From Estimates to Guarantees: Verified Exceedance Analysis on Commodity Hardware

Kimaya Bedarkar, Bite Ye, Deepak Garg, Derek Dreyer, and Björn B. Brandenburg
Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany

***Abstract*—It is a longstanding challenge of real-time systems research to develop high-assurance verification of response-time guarantees for real scheduler implementations. Recent work on RefinedProsa has made an important step towards this goal: it shows how to verify response-time bounds for an interrupt-free scheduler by combining mechanized verification of its C implementation (in RefinedC) with a mechanized proof of correctness of the underlying schedulability analysis (in Prosa), all within the Rocq proof assistant. However, RefinedProsa makes the simplifying assumption that tasks, as well as basic sections of the scheduler code, exhibit known worst-case execution times (WCETs). Unfortunately, it is notoriously difficult to statically bound WCETs on commercial off-the-shelf (COTS) platforms, thus undermining RefinedProsa's rigorous guarantees.**

In this paper, we eliminate RefinedProsa's reliance on WCETs by leveraging the recently introduced methodology of *exceedance analysis*. Exceedance analysis replaces WCETs with *nominal execution times* (NETs), which can be easily measured for any commodity hardware, and formulates the response-time bound not purely as a static guarantee but as a function of the runtime *exceedance* (execution time overrun on top of the NETs). By integrating exceedance analysis into RefinedProsa, we are able to discharge the timing assumptions underlying its mechanized proof in a real, Linux-based system such that the system's resilience to transient NET overruns can be systematically assessed.

We substantiate the benefits of our approach with a comprehensive case study that runs Rössl (the C scheduler verified by RefinedProsa) on COTS hardware. We demonstrate the soundness of the resulting bounds under varying degrees of exceedance, assess their tightness, and show how several intuitively interpretable measures of a system's temporal robustness can be derived from the proposed verified exceedance analysis.

I. Introduction

Real-time systems are frequently deployed in safety-critical domains, where correctness is paramount. In such settings, formal verification provides guarantees that go beyond empirical testing by rigorously proving that a system satisfies its specified properties under all possible executions. Consequently, the real-time systems community has increasingly embraced formal methods to establish schedulability guarantees [2, 7, 9, 11, 17, 18, 26, 41].

Among these approaches, a prominent line of research is embodied by the Prosa library [32]. In Prosa, an interactive theorem prover (here, Rocq [1]) is used to formally specify system models and mechanize their timing analysis, with all guarantees established through machine-checked proofs.

Until recently, Prosa was used primarily to verify schedulability analyses of abstract system models. Going a significant step further, Bedarkar et al. [3] recently proposed a novel technique for combining mechanized analyses of real-time task models with formally verified properties of concrete code. The resulting *RefinedProsa* approach [3], demonstrated on a small C scheduler (called *Rössl*), yields a fully formal guarantee that the concrete system implementation satisfies the verified response-time bounds.

Central to RefinedProsa's proof methodology is the *worst-case execution time* (WCET) assumption, which RefinedProsa uses to give a *timed semantics* to concrete code. Specifically, the code of Rössl is divided into a series of distinct, loop-free segments and OS-provided system calls, referred to as *basic actions*. Each basic action is assumed to have a WCET. The scheduler's timing behavior is then modeled as a sequence of basic actions and their corresponding WCETs. Consequently, the soundness of the overall response-time guarantees—and thus the practical applicability of RefinedProsa—critically hinges on the validity of these WCETs.

Unfortunately, WCETs are notoriously difficult to justify on modern hardware. Modern processors use features such as out-of-order pipelines, speculative execution, and shared cache hierarchies, which complicate precise timing analysis [42]. This difficulty is reflected in the limited availability of WCET analysis tools: even comparatively simple commercial off-the-shelf (COTS) platforms are often not fully supported by existing static-analysis-based analyzers [19, 27, 30, 36, 47].

The unavailability of WCETs can significantly compromise any response-time guarantees that rely on them, as even small deviations from assumed WCETs can trigger disproportionately large and potentially nonlinear, cascading effects.

To systematically account for such runtime overruns, recent work on *exceedance analysis* [47] investigates system behavior when only *nominal execution times* (NETs) are available. NETs are empirically derived estimates that capture typical execution demand but are not sound upper bounds (*e.g.*, observed maxima or high-percentile measurements). Exceedance analysis addresses the fact that NET bounds can be violated at runtime by modeling overruns explicitly as *transient execution-time exceedance* and expressing response-time bounds in terms of both NETs and the accumulated total exceedance, enabling reasoning about how overruns propagate through the schedule.

