LiME

The Linux Real-Time Task Model Extractor

31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)
May 9, 2025

Björn Brandenburg
MPI-SWS, Germany

Cédric Courtaud
Huawei Technologies, France

Filip Marković
University of Southampton, UK

Bite Ye
MPI-SWS, Germany

[Authors listed in alphabetical order. Work carried out while affiliated with MPI-SWS.]

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

European Research Council
Established by the European Commission

LiME

https://lime.mpi-sws.org/

LiME in a nutshell:

no source code
& no static analysis of binary

blackbox threads running on Linux

online (= constant-memory) inference
& conservative w.r.t. observations

no user input
& no RT or kernel expertise required

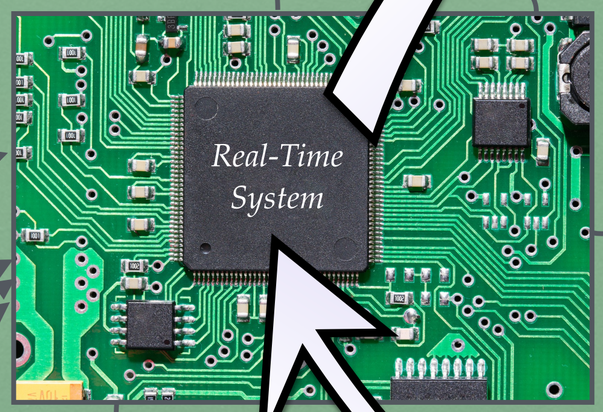
well-studied in the RT literature
& schedulability analysis available

established real-time task models

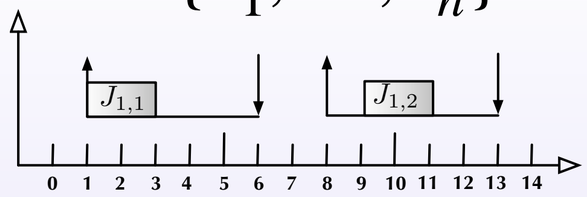
The Problem Being Solved

The *System-Model-Analysis-Prediction* Cycle

Cyber-Physical System (CPS)



$\tau = \{\tau_1, \ldots, \tau_n\}$



$\tau_l = (C_l, T_l)$ $\tau_i = (C_i, T_i, \Phi, J_i)$
$\tau_j = (C_j, \alpha_j(\Delta))$ $\cdots$ $\tau_k = (C_k, T_k, S_k)$

real-time task models

schedulability analysis

Scheduling theory
guides & validates
system implementation.

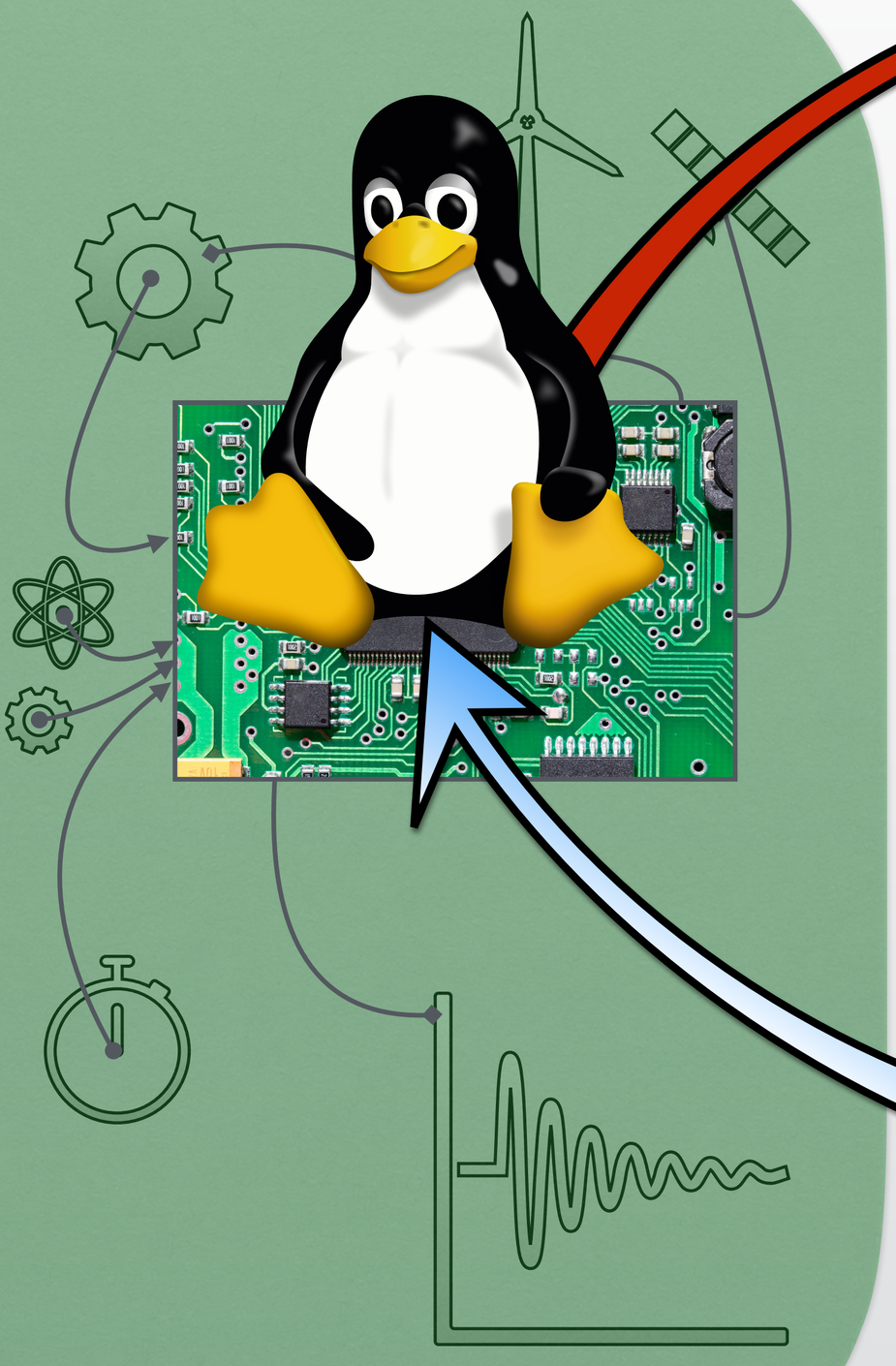
Bounds on

response times, data age, end-to-end latency, deadline miss rates, deadline-miss patterns, tardiness, buffer sizes, sampling jitter, synchronization delays…

