longer runnable) until a new period begins, at which point the thread resumes and exits the system call.

In total, LIME detects more than 20 different job separators, including all major system calls for device I/O, networking, IPC, signal handling, and timers. Great care is taken by the eBPF probes to exclude non-blocking variants of these system calls. Two job separators warrant additional attention.

First, the *suspension separator* declares any suspension to indicate a job boundary. This separator is of particular interest because, on the one hand, the resulting models, by definition, consist exclusively of *non-self-suspending* jobs, which is highly attractive from a schedulability analysis perspective. On the other hand, it is applicable even to kernel threads that do not invoke any system calls (as discussed in Sec. V-A).

Second, the `sched_yield` separator provides support for tasks designed for the SCHED_DEADLINE policy (and is exclusive to that policy). A thread scheduled by this policy can call `sched_yield` to relinquish any remaining budget to SCHED_DEADLINE's slack reclamation mechanism, which is an unambiguous indicator that a job has finished.

Conceptually, job separators transform the per-task stream of events into *per-separator* streams of *jobs* (*i.e.*, there is a distinct stream of jobs for each task and each type of separator). Each such stream of jobs is then consumed by the model extractors. There is one model extractor instance for each job stream and each supported type of model, as we discuss next.

IV. Model Inference

We now focus on LIME's algorithmic core, namely its model extractors. To begin, we clarify the goal of the extraction stage.

A. Conservative Model Extraction

We adopt *conservative extraction* as LIME's correctness specification. As a measurement-based tool, LIME is inherently limited to describing the thread activity it has observed. Beyond this fundamental constraint, however, we require that the models it produces faithfully describe *all* observations.

More precisely, recall from Sec. II-A that we can model all Linux thread activity as an alternating sequence of ready and suspension intervals, with sequences of system calls interspersed in ready intervals. Let $\mathbb{T}$ denote the set of all possible thread traces, and let $\mathcal{M}$ be an instance of a sequential real-time task model (*e.g.*, a sporadic task τ_i with concretely defined parameters C_i and T_i). We can understand $\mathcal{M}$ to describe a restricted subset $\mathit{traces}(\mathcal{M}) \subset \mathbb{T}$, which contains all thread traces compliant with $\mathcal{M}$'s parameters and semantics (*e.g.*, all thread traces in which the start times of ready intervals are at least T_i time units apart and the thread never executes for longer than C_i time units during any ready interval).

In this view, LIME can be seen as function in the reverse direction: it maps an observed trace $t \in \mathbb{T}$ to a set of inferred models $\textsc{Lime}(t) = \{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$. The *conservative extraction* criterion then requires:

$$\forall t \in \mathbb{T},\ \forall \mathcal{M}_i \in \textsc{Lime}(t),\ t \in \mathit{traces}(\mathcal{M}_i).$$

That is, if LIME claims a model instance $\mathcal{M}_i$ to explain the observed behavior t, then the observed trace t must indeed be in $\mathit{traces}(\mathcal{M}_i)$, the subset of traces permitted by $\mathcal{M}_i$.

B. Directly Extractable Models

Models defined solely in terms of release, execution, and suspension times are easy to infer since all relevant parameters are explicitly part of the intermediate representation (*i.e.*, conceptually the stream of jobs) consumed by the model extractors.

For example, to infer the sporadic task model's job separation parameter T_i, it suffices to keep a running minimum of the distance between any two consecutive job releases for each task. Similarly, the dynamic self-suspension model parameter S_i is trivially determined by a running maximum, and inference of segmented self-suspension models is also straightforward.

The inference of arrival and execution-time curves is algorithmically more nuanced, but has long been well understood. For example, Stark provides a detailed discussion [51, pp. 99–105] of the extraction algorithms also used in LIME.

C. Periodic Model Inference

This leaves the periodic model, which is by far the most challenging to infer. The principal source of difficulty is that a job's arrival time $a_{i,j}$ cannot be directly observed for most job separators. We therefore focus in the following on extracting the *arrival tuple* (Φ_i, T_i, J_i) for separators where $a_{i,j}$ cannot be directly observed. To reduce clutter, we focus on an individual thread in the remainder of this section and use job indices only (*e.g.*, we write r_j instead of $r_{i,j}$ in the following).

Inference goal. Given a sequence $\mathbf{R} \triangleq \langle r_1, r_2, \ldots, r_z \rangle$ of z jitter-affected releases observed by LIME, the objective is to infer a *conservative* arrival tuple (Φ, T, J) satisfying Eq. (1). For brevity, let $A_{\Phi,T}(j) = \Phi + (j-1) \cdot T$ be the arrival sequence implied by (Φ, T, J).

Consider Fig. 2a, which illustrates a running example $\mathbf{R} = \langle 100, 115, 120, 135 \rangle$. Three arrival sequences are shown:

- ▲ $A_{100,15}(j)$: which is *unsound*, as the projected arrivals for $j = 3$ and $j = 4$ occur after the observed releases.
- ■ $A_{100,5}(j)$: which is *sound*, but has an overly pessimistic maximum jitter $r_4 - A_{100,5}(4) = 135 - 115 = 20$.
- ● $A_{100,10}(j)$: which is *sound* with a maximum jitter of 5 time units, the least obtainable by any sound sequence.

This example shows that multiple arrival sequences can fit an observed release sequence. However, a single model—the one *minimizing maximum jitter*—provides the most accurate representation. Thus, our approach aims to find the arrival tuple (Φ, T, J) with the minimal maximum jitter bound J.

To this end, we leverage two key properties $(\mathbf{I})$ and $(\mathbf{II})$:

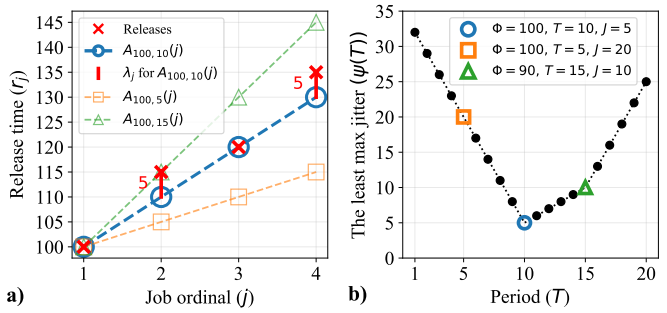


Fig. 2: **a)** Arrival sequences and observed releases. **b)**: The least sound max. jitter bounds obtainable with the given periods.

(**I**) For each period T, there exists a unique offset $\phi(T)$ that minimizes the maximum jitter for $\mathbf{R}$.