While these ideas hold promise for eliminating the WCET assumption, exceedance analysis is not directly applicable within RefinedProsa, for two main reasons. First, existing work [47] develops exceedance-based analyses using pen-and-paper proofs, whereas RefinedProsa derives its strong guarantees through mechanization. Second, existing exceedance analyses apply to idealized system models and do not explicitly

model any implementation-level effects such as scheduler overheads or transient deviations from ideal scheduling behavior. RefinedProsa, in contrast, explicitly accounts for these effects, but does so using WCET bounds for basic actions, placing it fundamentally at odds with exceedance-based reasoning.

Contributions. In this paper, we address these challenges and integrate exceedance analysis into RefinedProsa to obtain a formally verified characterization of the system's timing behavior without relying on statically derived WCETs. Removing the WCET assumption paves the way for the first instantiation of the analysis on COTS platforms. To this end, we make the following technical contributions:

First, in Sec. II, we derive from first principles an overhead- and exceedance-aware response-time bound for Rössl. In contrast to Zini et al.'s original analysis [47], our derivation explicitly accounts for runtime overheads and differs structurally from their bound in small but important details. This is because, in Zini et al.'s idealized setting, all activations are processed strictly in order, which Rössl cannot guarantee in practice.

Second, in Sec. III-A, we revise the proof methodology pioneered by RefinedProsa [3] to represent scheduler overheads as extensions of tasks or as pseudo tasks, yielding a model compatible with exceedance analysis. This key model simplification required major design changes and a thorough refactoring of the entire RefinedProsa proof pipeline.

Third, in Sec. III-B, we mechanize exceedance-aware response-time analysis by adopting a novel perspective on NET overruns that minimizes proof effort by modeling all exceedance as *blackouts* (*i.e.*, supply restrictions as previously studied in the context of reservation-based scheduling [38], for which there exists prior support in Prosa [32]). This represents the first formal verification of NET-based reasoning.

Fourth, in Sec. IV, we present a case study evaluating Rössl on a commodity multicore platform, namely a Raspberry Pi 5 running Linux. We report on empirical assessments of the soundness and tightness of the verified analysis, and propose interpretable measures of a system's temporal robustness that can be readily understood by engineers in practice. Our case study is the first evaluation of NET-based reasoning on real hardware (Zini et al. [47] reported only a synthetic case study).

The supplementary material [4] includes the complete Rocq mechanization, the experimental setup, and an appendix with additional proof details.

II. Rössl and its Response-Time Bound

We begin by introducing Rössl, the system under analysis, originally due to Bedarkar et al. [3], before presenting the exceedance-aware response-time bound that we prove for it.

Rössl [3] is a single-threaded, nonpreemptive, fixed-priority scheduler for POSIX systems implemented in the *C* programming language. Loosely inspired by ROS 2's default executor, it invokes callbacks in response to incoming network messages.

More specifically, Rössl is initialized with a set of datagram sockets (*i.e.*, typically UDP sockets) and a map of message IDs to user-defined callback functions and their assigned priorities.

```
1  int roessl_main(struct fd_scheduler *fds) {
2   while (1) {
3    // receive jobs on all sockets
4    check_sockets_until_empty(fds);
5    M_Selection();
6    // get highest-priority job
7    struct job *j = npfp_dequeue(&fds->sched);
8    if (!j) { // wait for new input
9     M_Idling();
10   } else {
11    M_Dispatch(j);
12    // execute the job
13    npfp_dispatch(&fds->sched, j);
14    free(j); }}}
```

Fig. 1: The scheduling loop of Rössl [3]. Annotations shown in light blue are discussed in subsequent sections.

After initialization, Rössl enters its main loop, as shown in Fig. 1 (ignore the annotations in light blue for now; we revisit them later). In Line 2, the scheduler first polls all registered sockets and ingests all queued messages into the scheduler's state. To this end, it copies each available message from the kernel into a message buffer in userspace, inspects the message ID included in the message's payload, and then inserts the message buffer into the scheduler's ready queue according to the priority set for the message ID. We refer to this part of Rössl's main loop as the *polling phase*. A polling phase ends once a nonblocking read operation has failed on each socket, indicating that no further messages are currently available.

The scheduler then enters the *execution phase* (Line 7), where it selects the highest-priority pending callback for execution, which we refer to as a *job* (following established real-time scheduling terminology). If such a job exists, Rössl dispatches the corresponding callback (Line 13), which runs to completion before Rössl begins the next loop iteration. Otherwise, the scheduler enters an *idling phase* and resumes polling.