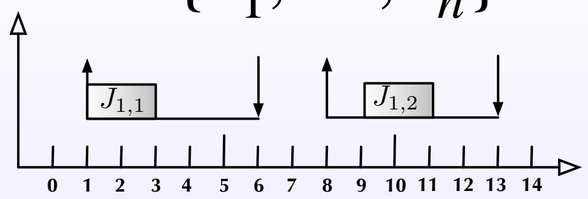
system behavior predictions

Linux has Become a Serious RT Platform

Cyber-Physical System (CPS)



$$\tau = \{\tau_1, \ldots, \tau_n\}$$



$\tau_l = (C_l, T_l)$ $\tau_i = (C_i, T_i, \Phi, J_i)$

$\tau_j = (C_j, \alpha_j(\Delta))$ $\cdots$ $\tau_k = (C_k, T_k, S_k)$

real-time task models

schedulability analysis

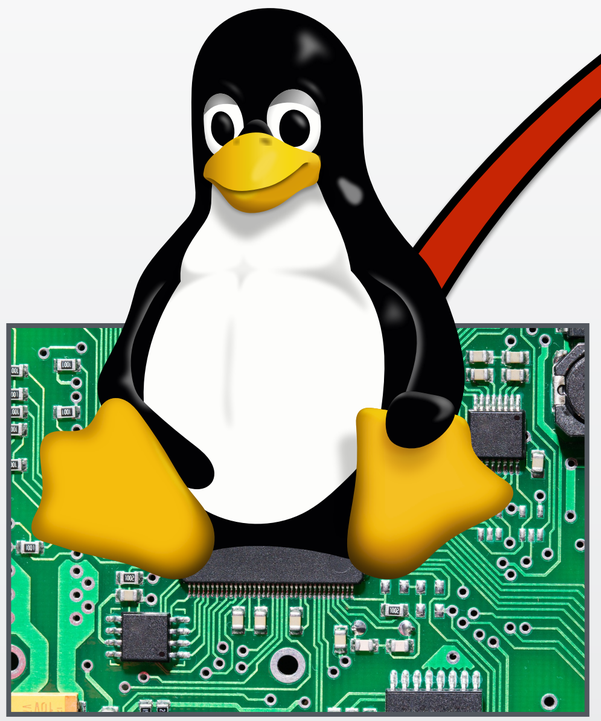
But what if we're using Linux…?

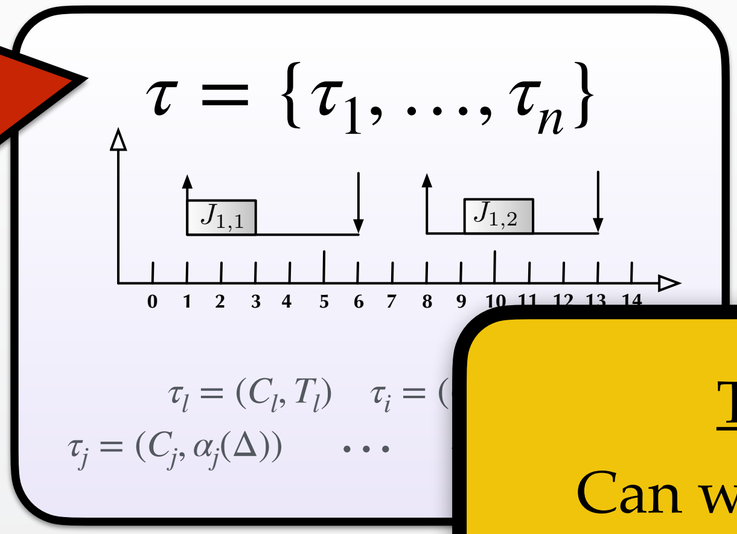
Bounds on

response times, data age, end-to-end latency, deadline miss rates, deadline-miss patterns, tardiness, buffer sizes, sampling jitter, synchronization delays…

system behavior predictions

The "Dual Expertise" Barrier





real-time task

State of the Art

➡ *No tool support whatsoever!*

Doing it the hard way:

1. Become familiar with Linux's low-level tracing facilities (*eBPF*).
2. Become familiar with Linux system call semantics *below* `libc` level.
   ➡ *Dozens of relevant syscalls and flags…*
3. Become familiar with applicable RT theory.
   ➡ *In particular, identify suitable task models that fit Linux thread behavior!*
4. Build and **test** your own in-house tooling.
   ➡ *Or a mess of buggy shell scripts…*

This Paper

Can we *fully automate* **Linux Model Extraction** and bridge this gap *once and for all*?