The offset $\phi(T)$ is the largest offset for which the arrival sequence $A_{\Phi,T}(j)$ is sound (w.r.t. $\mathbf{R}$). It can be computed iteratively as follows: for $j = 1$, $\Phi(1) = r_1$, and for $j > 1$, $\Phi(j) = \Phi(j-1)$ if $r_j - A_{\Phi(j-1),T}(j) \geq 0$, and $\Phi(j) = \Phi(j-1) - (A_{\Phi(j-1),T}(j) - r_j)$ otherwise . Finally, $\phi(T) \triangleq \Phi(z)$, and $\psi(T) \triangleq \max_{1 \leq j \leq z} \left(r_j - A_{\phi(T),T}(j)\right)$ is the least maximum jitter for any sound arrival sequence with period T.

However, property (**I**) alone is insufficient, as a brute-force approach to minimizing $\psi(T)$ would require an exhaustive search over a broad range of periods. Conveniently, as illustrated in Fig. 2b, $\psi(T)$ first decreases with increasing period until reaching a minimum, after which it increases again. Intuitively, the reason is that "ill-fitting" periods that are either too short or too long result in "more" maximum jitter to compensate for the mismatch. The best fit is hence achieved with a period that results in minimal maximum jitter, which gives us the second important property:

(**II**) A period $T_{\min}$ that minimizes $\psi(T)$ can be found via *ternary search*, which is computationally efficient.

Ultimately, the arrival tuple $(\phi(T_{\min}),\ T_{\min},\ \psi(T_{\min}))$ minimizes the maximum jitter for $\mathbf{R}$.

Online inference. So far, we have outlined the approach under the implicit assumption that the entire sequence $\mathbf{R}$ is known (an offline setting). To keep LIME's memory footprint and runtime low in its online mode, at most b recent releases are retained, which are analyzed with Algorithm 1 in sequential batches $\mathbf{B}_1, \mathbf{B}_2, \ldots$ of length b each. Each batch (except $\mathbf{B}_1$) includes the final release of the previous one to maintain continuity.

(1) For a model derived from an arbitrary batch $\mathbf{B}_k$, we must account for two sources of imprecision. First, to deal with potentially noisy observations, which could skew the estimate if left uncorrected, LIME first identifies outliers among the inter-release gaps within $\mathbf{B}_k$ using the well-known lightweight *Hampel identifier* [*e.g.*, 45, Ch. 3.2.2]. Then, it identifies the first release r' not followed by an outlier gap and the last release r'' not preceded by one. The optimal period $T_{\min}$ is computed based on property (**II**) from the release times in $[r', r'']$.

To anticipate that slightly different models may better fit *future releases*, the initial batch $\mathbf{B}_1$ is processed differently (lines 1–13) from subsequent ones by computing $T_{\min}$ (line 4) and generating two sets of *candidate periods* (lines 5 and 6):

Algorithm 1 Periodic Model Inference

1: **procedure** INIT($\mathbf{B}_1$) ▷ $\mathbf{B}_1 = \langle r_1, \ldots, r_z \rangle \wedge z \geq 2$
2: $\mathbf{G}$, $\mathbf{H} \leftarrow [\,]$ ▷ *Init collections of candidates*
3: $\mathbf{R}' \leftarrow \langle r \in \mathbf{B}_1 \mid r' \leq r \leq r'' \rangle$ **as per** (1) ▷ *Outlier removal*
4: $T_{\min} \leftarrow \text{ternary}\,(\mathbf{R}', \psi(\cdot))$ ▷ *Minimize $\psi(T)$ w.r.t. $r \in \mathbf{R}'$*
5: $\mathbb{G} \leftarrow$ Generate a set of fine-grained periods following (2)
6: $\mathbb{H} \leftarrow$ Generate a set of rounded periods following (3)
7: **for** $T \in \mathbb{G} \cup \{T_{\min}\}$ **do** ▷ *Fine-grained model candidates*
8: **append** $\delta(\mathbf{B}_1, r_1, T, 0, 0)$ **to** $\mathbf{G}$
9: **for** $T \in \mathbb{H}$ **do** ▷ *Rounded model candidates*
10: **append** $\delta(\mathbf{B}_1, r_1, T, 0, 0)$ **to** $\mathbf{H}$
11: $T_{\mathbb{E}} \leftarrow T_{\min}$ ▷ *Init mean period*
12: $\ell \leftarrow z$ ▷ *Init ordinal of the last observed release*

13: **procedure** UPDATE($\mathbf{B}_k$) ▷ $\mathbf{B}_k = \langle r_1, \ldots, r_z \rangle \wedge k > 1 \wedge z \geq 2$
14: $\mathbf{R}' \leftarrow \langle r \in \mathbf{B}_k \mid r' \leq r \leq r'' \rangle$ **as per** (1) ▷ *Outlier removal*
15: $T_{\min} \leftarrow \text{ternary}\,(\mathbf{R}', \psi(\cdot))$ ▷ *Minimize $\psi(T)$ w.r.t. $r \in \mathbf{R}'$*
16: $T_{\mathbb{E}} \leftarrow T_{\mathbb{E}} + (T_{\min} - T_{\mathbb{E}})/(k-1)$ ▷ *Welford's alg. for mean*
17: **for** $T \in \{T_{\min}, T_{\mathbb{E}}\}$ **do** ▷ *Derive sound offset and jitter*
18: $(\Phi', T', J') \leftarrow \text{argmin}_{(\Phi'', T'', J'') \in \mathbf{G} \cup \mathbf{H}} (|T'' - T|)$
19: **if** $T > T'$ **then**
20: $\Phi \leftarrow A_{\Phi',T'}(\ell) - \ell \cdot T$
21: $J \leftarrow \Phi' + J' - \Phi$
22: **if** $T < T'$ **then**
23: $\Phi \leftarrow \Phi'$
24: $J \leftarrow A_{\Phi',T'}(\ell) + J' - A_{\Phi,T}(\ell)$
25: **append** (Φ, T, J) **to** $\mathbf{G}$
26: **for** $(\Phi, T, J) \in \mathbf{G} \cup \mathbf{H}$ **do** ▷ *Sound update of Φ and J*
27: **update** (Φ, T, J) **with** $\delta(\mathbf{B}_k, \Phi, T, J, \ell)$
28: **prune** $\mathbf{G}$ **and** $\mathbf{H}$ **as per** (4)
29: $\ell \leftarrow \ell + z - 1$ ▷ $(\ldots - 1)$ because $r_1 \in \mathbf{B}_k = r_z \in \mathbf{B}_{k-1}$

30: **procedure** $\delta(\mathbf{B}, \Phi, T, J, x)$ ▷ $\mathbf{B} = \langle r_1, \ldots, r_z \rangle \wedge z \geq 2$
31: **for** $j = 2$ to z **do**
32: **if** $r_j - A_{\Phi,T}(x + j) < 0$ **then** ▷ *If implausible arrival*
33: $\Phi \leftarrow \Phi - (A_{\Phi,T}(x + j) - r_j)$ ▷ *update offset*
34: $J \leftarrow J + (A_{\Phi,T}(x + j) - r_j)$ ▷ *update max. jitter*
35: **return** (Φ, T, J)

$\mathbb{G}$ *fine-grained* periods close to $T_{\min}$, intended to account for minor estimation inaccuracies; and

$\mathbb{H}$ *human-designed* (rounded) periods (*e.g.*, favouring "50 ms" rather than "52.001 ms" if supported by the data).