Job response times. The main timing metric of interest is how quickly the system reacts to incoming messages. More precisely, our objective is to bound each job's *response time*, which is the time elapsed between a message *arriving* on a socket (*i.e.*, the earliest point in time at which it could be read by Rössl's polling phase) and the *completion* of the associated callback invocation. This response time includes not only the execution of other jobs, but also scheduler overheads such as the time spent by Rössl on polling, checking "empty" sockets, and job selection. Accounting for these sources of overhead is a key challenge in our formal proof, as discussed in Sec. III.

Basic actions. To reason about the timing behavior of Rössl, the scheduler implementation is divided into a set of *basic actions* (Table I). Intuitively, a basic action is a small, loop-free section of scheduler code corresponding to one logically distinct activity, such as invoking a system call to read a message from a socket or selecting the highest-priority pending job. The precise formalization of basic actions and their connection to the implementation are introduced later; for now, it is sufficient to view them as the atoms of our timing model.

Most importantly, each basic action is characterized by a NET. Recall that a NET is an empirically estimated (*i.e.*, unsound) upper "bound" on the maximum execution time typically required to complete a given activity, such as one of

TABLE I: Basic actions and their corresponding NETs.

Basic Action	NET	Description
READ *None*	$NetFR$	Failed attempt to read a message
READ j	$NetSR$	Read of a message of job j
SEL j	$NetSel$	Selection of job j for execution
SEL *None*	$NetSel$	Failed attempt to select a job
DISP j	$NetDisp$	Dispatch of the callback for j
COMPL j	$NetCompl$	Return control to scheduler after executing the callback for job j
IDLING	$NetIdling$	Return to polling after failed selection

the basic actions. In other words, the basic actions define the execution paths that we must *measure* when Rössl is deployed on actual hardware, as discussed further in the context of our case study in Sec. IV. Conversely, our formal proof is parametric in the concrete NETs associated with each basic action, as listed in Table I. Addressing the issue that NETs are not strict bounds and can be violated at runtime is a second major complication that sets our formal proof apart from earlier mechanized response-time analyses.

A. System Model

As discussed above, a job's response time is determined by its own execution time and two sources of delay: the execution of other callbacks and Rössl's scheduling overheads.

The callbacks. The main workload is the set of user-defined callbacks (*i.e.*, message handlers), where each callback corresponds to one message ID accepted by the system. We let $\mathcal{CB} \triangleq \{\chi_1, \ldots, \chi_n\}$ denote the set of callbacks managed by Rössl. Each callback $\chi_i \in \mathcal{CB}$ is characterized by a fixed priority $\pi_i \geq 1$ (where numerically smaller values correspond to higher scheduling priority), a per-invocation NET $NetCB_i$, and an *arrival curve* α_i. The arrival curve $\alpha_i(\Delta)$ is a monotonically nondecreasing function that bounds the maximum number of arrivals of messages handled by χ_i in any interval of length Δ, where $\alpha_i(0) = 0$. We measure time in terms of discrete clock cycles, and let $\epsilon \triangleq 1$ denote the smallest integral unit of time.

Overhead accounting. Due to the impact of scheduling overheads, we cannot analyze $\mathcal{CB}$ directly. Rather, we derive a *modified task set* $\mathcal{M}$ from $\mathcal{CB}$ by combining two well-established overhead accounting methods [25]: *execution-time inflation* and the addition of *pseudo tasks* representing system activities. The resulting task set $\mathcal{M}$ is then analyzed to obtain response-time bounds that reflect all sources of delay, including all overheads. This approach allows us to reason about NET exceedance—which can manifest during both regular callback execution and overhead-inducing basic actions—in a uniform manner, which greatly simplifies our formal proof.

To this end, we construct $\mathcal{M} \triangleq \{\tau_i^\chi \mid \chi_i \in \mathcal{CB}\} \cup \{\tau^{read}, \tau^{idle}\}$, where τ^{read} and τ^{idle} are two pseudo tasks that represent Rössl's polling and idle phases, respectively, and each τ_i^χ represents the callback χ_i after execution-time inflation. To abstract over the two kinds of *tasks* in $\mathcal{M}$, we let τ denote an arbitrary element of $\mathcal{M}$. Additionally, we let $P^{ovh}(\tau)$, $\alpha^{ovh}(\tau, \Delta)$, and $C^{ovh}(\tau)$ denote τ's respective priority, arrival curve, and NET, which we define below.