LiME

LiME in Action

Demo: Tracing *cyclictest*

```
$ sudo lime-rtw extract -o demo -- cyclictest -p fifo  -D 10

defaulting realtime priority to 2
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.08 0.02 0.00 1/214 567452

T: 0 (567452) P: 2 I:1000 C:   9991 Min:     18 Act:   54 Avg:   52 Max:      75

Results saved in demo.
```

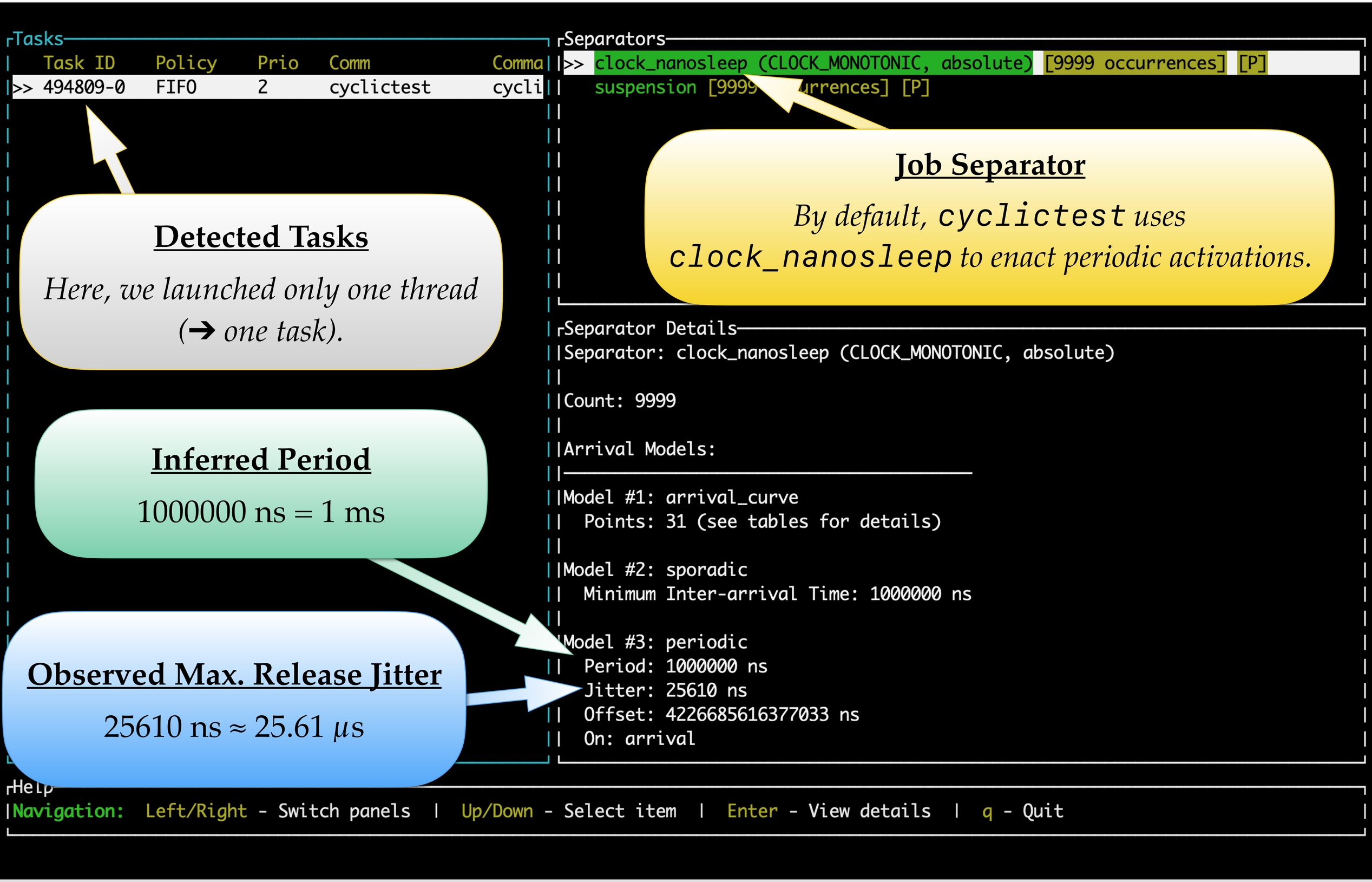
Normal output of *cyclictest*.

Online extraction of task models.
(Offline extraction also supported.)

JSON files containing task model parameters.

Let's have a look with LiME's viewer...

`lime-rtw view /tmp/demo`



Demo: Tracing *ping*

Can trace <u>arbitrary</u> (and non-RT) tasks.

```
$ sudo lime-rtw extract --best-effort -o demo-ping
                        -- ping -c 1000 -i 0.1 contact.mpi-sws.org

PING contact.mpi-sws.org (139.19.171.199) 56(84) bytes of data.
64 bytes from contact.mpi-sws.org (139.19.171.199): icmp_seq=1 ttl=59 time=1.27 ms
64 bytes from swscontact.mpi-sws.org (139.19.171.199): icmp_seq=2 ttl=59 time=1.30 ms
64 bytes from swscontact3.mpi-sws.org (139.19.171.199): icmp_seq=3 ttl=59 time=1.30 ms
64 bytes from swscontact3.mpi-sws.org (139.19.171.199): icmp_seq=4 ttl=59 time=1.32 ms
[…]
64 bytes from swscontact3.mpi-sws.org (139.19.171.199): icmp_seq=1000 ttl=59 time=1.28 ms

--- contact.mpi-sws.org ping statistics ---
1000 packets transmitted, 1000 received, 0% packet loss, time 100508ms
rtt min/avg/max/mdev = 1.248/1.309/2.106/0.058 ms

Results saved in demo-ping.
```