(2) For set $\mathbb{G}$ (line 5), LIME generates up to 50 evenly spaced candidate periods around $T_{\min}$, spanning the range $[T_{\min} - \sigma \cdot \psi(T_{\min}) - 10,\ T_{\min} + \sigma \cdot \psi(T_{\min}) + 10]$, where σ is an arbitrary granularity parameter (by default, $\sigma = 3$).

(3) Set $\mathbb{H}$ (line 6) is obtained by rounding $T_{\min}$ at each significant digit and including candidates with adjustments of ± 1 and ± 2 (*e.g.*, a $T_{\min} = 5.01$ ms results in $3, 4, 5, 6, 7$ at millisecond granularity, and similarly at smaller granularities).

Model updates. For each generated period $T \in \mathbb{G} \cup \mathbb{H}$, a candidate model $(\Phi, T, J) \in \mathbf{G} \cup \mathbf{H}$ is constructed according to property (**I**) (lines 7–10). As new batches arrive, the models are iteratively updated (lines 13–29) by recomputing Φ and J for newly observed releases (lines 27 and 30–35).

For each batch $\mathbf{B}_k$ beyond $\mathbf{B}_1$ (*i.e.*, $k > 1$), before updating all candidates as explained above, we introduce new candidates to $\mathbf{G}$ using the optimal period for $\mathbf{B}_k$ and the mean of all optimal periods up to $\mathbf{B}_k$ (inclusive). To preserve soundness,

any newly introduced model with T must remain valid for previously processed (but no longer available) releases by computing its Φ and J from sound models in $\mathbf{G}$ and $\mathbf{H}$. For a newly estimated period T, LIME selects the sound candidate (Φ', T', J') with the closest period T' (lines 17–25). Then, given the ordinal ℓ of the last release from $\mathbf{B}_{k-1}$, LIME first selects the largest offset Φ such that $A_{\Phi,T}(j) \leq A_{\Phi',T'}(j)$ for $1 \leq j \leq \ell$; by transitivity, this ensures it is also less than or equal to previously observed releases. To determine the jitter bound, LIME identifies the j where the two sequences deviate most, implicitly assuming that the sound model always experiences its maximum jitter J'.

If $T > T'$, a sound offset is given by $\Phi = A_{\Phi',T'}(\ell) - \ell \cdot T$ and a sound jitter is given by $J = \Phi' + J' - \Phi$. Offset Φ is chosen to align the sequences at the last release ℓ.

If $T < T'$, a sound offset is $\Phi = \Phi'$, and a sound jitter bound is $J = A_{\Phi',T'}(\ell) + J' - A_{\Phi,T}(\ell)$. The reasoning here is reversed: for $A_{\Phi,T}(j)$ to remain below $A_{\Phi',T'}(j)$ for any $j \leq \ell$, it suffices to start from the same offset.

④ To reduce runtime overhead, the candidate sets are pruned after each batch to remove excessively pessimistic estimates (*e.g.*, $T = 15$ in Fig. 2a). To this end, the candidate with the least jitter bound J_m in $\mathbf{G} \cup \mathbf{H}$ is identified and any candidate with a jitter bound exceeding $5 \times J_m$ is discarded.

Output. Finally, LIME derives the candidate (Φ_m, T_m, J_m) with the lowest maximum jitter J_m from $\mathbf{G} \cup \mathbf{H}$. To favor periods likely preferred by system designers, it rounds T_m at each significant digit in decreasing order, selecting the first matching period candidate from $\mathbf{H}$ with a jitter bound within 25% of J_m. If no such candidate exists, LIME returns (Φ_m, T_m, J_m) to balance jitter minimization and human biases.

V. Evaluation

We evaluated LIME using synthetic benchmark programs to explore its model inference accuracy and runtime overhead. Additionally, we report on experiences with programs not written for benchmarking purposes in Sec. VI.

Platform. All experiments and case studies were conducted on a *Raspberry Pi 4 Model B* (or *RPi* for short), featuring a quad-core ARM Cortex A72 processor and 8 GiB RAM. The RPi ran Ubuntu Server 22.04 LTS, which is based on Linux 5.15.

A. Model Inference Accuracy

To assess LIME's model extractor across a wide range of workloads, we let LIME extract all types of models introduced in Sec. II for workloads comprised of synthetic benchmark programs that implemented randomly chosen periods.

For variety, we used four different period generators:

- *automotive periods* were drawn randomly from the set of periods $\{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$ ms commonly encountered in the automative industry, using the distribution of periods reported by Kramer et al. [33];
- *LogU-ms periods* were chosen log-uniformly at random from the range $\{1, 2, \ldots 1000\}$ ms;
- *LogU-μs periods* were chosen log-uniformly at random from the range $\{1000, 1100, 12000, \ldots, 1000000\}$ μs; and
- *LogU-ns periods* were chosen log-uniformly at random from the range $\{10^6, 10^6 + 1, 10^6 + 2, \ldots, 10^9\}$ ns.

For each period distribution, we generated 100 workloads comprised of $n = 20$ periodic tasks each. For each task, we randomly selected a *driver program* to implement it. We implemented driver programs to exercise the following system calls, which can all be used to effect periodic thread activations: `clock_nanosleep`, `sig_timed_wait`, `poll`, `read`, `recvfrom`, `mq_timed_receive`, `futex_wait_bit_set`, `msgrcv`, and `semop`. Additionally, we used Linux's standard latency-testing benchmark `cyclictest` as a driver, too.