The magnitude of certain overheads, such as the polling phase, depends on the number of sockets polled by Rössl, which we denote by s in the following.

Execution-time inflation. Overheads that always occur immediately before or after the execution of a specific callback are folded into the effective cost of the corresponding task. Therefore, the derived NET of a callback task τ_i^χ is defined as $C^{ovh}(\tau_i^\chi) \triangleq s \times NetFR + NetSel + NetDisp + NetCB_i + NetCompl + (s-1) \times NetFR$, based on the following intuition.

First, $s \times NetFR$ represents the final iteration of the polling loop in which no messages are found (thereby terminating the polling phase) just before χ_i executes. Second, $NetSel$ and $NetDisp$ account for the cost of selecting and dispatching χ_i, respectively. Third, $NetCB_i$ is simply the cost of executing χ_i itself, and $NetCompl$ accounts for the cost of returning control to Rössl after χ_i completed. Fourth, the final charge of $(s-1) \times NetFR$ accounts for the possibility of Rössl checking up to $s-1$ "empty" sockets before finding the next message.

The priority and arrival curve are not modified, and hence $P^{ovh}(\tau_i^\chi) \triangleq \pi_i$ and $\alpha^{ovh}(\tau_i^\chi, \Delta) \triangleq \alpha_i(\Delta)$.

Pseudo tasks. The reading task τ^{read} models read overheads and therefore has the highest effective priority: $P^{ovh}(\tau^{read}) \triangleq 0$. This reflects that, in Rössl's main loop, reading new messages takes precedence over executing callbacks.

Since every message arrival will trigger a read operation, the reading task's derived arrival curve is the sum of the arrival curves of all callbacks: $\alpha^{ovh}(\tau^{read}, \Delta) \triangleq \sum_{\chi_i \in \mathcal{CB}} \alpha_i(\Delta)$.

Finally, its NET is given by $C^{ovh}(\tau^{read}) \triangleq (2s-1) \times NetFR + NetSR$, based on the following rationale. In the worst case, Rössl's polling phase checks all s sockets to discover only a single message. The cost of reading this message is accounted for by $NetSR$; the unproductive checks of the $s-1$ other sockets incur a cost of $NetFR$ each. The factor of 2 arises because the polling phase ends only after all s sockets have been observed to no longer yield any messages.

The idling task τ^{idle} models the overhead incurred while determining that no pending work exists and therefore receives the lowest priority: $P^{ovh}(\tau^{idle}) \triangleq 1 + \max\{\pi_i \mid \chi_i \in \mathcal{CB}\}$.

Its arrival curve plays no role in the analysis; we therefore use the identity arrival curve $\alpha^{ovh}(\tau^{idle}, \Delta) \triangleq \Delta$ as a placeholder to express that idling may occur arbitrarily often.

Finally, the idling task's NET is $C^{ovh}(\tau^{idle}) \triangleq (2s-1) \times NetFR + NetSel + NetIdling$, which represents a scenario in which all s sockets are found to be "empty," followed by a failed selection, one execution of the idle loop, and possibly $s-1$ unproductive reads before the next message is discovered.

A significant fraction of our formal proof is dedicated to establishing that the above construction of $\mathcal{M}$ indeed safely over-approximates the worst-case impact of scheduling overheads in Rössl. With all delays accounted for, we next explain how our analysis deals with NET exceedance.

B. Exceedance-Parametric Response-Time Analysis of Rössl

Consider an arbitrary job j of a given task under analysis $\tau \in \mathcal{M}$. We seek to establish an upper bound $R(e)$ on j's response

time that is parametric in the *total amount of exceedance* e "encountered" by j, which we make precise below.

Following Zini et al. [47], our analysis is based on a classic busy-window argument [9]. Busy windows are defined in terms of *quiet times*: an instant t is *quiet* with respect to job j iff all higher-or-equal-priority jobs that arrived before time t are complete at time t. Intuitively, quiet times are points without relevant backlog, allowing the analysis to begin from a point of zero backlog. The busy window of job j is then the maximal interval satisfying three conditions: 1) it contains j's arrival time, 2) it both starts and ends with quiet times, and 3) every other time within the interval is non-quiet. In other words, at every time in a busy window that is not one of the two endpoints, there exists some previously arrived, still incomplete job with priority at least that of j, which may be j itself.