lime-rtw view /tmp/demo-ping

```
┌Tasks──────────────────────────────────────────────────────────────────────────┐
│   Task ID     Policy     Prio     Comm                 Comma                  │
│>> 494691-0    CFS        -        ping                 ping                   │
└───────────────────────────────────────────────────────────────────────────────┘
┌Separators─────────────────────────────────────────────────────────────────────┐
│>> suspension [3001 occurrences] [P]                                           │
│   recvmsg [999 occurrences] [P]                                               │
│   poll [998 occurrences]                                                      │
└───────────────────────────────────────────────────────────────────────────────┘
┌Separator Details──────────────────────────────────────────────────────────────┐
│Separator: suspension                                                          │
│                                                                               │
│Count: 3001                                                                    │
│                                                                               │
│Arrival Models:                                                                │
│──────────────────────────────────────────────────                             │
│Model #1: arrival_curve                                                        │
│  Points: 31 (see tables for details)                                          │
│                                                                               │
│Model #2: sporadic                                                             │
│  Minimum Inter-arrival Time: 1100041 ns                                       │
│                                                                               │
│Model #3: periodic                                                             │
│  Period: 33520000 ns                                                          │
│  Jitter: 118511893 ns                                                         │
│  Offset: 4226024900838459 ns                                                  │
│  On: release                                                                  │
└───────────────────────────────────────────────────────────────────────────────┘
┌Help───────────────────────────────────────────────────────────────────────────┐
│Navigation:  Left/Right - Switch panels  |  Up/Down - Select item  |  Enter - View details  |  q - Quit │
└───────────────────────────────────────────────────────────────────────────────┘
```

Suspension Separator

Each thread wake-up is a new activation.

Release Curve

Suitable for complex activation behavior.

lime-rtw view /tmp/demo-ping

```
┌Detailed View: suspension (Scroll: Up/Down)──────────────────────────────────────────────────────────────────────────────┐
│                                                                                                                         │
│ Separator: suspension   Count: 3001                                                                                     │
│                                                                                                                         │
│                                                                                                                         │
│                                                                                                                         │
│ Details                                                                                                                 │
│                                                                                                                         │
│   Summary:                                                                                                              │
│   WCET range: 6236790 - 17705610 ns                                                                                     │
│                                                                                                                         │
│ ⭐ Arrival Curve Table:                                                                                                 │
│   Shows the minimum and maximum leng   of intervals containing N job releases                                           │
│                                                                                                                         │
│   Number of jobs | Minimum interval | Maximum interval                                                                  │
│   ──────────────────────────────────────────────────────                                                                │
│                 2 |          1100041 |          98350299                                                               │
│                 3 |          2471339 |          99767058                                                               │
│                 4 |         10642695 |         101734014                                                               │
│                 5 |         19561043 |         199219047                                                               │
│                 6 |        102438821 |         200622492                                                               │
│                 7 |        107209759 |         202527597                                                               │
│                 8 |        116128107 |         299959667                                                               │
│                 9 |        202985815 |         301397149                                                               │
│                10 |        207862863 |         303244402                                                               │
│                11 |        216781211 |         400678491                                                               │
│                12 |        303536494 |         402182860                                                               │
│                13 |        308739186 |         403955856                                                               │
│                14 |        317657534 |         501379167                                                               │
│                                                                                                                         │
└─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
┌Help─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│Detail View Mode  Press ESC to return to the main view.                                                                  │
└─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Minimum Separation ("delta min")

Shortest interval in which N jobs arrive.

➔ *enables* ***response-time analysis (RTA)***

How Does LiME Work?

The LiME Pipeline

Blackbox Threads

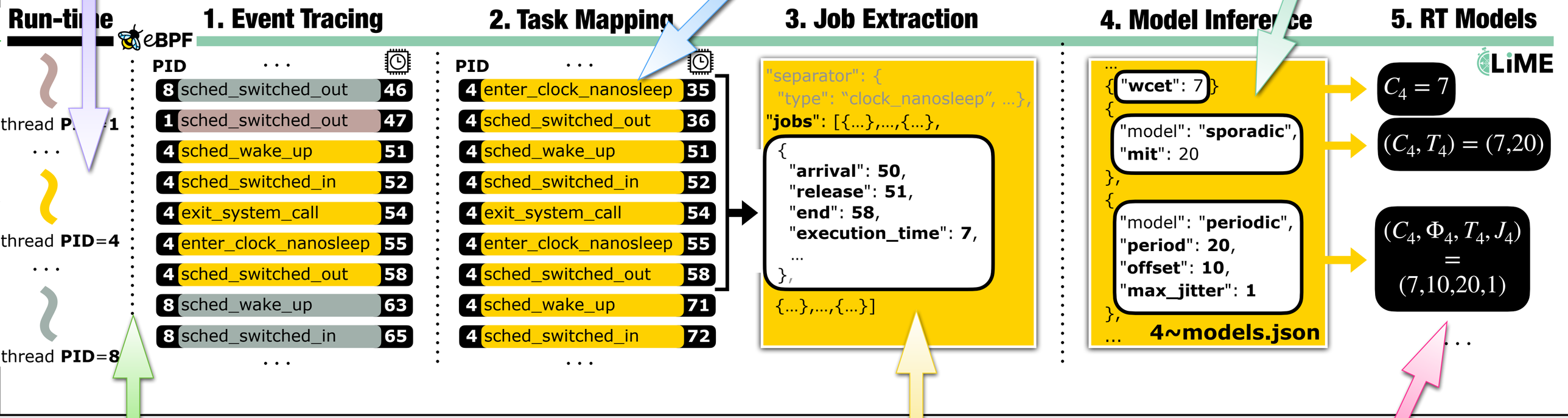
any code using any system calls in any order, without access to source code or static analysis of binary

Split Event Stream by Task

One thread can correspond to *multiple* tasks over time (see paper for details).

Online Model Inference

Infer task parameters from stream of jobs with *bounded memory.*



Kernel Instrumentation

*installs **65 eBPF probes*** in scheduler, selected system calls, etc. monitoring *all threads*

Identify Individual Task Activations (= Jobs)

RT task models assume repeatedly activated tasks:

task = stream of jobs

Output

Conservative models: each model must reflect *all* observations.

Key Idea: Job Separators

Target real-time task models: *"a **task** is a sequence of **jobs**"*

But Linux is ***jobless***: each thread is an *arbitrary* sequence of system calls!

Fundamental Challenge

Where does one job end and the next start?

(1) LiME Insight

On Linux, a job (= one task activation) *always* starts with a *potentially blocking* system call.

*A real-time task **must wait** for next **event** or **passage of time** ➔ blocking system calls.*

(2) LiME Heuristic

Recurrent real-time tasks
≈ *"doing the same thing over and over"*

➔ *typically every job starts with the **same** system call, so LiME assumes this.*

Job Separator Examples

Time-driven:

```
struct timespec next = now();
while (true) {
  /* periodic activation */
  clock_nanosleep(&next);
  do_something();
  timespec_add(&next, PERIOD);
}
```

➔ job separator: `clock_nanosleep`

Event-driven:

```
int fd = open_udp_socket(PORT);
while (true) {
  /* sporadic activation */
  int nbytes = read(fd, &buf);
  if (nbytes <= 0) break;
  do_something(buf, nbytes);
}
```

➔ job separator: `read(fd)`

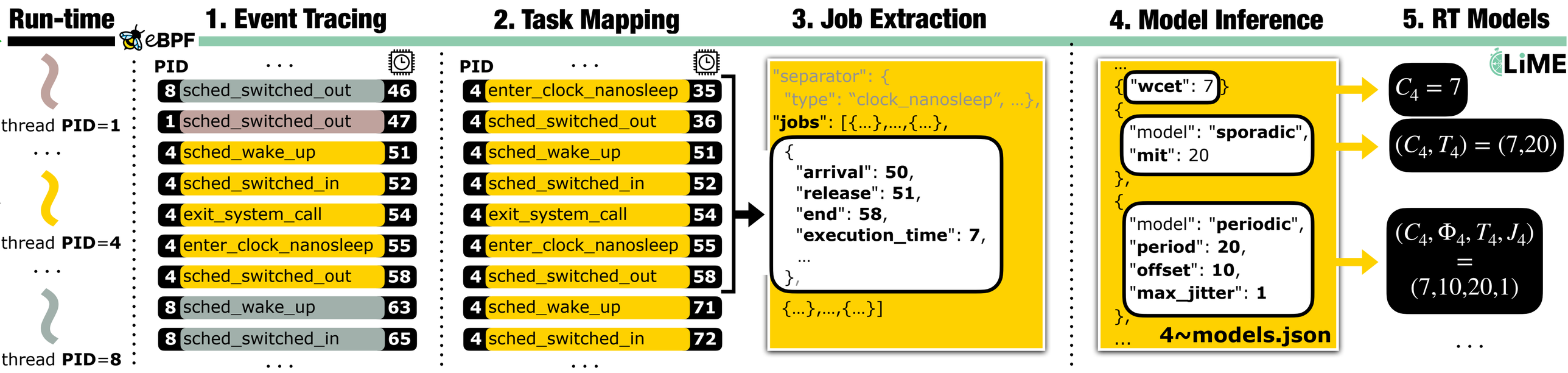
In both examples:

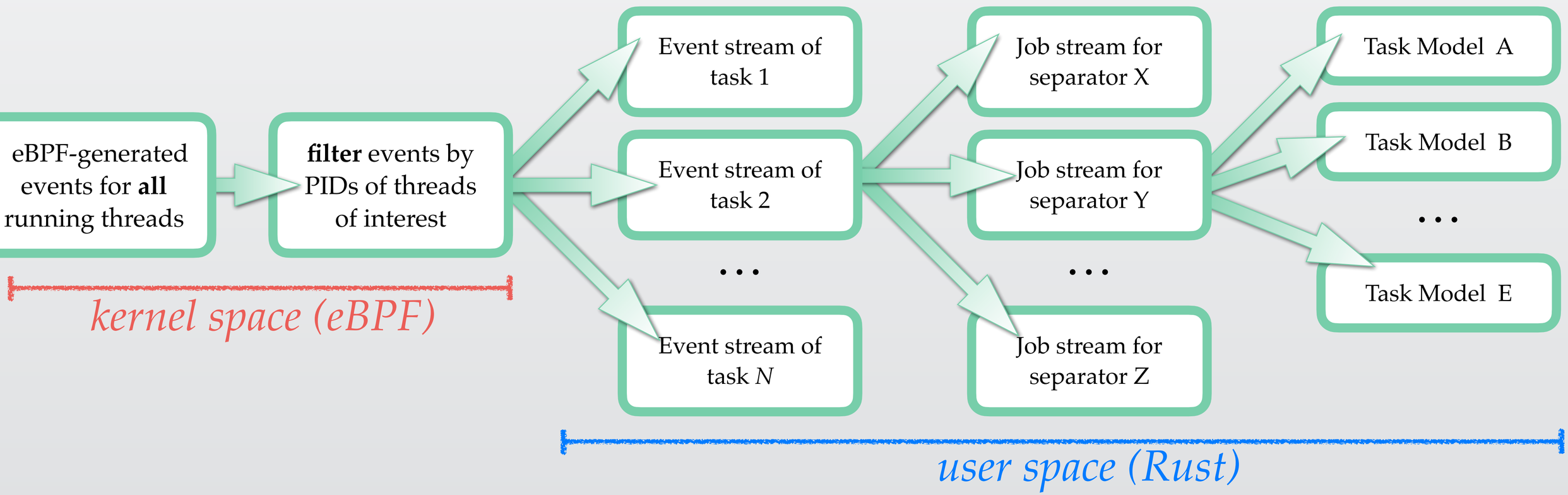
- one job = one loop iteration
- each job starts with *potentially* blocking system call
- all jobs of one task start with the *same* system call

Inherent ambiguity:

- task may execute many different system calls in `do_something()`
- There are potentially *many* job separator *candidates*.

Dataflow from Kernel to Model Extraction





LiME design choice: expose job-separator ambiguity to user.

Supported Task Models

Supported Task Models

Arrival Models

- **sporadic**
  (Mok, 1983)
- **arrival curves**
  (Thiele *et al.*, 2000; Richter, 2005)
- **periodic + jitter & offset**
  (Liu & Layland, 1973; Leung & Merril, 1980; Audsley *et al.*, 1993)

Execution Time
(measurement-based)

- **WCET**
  (Liu & Layland, 1973)
- **WCET(*n*)**