For each workload, we generated a shell script to launch the driver programs realizing the periodic tasks on the target platform. Many of the exercised system calls require input (*e.g.*, for a thread to be periodically activated via UDP packets, some other thread must send those packets). In those cases, we launched *two* processes: a consumer process that we evaluate, and another producer process to periodically generate inputs. The number of real-time *processes* launched in the system can thus exceed the number of evaluated *tasks* by a factor of two.

Each of the 400 workloads was executed with rate-monotonic priorities under SCHED_FIFO and traced for 10 minutes, for a total runtime of over 66 hours of traced real-time execution. To enable independent validation of the extracted models, we used LIME's offline mode to record the full event trace. In total, we collected more than 330 GiB of event traces. The volume of the collected traces highlights a key advantage of online inference, especially on embedded platforms with limited storage.

Offline validation. Recall that LIME emits instances of all models reviewed in Sec. II-B for each monitored thread. To assess LIME's inference accuracy for the sporadic model (T_i), upper and lower arrival curves (α_i^+ and α_i^-), the dynamic and segmented self-suspension models (S_i and V_i), and cumulative execution-time curves (WCET_i), we compared the models extracted by LIME's *online* model-inference algorithms against an *offline* baseline that extracts models from the recorded event traces. Importantly, the offline model extractor is a completely independent baseline that shares no code with LIME.

Fig. 3 summarizes the results: For the just-mentioned types of models (*i.e.*, T_i, α_i^+, α_i^-, S_i, V_i, and WCET_i), LIME achieves 100% accuracy relative to the offline baseline for each of the considered period distributions. That is, the models produced by LIME's online inference algorithms are *identical* to those produced by the independent offline model extractor, which confirms that for these models online inference does *not* imply a tradeoff in precision. Importantly, this also confirms that LIME satisfies the *conservative extraction* criterion specified in Sec. IV-A: the models inferred online by LIME are not "lossy" and correctly reflect *all* observations made at runtime.

Ground-truth periods. We further confirmed that *all* instances of the periodic model with jitter (*i.e.*, Φ_i, T_i, and J_i) inferred online by LIME satisfy the conservative extraction criterion, too. Additionally, for the periodic model, we compared against an even more demanding baseline, namely the *ground-truth period* that each traced driver program was configured to realize.

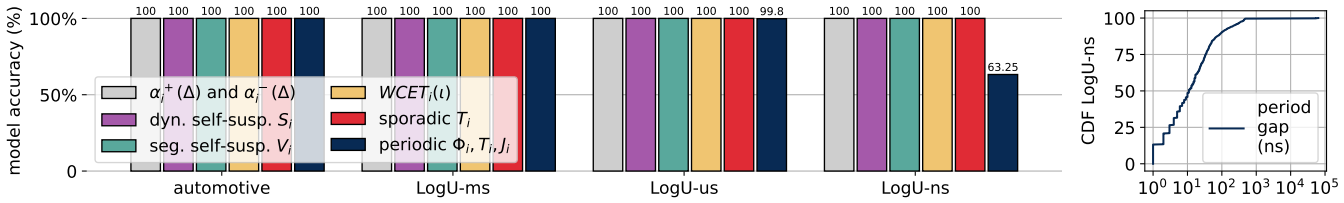


Fig. 3: Online model-inference accuracy. Right-hand side: Distribution of the gap between inferred and ground-truth periods for the 36.75% of the extracted periodic models *not* deemed accurate under the LogU-ns distribution (99^{th} percentile $< 500\,\text{ns}$).

A practical complication is that system overheads can prevent driver programs from enacting the intended period perfectly down to the exact nanosecond. We thus counted a periodic model inferred by LIME's online model extractor as *accurate* in two separate cases: **(i)** when LIME identifies *exactly* the ground-truth period given as input to the driver program, and **(ii)** when LIME identifies a period diverging from the intended ground-truth period and the independent offline extractor *confirms* that the driver program indeed failed to precisely enact the specified period (*i.e.*, the offline extractor confirms that LIME correctly identified system limitations).

Overall, as Fig. 3 shows, LIME achieves near-perfect ground-truth accuracy for three of the four period distributions. In particular, for automotive and LogU-ms periods, LIME achieves 100% ground-truth accuracy. The reason is that periods rounded to millisecond granularity leave a clear signal in the trace that LIME is able to quickly detect thanks to its rounding heuristic. LIME also succeeds in inferring almost all of the LogU-μs periods accurately, as this period distribution's signal emerges through the noise relatively quickly, too.

The most difficult period distribution is LogU-ns, as there is no structure for LIME to uncover (*i.e.*, the rounding heuristic fails). Nonetheless, LIME is still accurate in over 60% of the cases. Furthermore, even when LIME is not perfectly accurate, it is usually very close. This can be seen on the right-hand side of Fig. 3, which shows the distribution of the gap between LIME's model period and the ground-truth period in those cases in which LIME's model is counted as inaccurate: in roughly 50% of the inaccurate models, the gap is only $10\,\text{ns}$ or less, it is less than $100\,\text{ns}$ in roughly 90% of the inaccurate models, and even the 99^{th} percentile is less than $500\,\text{ns}$, with a maximum observed gap of roughly $64\,\mu\text{s}$. Longer observations could likely further improve the accuracy for these workloads, as additional samples help LIME to narrow in on the ground-truth period that minimizes the maximum jitter.

The data shown here was obtained with LIME's default batch size $b = 200$. We repeated the experiment while varying the batch size from $b = 10$ to $b = 250$ and found that LIME's online period extractor is relatively insensitive to the batch size parameter. Only in case of the LogU-ns periods does a larger batch size show a clearly positive effect, but even then a point of diminishing returns is reached already around $b = 100$.

In summary, LIME is highly accurate. It is thus well suited for *behavior validation*: LIME can reliably confirm that a thread realizes its timing specification correctly, and conversely, if a model reported by LIME does not match expectations, it is likely indicative of actual thread behavior diverging from the intended specification. We revisit this point in Sec. VI-B.

B. Runtime Overhead

To assess LIME's impact on processor utilization, we generated 5 workloads for each $n \in \{1, 2, \ldots, 20\}$ using the LogU-ms period distribution, for a total of 100 workloads. We configured LIME to run in its default online mode, in which it extracts models directly without storing traces for offline analysis. Each workload was observed for 5 minutes, for a total of over eight hours of real-time execution.

There are two sources of runtime overhead. Most critically, the eBPF probes installed to observe thread behavior cause some overhead on critical paths in the kernel and thus affect all real-time tasks. These overheads can be assessed with the kernel's built-in eBPF introspection facilities that provide access to per-probe runtime statistics (via the `/proc` file system).