Total exceedance. We can now define the total exceedance e precisely. Let $c(j')$ denote the *actual* execution time of a job j' (which, again, may be larger than its NET). With a slight abuse of notation, let $\tau(j')$ denote the task of job j'. Additionally, let J be the set of all jobs executing at any point in the busy window of job j. The total exceedance experienced by job j is then given by

$$e \triangleq \textstyle\sum_{j' \in J} \max\big(0,\, c(j') - C^{ovh}(\tau(j'))\big). \tag{1}$$

The first step of the analysis is to establish a condition that ensures that job j's busy window exists—otherwise, the system may be permanently overloaded and no response-time bound can be established. To this end, we require bounds on the blocking and competing workload encountered by j, which we define next. In the following, we use $hep(\tau) \triangleq \{\tau' \in \mathcal{M} \mid P^{ovh}(\tau') \leq P^{ovh}(\tau)\}$ to denote the set of tasks with priority at least that of τ, and $lp(\tau) \triangleq \{\tau' \in \mathcal{M} \mid P^{ovh}(\tau') > P^{ovh}(\tau)\}$ to denote the set of lower-priority tasks.

Blocking bound. Since Rössl is a nonpreemptive scheduler, tasks can suffer from *priority inversion*: specifically, when job j arrives, Rössl may currently be executing a lower-priority callback and not enter a polling phase until the currently executing callback completes. The resulting delay experienced by j is upper-bounded by the *nominal blocking bound* $B(\tau)$, which is obtained from the maximum inflated NET of any lower-priority task, and thus accounts for scheduling overheads:

$$B(\tau) \triangleq \begin{cases} \max_{\tau' \in lp(\tau)}(C^{ovh}(\tau') - \epsilon), & \text{if } lp(\tau) \neq \emptyset, \\ 0, & \text{otherwise.} \end{cases}$$

As is the case with NETs, the nominal blocking bound can be violated at runtime due to exceedance.

Competing workload. Job j can experience queueing delays if there are other pending jobs of higher or equal priority. Recall that, for each task $\tau' \in hep(\tau)$, at most $\alpha^{ovh}(\tau', \Delta)$ jobs of τ' arrive within any interval of length Δ, each of which has an inflated NET of $C^{ovh}(\tau')$. Consequently, the delay incurred by job j due to arrivals in an interval of length Δ is upper-bounded by the *nominal workload bound*

$$H(\tau, \Delta) \triangleq \textstyle\sum_{\tau' \in hep(\tau)} C^{ovh}(\tau') \cdot \alpha^{ovh}(\tau', \Delta).$$

Note that $H(\tau, \Delta)$ includes the NETs of jobs of task τ itself.

Busy window. With $B(\tau)$ and $H(\tau, \Delta)$ in place, we can finally state a sufficient condition for the existence of job j's busy window: if there exists a solution $L(e) > 0$ such that

$$L(e) \geq e + B(\tau) + H(\tau, L(e)), \tag{2}$$

then job j's busy window exists and is of length at most $L(e)$. This condition is necessarily parametric in e, as any exceedance that arises during the busy window may extend it.

To see why Eq. (2) is sufficient, consider any interval that starts with a quiet time (one always exists since time 0 is trivially quiet). Since the right-hand side bounds the total demand for processor time within an interval of length $L(e)$, an interval of length $L(e)$ can accommodate all such demand. Hence, no remaining backlog can exist, implying the existence of another quiet time and hence the end of the busy window.

Arrival offset. While job j's busy window includes j's arrival time by definition, the busy window does not necessarily start with the arrival of j. In the following, we let A denote the *arrival offset* (*i.e.*, j's arrival time relative to the start of its busy window). We first derive a response-time bound for a fixed (but unknown) arrival offset A, and then narrow down the search space of relevant offsets that must be considered.

For a fixed offset A, let $F_A(e)$ denote a bound on the time (relative to the start of the busy window) at which j is guaranteed to have started execution, where $F_A(e) > A$. Consider the work carried out in the busy window up to and including $F_A(e)$. There are four sources of processor demand: 1) ϵ amount of service required to start the execution of j, 2) any priority inversion (imposing a nominal demand of at most $B(\tau)$), 3) any *other* interfering higher-or-equal-priority jobs (imposing a nominal demand of at most $H(\tau, F_A(e)) - C^{ovh}(\tau)$), and 4) any exceedance (e). Regarding (3), $C^{ovh}(\tau)$ is subtracted because $H(\tau, F_A(e))$ includes j's own demand.