  (Quinton et al., 2012)

Self-Suspension

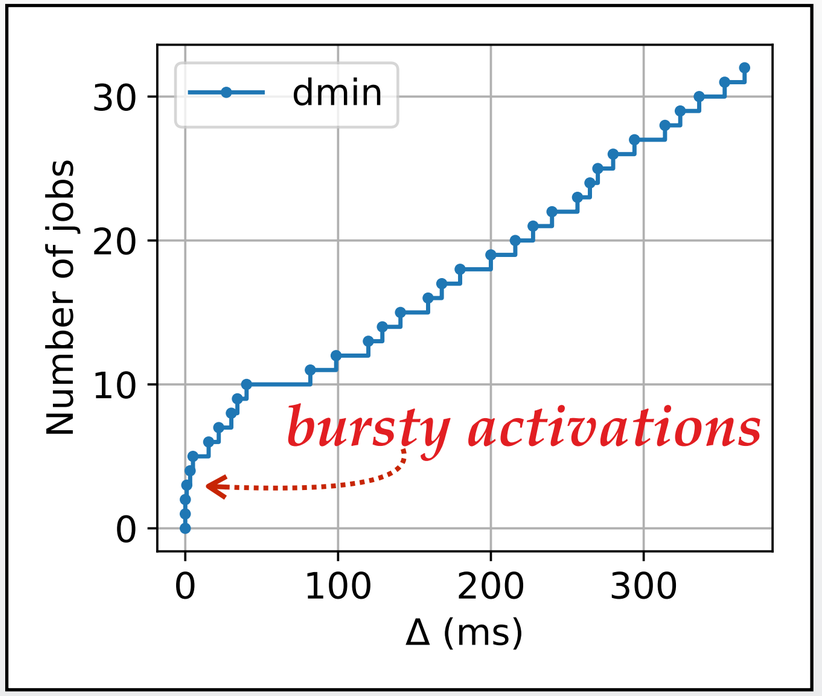
- **dynamic**
  (Ming, 1994)
- **generalized segmented**
  (von der Brüggen et al., 2017)

Why Curve-Based Models?

Why not stick with "tried and true" scalar model parameters?

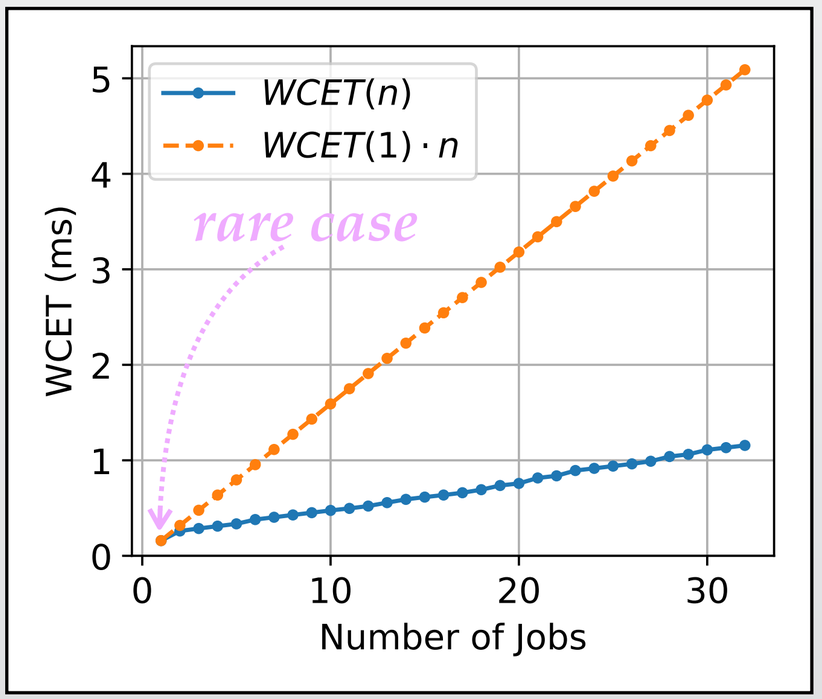
Example: Linux kernel threads

- ➡ `migration` threads (one per core)
- ➡ assist Linux process scheduler
- ➡ execute at **max. RT priority**
- ➡ their CPU use must be accounted for



How much CPU use in 300 ms?

WCET	Arrivals	Bound
scalar	scalar	> **1500** ms
curve	scalar	> **340** ms
scalar	curve	≈ **4.3** ms
curve	curve	≈ **0.99** ms



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?

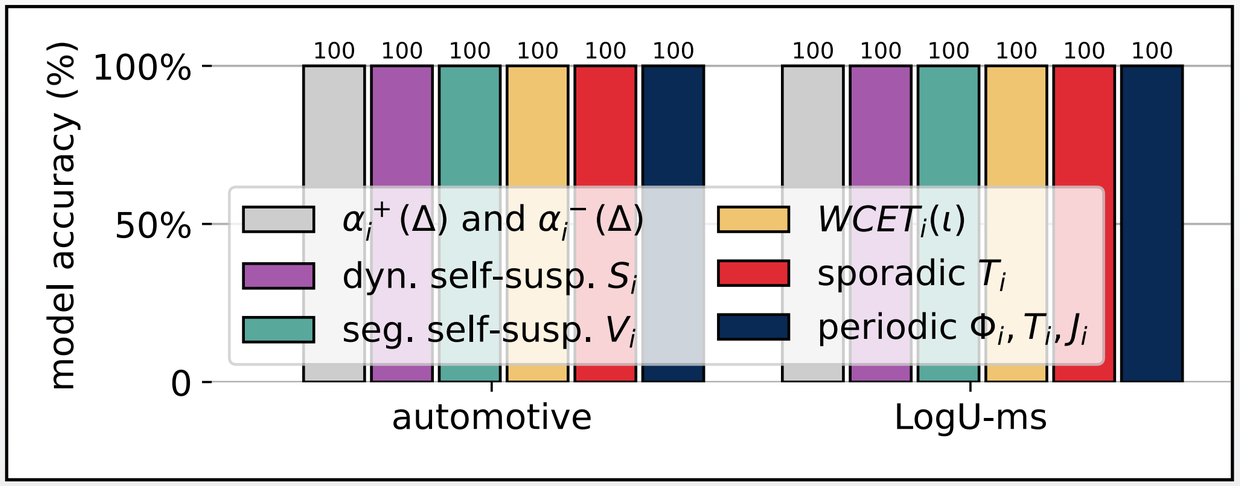
This paper. We show that the answer to each of these questions is 'yes' by presenting LiME, the **Li**nux real-time task **M**odel **E**xtractor (Sec. III). LiME is a dynamic introspection tool that maps sequences of low-level thread-kernel interactions (Sec. II-A), observed via the kernel's *eBPF* tracing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

Evaluation in the Paper

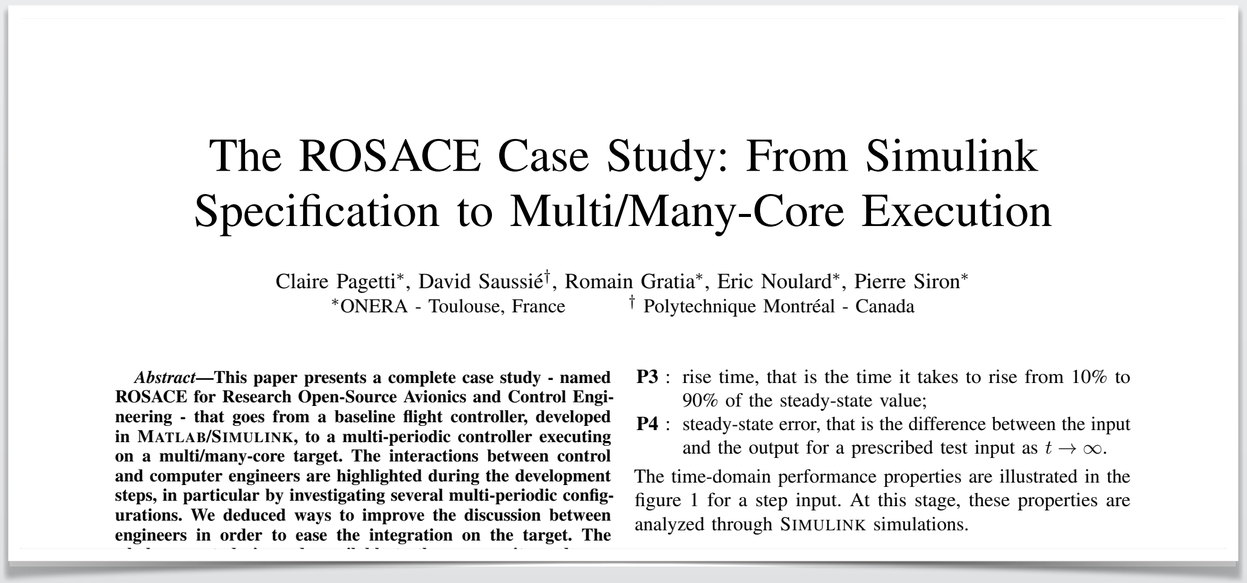
LiME is Accurate and Effective



<u>100% Inference Accuracy</u>

➔ *automotive periods*