Additionally, in user space, LIME processes the stream of thread events with the model-extraction algorithms discussed in Sec. IV. We measured LIME's user-space runtime via Linux's `schedstat` interface (also via `/proc`). However, the LIME process does *not* run with real-time priority, and hence its runtime has no immediate effect on the real-time workload.

Workloads impose different levels of stress on the kernel and LIME, depending on the rate at which threads execute and how many system calls they make on average per thread activation. As a proxy for a workload's intensity, we computed for each workload the total rate of thread wake-ups per second (across all monitored threads). Fig. 4 shows the observed average utilization (normalized to the RPi's four cores) of both all eBPF probes and the LIME user-space process for each workload as a scatter plot in relation to the workload's total wake-up rate.

There are several key takeaways. First, as expected, the cumulative overhead of all eBPF kernel probes exhibits a clear linear dependency on the traced workload's rate of thread wake-ups. More intense workloads trigger eBPF probes more often and require more events to be relayed to user space, with obvious effects. The average utilization of the LIME process also follows a clear linear trend with a similar slope.

Second, eBPF overheads, which directly impact the real-time workload, are reassuringly low. Even for the most demanding workloads we observed, LIME's eBPF probes consume less than 2.5% of the system's total processing capacity. This confirms, together with experiments reported on in Secs. V-C and V-D below, that LIME has little impact on the traced workload.

Third, LIME's user-space utilization is also low, topping out at less than 2% average utilization (w.r.t. four cores) on our RPi

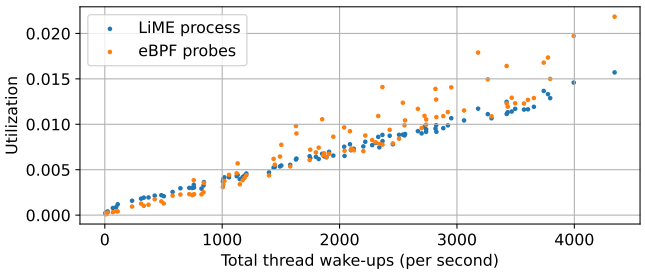


Fig. 4: Runtime overhead on the RPi vs. thread wake-ups

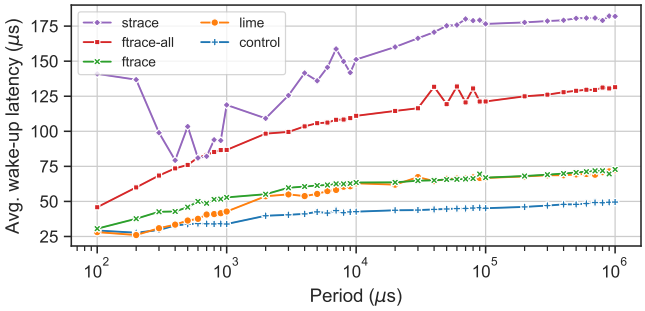


Fig. 5: Impact on average thread-activation latency

platform. Additionally, since the runtime of the LiME process does not immediately affect real-time threads (because it runs as a regular SCHED_OTHER process), there is considerable headroom to accommodate even more intense workloads.

In summary, our evaluation shows that LiME's runtime overhead is low on our relatively old RPi—and for low-intensity workloads even negligible—which suggests that LiME is more than fast enough for most modern embedded Linux platforms.

C. Latency Impact

Since eBPF probes execute on critical kernel paths, it is important to understand their effect on thread-activation latency (*i.e.*, how quickly a thread executes in response to an interrupt). Linux's standard latency benchmark is `cyclictest`, which periodically measures the difference between when an activation should ideally take place and when it is actually recorded in user space, with a configurable period.

We repeatedly ran `cyclictest` with varying periods from 100 μs to 1000 ms for 15 minutes for each period choice. For comparison purposes, we repeated the experiment without instrumentation ("control") and with three other tracing approaches (`strace`, `ftrace`, and `ftrace-all`). Fig. 5 shows the average thread activation latency reported by `cyclictest`.

LiME is at the lower end of the spectrum, causing a latency impact of only around 20 μs on our platform. This is in large parts due to the efficient eBPF infrastructure upon which LiME builds. Its overheads are in line with the similarly eBPF-based `ftrace`. The results also indicate that intercepting and inspecting all system calls in user space (as done by `strace`) would not be a viable alternative design for LiME.

D. Throughput Impact

Finally, we explored LiME's impact on overall system performance with a standard *Redis* benchmark. Redis, a key-value server widely used as an in-memory database and cache, is representative of a class of latency-sensitive, throughput-oriented workloads commonly encountered on Linux. We deployed Redis on our target platform and traced the server process with LiME while it served requests generated by `memtier_benchmark` (a standard Redis benchmark [1]) running on a server-class machine directly connected to the target RPi. The load was generated by 16 threads sending 1 KiB requests without pipelining. We repeated the experiment for all baseline tracers as in Sec. V-C and without instrumentation ("control").

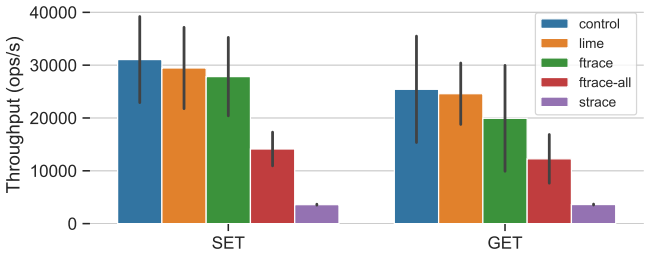


Fig. 6: Redis throughput (bars indicate one standard deviation)

Fig. 6 reports the average throughput measured over 10 runs in which each thread sent 500,000 requests to the server. The throughput reduction caused by LiME is modest and compares favorably to the heavier tracing approaches (*i.e.*, `ftrace-all` and `strace`). This confirms the trends from Secs. V-B and V-C: LiME's runtime and latency overheads do not place an undue burden on the workload under observation.

VI. Case Studies

In addition to the benchmarks discussed in Sec. V, we also explored LiME's utility with two case studies involving workloads not specifically designed for evaluation purposes.

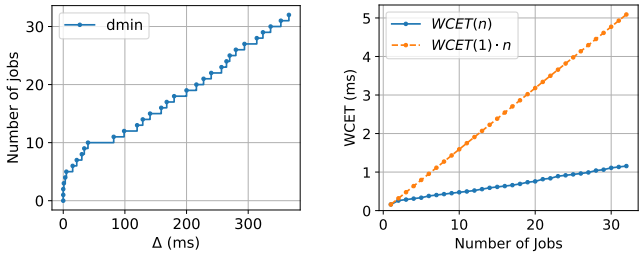
A. Identifying and Characterizing System Threads

One of the roles in which LiME is useful, as highlighted in Sec. I, is to aid with *system exploration*. In particular, if LiME is configured to trace *all* real-time threads in the system, it can reveal threads executing with real-time priorities that a system designer might not have anticipated or overlooked otherwise.

Case in point, all results collected for the accuracy and overhead evaluations (Secs. V-A and V-B) reveal the presence of *system threads* executing at maximum real-time priority. As these threads are inherently part of the system and consume processor capacity at high priority, a schedulability analysis must take them into account, even if they are not part of the system's *payload* (*i.e.*, *intended* workload).

With LiME, their presence and impact becomes obvious. In total, we observed four kernel `migration` threads (one per core), which assist the Linux scheduler in certain cases with moving threads from core to core, and three `multipathd` threads, which assist the kernel with peripheral device management.

As a representative example, Fig. 7a shows the arrival curve $\alpha^+(\Delta)$ of the kernel `migration` threads; Fig. 7b shows the corresponding cumulative execution-time curve $\mathrm{WCET}(\iota)$. As kernel threads do not issue system calls, the only viable job separator for these threads is the suspension separator. These



(a) Arrival curve (b) WCET

Fig. 7: Arrival curve and WCET of kernel `migration` threads.

observations reflect about 75 hours of *in situ* observation (all traces discussed in Secs. V-A and V-B) and thus represent a considerable empirical basis for parameter estimation.

The arrival curve in Fig. 7a shows that `migration` threads are activated relatively regularly, with a longterm rate of roughly $\frac{\alpha^+(360\,\text{ms})-\alpha^+(75\,\text{ms})}{355\,\text{ms}-75\,\text{ms}} \approx \frac{1\,\text{job}}{14\,\text{ms}}$, but with significantly bursty arrivals for $\Delta \leq 50\,\text{ms}$. The $\text{WCET}(\iota)$ curve in Fig. 7b reveals a mostly linear trend, but starting only at $\iota \geq 2$. For comparison purposes, Fig. 7b also shows the extrapolated scalar WCET bound $C_i \cdot \iota$, which reveals the difference in slope.

Interestingly, the observed system threads clearly demonstrate the need for non-scalar task models to accurately characterize real-world system behavior. For example, when performing a response-time analysis [12, 36], a central concept is the *request-bound function* (RBF), which bounds the total execution time of all jobs released by a given task τ_i in any interval of length Δ. For the scalar sporadic task model, it is defined as $RBF_i^{sp}(\Delta) = \lceil \Delta/T_i \rceil \cdot C_i$. Given the curves $\alpha_i^+(\Delta)$ and $\text{WCET}_i(\iota)$, we can define the RBF also more accurately as $RBF_i^{wc}(\Delta) = \text{WCET}_i(\lceil \Delta/T_i \rceil)$, $RBF_i^{ac}(\Delta) = \alpha_i^+(\Delta) \cdot C_i$, or simply $RBF_i(\Delta) = \text{WCET}_i(\alpha^+(\Delta))$.

For the kernel `migration` threads, the degree of overestimation by (partially) scalar models is significant: for $\Delta = 300\,\text{ms}$, we obtain a grossly pessimistic bound $RBF_i^{sp}(300\,\text{ms}) \approx 1515.8\,\text{ms}$ for the sporadic model, still pessimistic bounds $RBF_i^{wc}(300\,\text{ms}) \approx 344.3\,\text{ms}$ and $RBF_i^{ac}(300\,\text{ms}) \approx 4.3\,\text{ms}$ for the hybrid models, and a more reasonable fully non-scalar bound $RBF_i(300\,\text{ms}) \approx 0.99\,\text{ms}$. The `multipathd` threads inherently require non-scalar models, too.

B. Detecting and Debugging Unintended Timing Behavior

To explore LiME's effectiveness for observing real-world runtime behavior, we experimented with ROSACE [44], a representative aircraft control system case study. In contrast to Sec. V, which involved synthetic benchmarks, this case study showcases LiME's practical utility in identifying and diagnosing runtime issues in a real-world workload not prepared by us.

ROSACE includes two distinct versions: one programmed in the C programming language directly targeting POSIX and one realized with Prelude [25, 43], which uses the SchedMCore [20] user-space scheduling framework. Prelude's use of SchedMCore—which dynamically adjusts the processor affinities and priorities of the threads managed by it to enact custom scheduling policies—provided us with an opportunity to evaluate LiME's compatibility with a non-standard scheduling approach. The fact that LiME successfully identified suitable task models regardless emphasizes its versatility in analyzing workloads using either kernel- or user-space scheduling.

When running the POSIX version on our RPi platform, LiME identified 5 active real-time threads, while in the Prelude implementation it detected 17 active threads—a result of SchedMCore's custom user-space scheduling, which manages job execution differently from traditional POSIX scheduling.

Given that ROSACE implements an avionics control system [44], periodic execution can be expected. However, for the POSIX implementation, LiME extracted a period of $T = 5.339\,\text{ms}$ and observed an unreasonably large maximum jitter of $J = 487.12\,\text{ms}$ for ROSACE's main worker thread. While the period may not seem too unusual on its own, such a massive jitter bound is a strong indication that something is amiss. In contrast, the Prelude implementation's main thread exhibited a period of $T = 2\,\text{ms}$ with a reasonable maximum jitter of $J = 0.205\,\text{ms}$, showing stable timing behavior.

A difference in period is not surprising because Prelude is a synchronous language that compiles periodic models differently before handing them to SchedMCore for execution. A *four-orders-of-magnitude* difference in maximum jitter, however, defies plausible explanation. We thus manually examined the code and found that ROSACE's POSIX version uses `usleep()` in an attempt to enact periodic behavior in the main thread, and `pthread_barrier_wait()` to propagate activations to worker threads. The use of `usleep()` caused delays due to its improper use. Specifically, it was called to unconditionally suspend the thread for 5 ms without accounting for the thread's own execution time, leading to unintended *timer drift*—which LiME correctly detected and accounted for in its reported model.

We modified ROSACE's POSIX version to use absolutely timed sleeps, which ensures precise, drift-free timing [13]. We then used LiME to validate the patch: it confirmed that the patched version achieves the desired period of $T = 5\,\text{ms}$ with less than $J = 0.1\,\text{ms}$ of jitter. We also validated the other worker threads relying on `pthread_barrier_wait()`, which LiME confirmed to exhibit the correct period of $T = 5\,\text{ms}$.