➔ *random periods (ms-granularity)*



The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution

Claire Pagetti*, David Saussié†, Romain Gratia*, Eric Noulard*, Pierre Siron*
*ONERA - Toulouse, France † Polytechnique Montréal - Canada

***Abstract*—This paper presents a complete case study - named ROSACE for Research Open-Source Avionics and Control Engineering - that goes from a baseline flight controller, developed in MATLAB/SIMULINK, to a multi-periodic controller executing on a multi/many-core target. The interactions between control and computer engineers are highlighted during the development steps, in particular by investigating several multi-periodic configurations. We deduced ways to improve the discussion between engineers in order to ease the integration on the target. The**

P3 : rise time, that is the time it takes to rise from 10% to 90% of the steady-state value;

P4 : steady-state error, that is the difference between the input and the output for a prescribed test input as $t \to \infty$.

The time-domain performance properties are illustrated in the figure 1 for a step input. At this stage, these properties are analyzed through SIMULINK simulations.

Pagetti *et al.*, RTAS 2014

<u>ROSACE Case Study</u>

➔ *longterm observation*

➔ *detect subtle timing **drift***

➔ *validate fix*

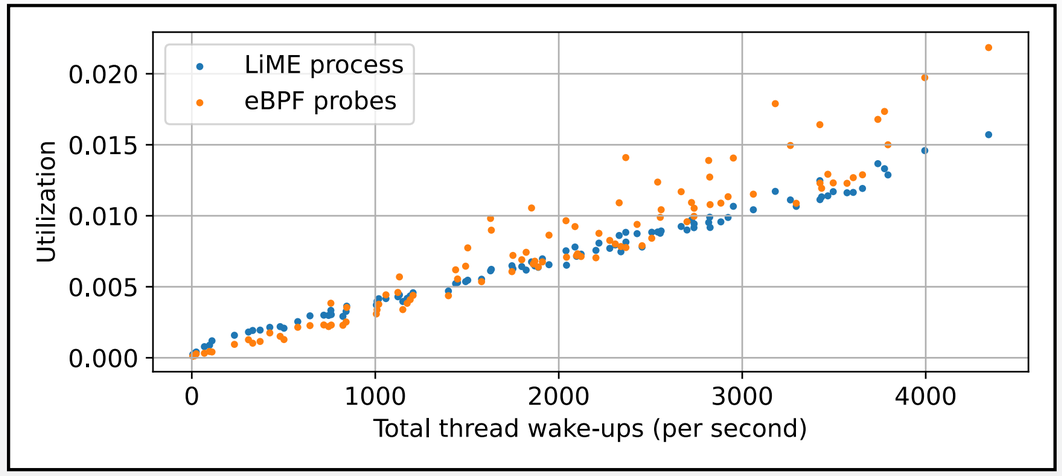
LiME Causes Only Low Overhead

<u>CPU overhead</u>

Kernel (eBPF): $< 2.5\%$ (of 4 cores)

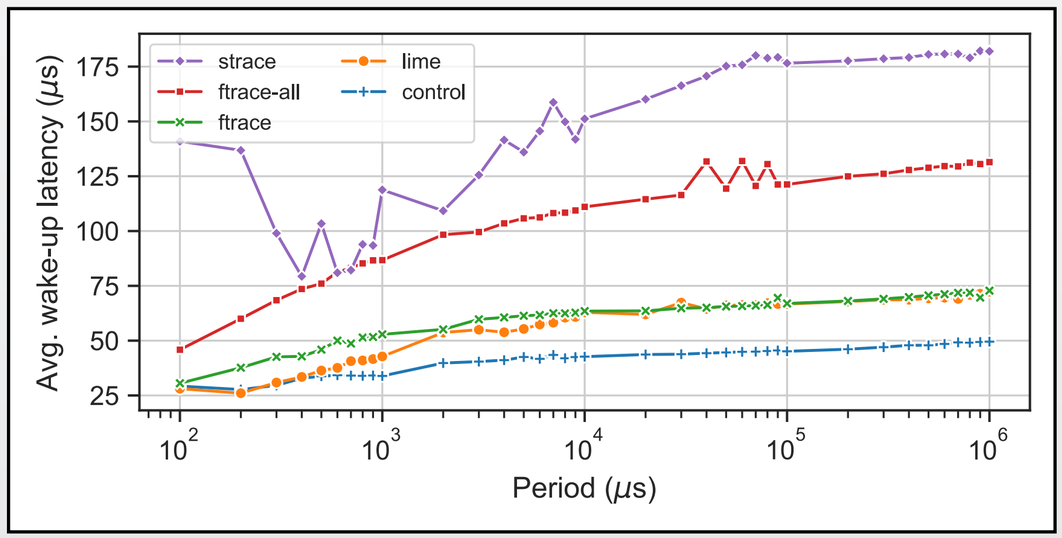
User space: $< 2.0\%$

(*on a Raspberry Pi 4 "Model B"*)



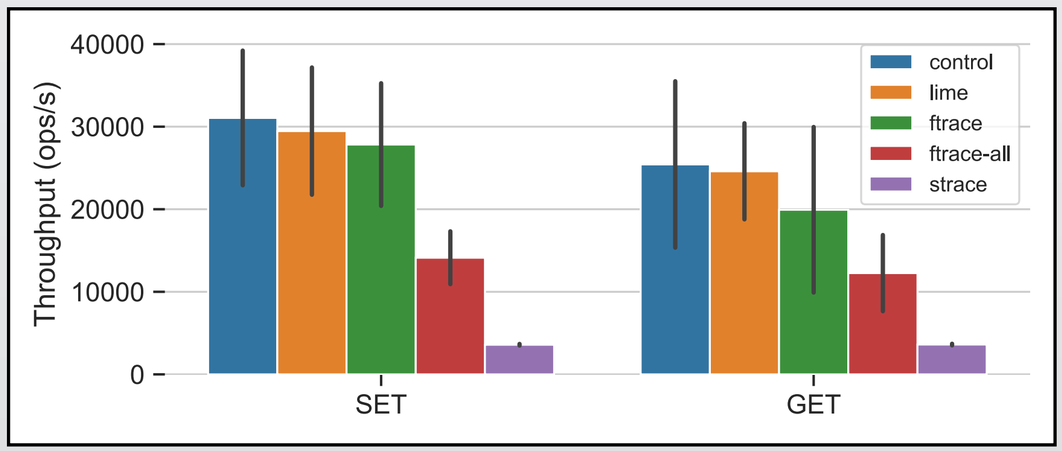
<u>`cyclictest` latency benchmark</u>

➔ *avg. latency impact ≈ 25µs*



<u>*`Redis`* key-value server</u>

➔ *minor throughput impact*



Conclusion

What Can LiME Do For You?

If you are doing **systems** research:

➔ *Integrate schedulability analysis (almost) "for free"*

If you are working with a **real system**:

➔ *Monitor, understand, debug, validate its timing behavior*

If you work on **schedulability analysis**:

➔ *Get some real models rather than just random numbers*

➔ *Demonstrate that your model assumptions are viable*

If you **teach**:

➔ *Let students explore RT foundations hands-on*

LiME

https://lime.mpi-sws.org/

blackbox threads running on Linux

established real-time task models

→ fully automated
→ online inference
→ highly accurate
→ low overheads

Thank you for your attention! Any Questions?

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

European Research Council
Established by the European Commission