Notably, to be sure of the absence of timer drift, we let LiME observe ROSACE for a full hour. Such prolonged use of LiME in a continuous monitoring role is made possible only by its online mode, as recording traces spanning an hour for offline analysis would have quickly exhausted the RPi's storage.

Without LiME, it would have been much more difficult to observe and detect such a minuscule difference between actual and intended period (5.338 ms vs. 5 ms), as such a small, but creeping deviation can easily escape notice during testing. By accurately extracting models across different scheduling mechanisms, LiME provides essential insights for developers that can help to pinpoint subtle timing issues.

VII. Limitations and Extension Opportunities

LiME provides a unique set of capabilities—online model inference for black-box Linux threads—not found in any prior tool. Nonetheless, in its initial version, it is subject to certain limitations that will be interesting to investigate in future work.

Foremost, LiME currently focuses on classic models of independent, sequential real-time tasks. For example, while LiME's models already reflect all observed synchronization delays (*e.g.*, threads busy-waiting on a spin lock or blocking on a mutex), the models that LiME emits do not include explicit information on *shared resources* (such as critical-section lengths or the set of tasks sharing a particular resource). Similarly, LiME's models do not yet explicitly express *precedence constraints*, although they already fully capture any release jitter resulting from such constraints. Furthermore, LiME currently cannot identify *parallel task models* such as fork-join or general DAG tasks.

The common reason is that LiME's eBPF probes are limited to kernel-observable events, whereas *user-space instrumentation* would be required to infer richer task models (*e.g.*, spin locks and futex fast-paths are not kernel-visible, middlewares such as OpenMP or ROS are largely opaque to the kernel). Incorporating Linux's *uprobes* could help provide the necessary user-space introspection capabilities, but doing so introduces significant challenges beyond the scope of this paper.

Another limitation is that LiME's execution-time model (WCET_i) currently reflects the time spent executing on any processor, irrespective of a processor's type or its current speed. In heterogenous systems (*e.g.*, with separate "efficiency cores" and "performance cores") or when dynamically adjusting processor speeds, it may be helpful to augment LiME's output with WCET_i models for specific core types and speed settings.

Lastly, like any measurement-based approach, LiME is limited to modeling observed behavior and cannot provide formal guarantees covering unobserved corner cases; stronger guarantees would require the integration of static analysis approaches (to the extent possible on typical Linux platforms, if at all). Nonetheless, LiME's *conservative extraction* criterion ensures that LiME never under-approximates observed behavior. In particular, LiME does *not* emit models based on incomplete observations. In the limit, if the system becomes overloaded to a degree such that LiME cannot keep up with the generated events (*e.g.*, if the LiME process is starved), it will detect that there are gaps in the event stream and simply not emit any models.

VIII. Related Work

A wide range of research has focused on analyzing system behavior through tracing and modeling. Over the years, much emphasis has been placed on low-overhead, minimally intrusive tracing mechanisms and toolkits suitable for embedded real-time systems, including for Linux [23], Linux-derived systems [14], microkernel OSs [32, 48], and popular middlewares such as ROS [2, 6, 38]. These differ from LiME in significant ways: in contrast to LiME, most tracing toolkits are typically deployed to collect traces for later offline analysis. Additionally, none of these tracing approaches is specifically targeting real-time task models and hence none include comparable model-extraction capabilities. LiME benefits from the ready availability and maturity of Linux's eBPF support, so that we did not have to develop low-level tracing capabilities ourselves.

As LiME targets classic real-time task models conceived for schedulability analysis, which adopt a high-level, abstract view of a real-time system, it does not track thread internals in much detail (*i.e.*, only system calls). In contrast, at the other end of the spectrum, tools like *MadT* [16] provide detailed insights into memory access patterns in multi-core real-time systems. Earlier, Bastoni et al. [4] presented an empirical method to approximate cache-related preemption delays as an input to an overhead-aware schedulability analysis, but using synthetic benchmark tasks rather than *in situ* observation.

Vădineanu and Nasri [55] previously studied the period inference problem in a different context, namely in an offline setting with access to complete traces but without access to job release times. In contrast, our online solution (Sec. IV) works directly on the release times observed via eBPF.

Much more closely related in spirit, though not in the choice of modeling abstraction, is work by De Oliveira et al. [21] who extract automata-based models from the Linux kernel, allowing for a detailed exploration of synchronization behaviors. Similarly, work by De Oliveira and De Oliveira [22] exploring the Linux kernel through a custom tracing tool to model timing delays, mapping Linux's low-level timing behavior to real-time scheduling abstractions, shares much of our motivation. Another offline approach is by Beamonte et al. [5], who use execution traces to generate behavioral models of real-time applications. Maggio et al. [40] proposed *rt-muse*, which combines tracing and modeling in the context of a controlled experiment with synthetic tasks on a Linux system to derive supply-bound functions as well as bounds on migration events and response times. LiME differs from these efforts by its focus on *in situ* online model extraction, the identification of a large number of job separators, and the range of supported classic task models.

IX. Conclusion

We are hopeful that LiME will prove useful in various contexts. As a research tool, it bridges theory and practice, laying a foundation for future work that connects both realms. For researchers focused on analysis, LiME provides a means to obtain models from actual workloads with relatively little effort, thereby increasing the validity of synthetic evaluations based on metrics like schedulability ratios. Notably, LiME's automated nature means it can be used without in-depth knowledge of Linux's low-level tracing facilities or system call semantics.

For systems-oriented researchers experimenting with practical workloads on Linux, LiME opens the door to schedulability analysis, also in a largely automated way, without a need for extensive prior study of the relevant real-time literature.

For practitioners, LiME can provide insights into the difficult-to-observe temporal behavior of the system in a number of use cases, including the validation of intended timing behavior and integration testing (Secs. V-A and VI-B), system exploration (Sec. VI-A), debugging of abnormal behavior (Sec. VI-B), and low-overhead continuous monitoring (Secs. V-B to V-D).

Many opportunities for future extensions of LiME remain, including support for parallel task models, synchronization APIs, and user-provided specification hints and annotations.

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

https://lime.mpi-sws.org

ACKNOWLEDGEMENTS

We thank Marco Perronet and Marco Maida for their contributions to an earlier exploration of the problem space [47].

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 803111).

REFERENCES

[1] "memtier_benchmark: A high-throughput benchmarking tool for Redis & Memcached," https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/, 2013.

[2] H. Abaza, D. Roy, S. Fan, S. Saidi, and A. Motakis, "Trace-enabled timing model synthesis for ROS2-based autonomous applications," in *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.

[3] N. 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] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2010, pp. 33–44.

[5] R. Beamonte, N. Ezzati-Jivan, and M. R. Dagenais, "Automated generation of model-based constraints for common multi-core and real-time applications using execution tracing," *International Journal of Parallel Programming*, vol. 49, no. 1, pp. 104–134, 2021.

[6] 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.

[7] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, 1991.

[8] 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.

[9] H. Bourbouh, P.-L. Garoche, T. Loquen, É. Noulard, and C. Pagetti, "CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models," in *Proceedings of the 10th European Congress on Embedded Real Time Systems (ERTS)*, 2020.

[10] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 586–601.

[11] F. Boussinot and R. De Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.

[12] S. Bozhko and B. B. Brandenburg, "Abstract response-time analysis: A formal foundation for the busy-window principle," in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020, pp. 22:1–22:24.

[13] B. B. Brandenburg, "Liu and Layland and Linux: A blueprint for proper real-time tasks," *SIGBED Review*, September 2020. [Online]. Available: https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/

[14] B. B. Brandenburg and J. H. Anderson, "Feather-Trace: A lightweight event tracing toolkit," in *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2007, pp. 19–28.

[15] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[16] M. Cesati, R. Mancuso, E. Betti, and M. Caccamo, "A memory access detection methodology for accurate workload characterization," in *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015, pp. 141–148.

[17] J.-J. Chen and B. B. Brandenburg, "A note on the period enforcer algorithm for self-suspending tasks," *Leibniz Transactions on Embedded Systems*, vol. 4, no. 1, pp. 01:1–01:22, 2017.

[18] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, pp. 144–207, 2019.

[19] J. Chen, Z. Feng, J.-Y. Wen, B. Liu, and L. Sha, "A container-based DoS attack-resilient control framework for real-time UAV systems," in *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1222–1227.

[20] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset," in *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS)*, 2011, pp. 107–116.

[21] D. B. De Oliveira, R. S. De Oliveira, and T. Cucinotta, "Untangling the intricacies of thread synchronization in the PREEMPT_RT Linux kernel," in *Proceedings of the 22nd IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2019, pp. 1–9.

[22] D. B. De Oliveira and R. S. De Oliveira, "Timing analysis of the PREEMPT RT Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 789–819, 2016.

[23] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Ottawa Linux Symposium (OLS)*, 2006, pp. 209–224.

[24] F. X. Dormoy, "SCADE 6 – a model based solution for safety critical software development," in *Proceedings of the 4th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2008.

[25] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A real-time architecture design language for multi-rate embedded control systems," in *Proceedings of the 25th ACM Symposium On Applied Computing (SAC)*, 2010, pp. 527–534.

[26] N. Gehani and K. Ramamritham, "Real-time concurrent C: A language for programming dynamic real-time systems," *Real-Time Systems*, vol. 3, pp. 377–405, 1991.

[27] W. Giernacki, P. Kozierski, J. Michalski, M. Retinger, R. Madonski, and P. Campoy, "Bebop 2 quadrotor as a platform for research and education in robotics and control engineering," in *Proceedings of the 2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2020, pp. 1733–1741.

[28] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[29] 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.

[30] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018.

[31] T. Kessler, J. Bernhard, M. Buechel, K. Esterle, P. Hart, D. Malovetz, M. T. Le, F. Diehl, T. Brunner, and A. Knoll, "Bridging the gap between open source software and vehicle hardware for autonomous driving," in *Proceedings of the 30th IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 1612–1619.

[32] D. Kim, J. Eom, and C. Park, "L4oprof: a performance-monitoring-unit-based software-profiling framework for the L4 microkernel," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 69–76, 2007.

[33] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

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

[35] J. Lee, J. Wang, D. Crandall, S. Šabanović, and G. Fox, "Real-time, cloud-based object detection for unmanned aerial vehicles," in *Proceedings of the 1st IEEE International Conference on Robotic Computing (IRC)*, 2017, pp. 36–43.

[36] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the 11th Real-Time Systems Symposium (RTSS)*, 1990, pp. 201–209.

[37] H. Leppinen, "Current use of Linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[38] 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.

[39] 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.

[40] 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.

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

[42] S. Natarajan, M. Nasri, D. Broman, B. B. Brandenburg, and G. Nelissen, "From code to weakly hard constraints: A pragmatic end-to-end toolchain for Timed C," in *Proceedings of the 40th IEEE Real-Time Systems Symposium (RTSS)*, 2019, p. 167–180.

[43] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.

[44] 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.

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

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

[47] M. Perronet, M. Maida, C. Courtaud, and B. B. Brandenburg, "Work in progress: Automatic response-time analysis for arbitrary real-time Linux workloads," in *Proceedings of the 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 317–320.

[48] M. Pohlack, B. Döbel, and A. Lackorzynski, "Towards runtime monitoring in real-time systems," in *Proceedings of the 8th Real-Time Linux Workshop*, 2006.

[49] 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.

[50] K. R. Richter, "Compositional scheduling analysis using

standard event models," Ph.D. dissertation, TU Braunschweig, Germany, 2005.

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

[52] Tesla Motors, "Linux distribution for Tesla vehicles," https://github.com/teslamotors/linux, 2024.

[53] 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)*, 2000, pp. 101–104.

[54] L. Tung, "SpaceX: We've launched 32,000 Linux computers into space for Starlink internet," https://www.zdnet.com/article/spacex-weve-launched-32000-linux-computers-into-space-for-starlink-internet/, 2020.

[55] 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.

[56] E. Vallow, "RTOS and Linux - their evolving roles in aerospace and defense," https://www.lynx.com/embedded-systems-blog/rtos-linux-aerospace-defense, 2024.

[57] S. Vaughan-Nichols, "To infinity and beyond: Linux and open-source goes to Mars," https://www.zdnet.com/article/to-infinity-and-beyond-linux-and-open-source-goes-to-mars/, 2021.

[58] ——, "20 years later, real-time Linux makes it to the kernel - really," https://www.zdnet.com/article/20-years-later-real-time-linux-makes-it-to-the-kernel-really/, 2024.

[59] ——, "From earth to orbit with Linux and SpaceX," https://www.zdnet.com/article/from-earth-to-orbit-with-linux-and-spacex/, 2020.

[60] G. von der Brüggen, W.-H. Huang, and J.-J. Chen, "Hybrid self-suspension models in real-time embedded systems," in *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–9.