If the interval is sufficiently long to accommodate all such demand, then job j is guaranteed to have been dispatched. Job j's relative start time is thus bounded by the least value $F_A(e)$ satisfying both $F_A(e) > A$ and

$$F_A(e) \geq \epsilon + B(\tau) + H(\tau, F_A(e)) - C^{ovh}(\tau) + e.$$

Since Rössl is nonpreemptive, j runs to completion once it has started execution, nominally requiring at most $C^{ovh}(\tau) - \epsilon$ additional service to do so. The response time of job j, when it arrives at offset A and is affected by at most e total exceedance, is thus bounded by $R_A(e) \triangleq F_A(e) + C^{ovh}(\tau) - \epsilon - A$.

Response-time bound. While a job's relative arrival offset A is generally unknown *a priori*, it can be shown that the worst case is realized only for specific offsets, which yields a sparse, finite *search space* $SP(e) \triangleq \{A \mid A < L(e) \wedge \alpha^{ovh}(\tau, A) \neq \alpha^{ovh}(\tau, A + \epsilon)\}$. Intuitively, $SP(e)$ includes all relative arrival offsets up to the busy-window bound $L(e)$ at which the derived arrival curve $\alpha^{ovh}(\tau, \cdot)$ "steps." The response-time bound that we have verified to hold for Rössl is then given by

$$R(e) \triangleq \max_{A \in SP(e)} R_A(e). \tag{3}$$

Theorem 1. *For task $\tau \in \mathcal{M}$ and any job j of τ, if the total exceedance encountered by j in its busy window is at most e, and if a bound $L(e)$ satisfying Inequality (2) exists, then the response time of j is bounded by $R(e)$.*

Now, because NETs are measured and execution-time exceedance is unpredictable, we generally do not know the precise value of e ahead of time, nor a sound bound on the maximum possible e. However, instead of viewing e as a parameter that must be given or assumed, it is more beneficial to treat it as a modeling abstraction that can be solved for. Working backwards, this perspective yields a robustness guarantee.

Corollary 1. *Let D_i denote the* relative deadline *of callback $\chi_i \in \mathcal{CB}$, and let $\hat{e}_i \triangleq \max\{e \mid R(e) \leq D_i\}$, where $R(e)$ is evaluated for $\tau_i^{\chi} \in \mathcal{M}$. If no invocation of χ_i (i.e., no job of τ_i^{χ}) encounters more than $\hat{e}_i$ total exceedance in its busy window, then χ_i is* schedulable *(i.e., every invocation meets its deadline).*

This underscores the practical utility of our work: as we later explain in Sec. IV, Corollary 1 yields *interpretable, formally verified robustness guarantees* with respect to disturbances in nominal timing behavior, based only on measurements that are readily obtainable on commodity platforms.

The remainder of the paper explains how Theorem 1 is formalized and mechanically verified in an end-to-end proof covering the entire chain of reasoning, from Rössl's low-level source code all the way to the abstractions underlying Eq. (3).

III. Proving the Response-Time Bounds

To formally state and prove Theorem 1, we build on RefinedProsa's methodology for connecting C-level verification of scheduler implementations to mechanized response-time analyses. However, RefinedProsa's existing methodology does *not* account for execution-time exceedance. We begin here by revisiting RefinedProsa's proof structure before explaining how we adapt it to support exceedance analysis.

Stating that response times are bounded. Formally, RefinedProsa models a run of Rössl as a pair of traces: (1) a trace of arrivals (referred to as arr_seq) that records a list of messages sent to Rössl along with their arrival timestamps and (2) a Rössl execution trace that consists of a list of basic actions whose boundaries (called *marker functions* and marked in light blue in Fig. 1) are timestamped. The basic-action trace is referred to as $marker\_tr$ and the timestamps are referred to as $marker\_ts$. Fig. 3 illustrates such a pair of traces.

To formalize environmental properties, RefinedProsa assumes consistency conditions on the two traces. For example, if the marker trace says that Rössl reads a job at time t, then the arrival trace must make that job available no later than t.

Using these traces and their properties, we restate Theorem 1:

Theorem 1 (more formally). *Let $tr = \langle arr\_seq, marker\_tr, marker\_ts \rangle$ be a trace of a Rössl run observed up to a timing horizon t_H. For every task $\chi_i \in \mathcal{CB}$, assume that a solution L to inequality (2) exists. If the cumulative exceedance in a busy window of χ_i in tr is bounded by e, then every job of χ_i with an arrival time t_a in that window such that $t_a + R(e) < t_H$ has a response time of at most $R(e)$. Here, $R(e)$ is defined by Eq. (3) and the response time of a job is the distance between the job's arrival time from arr_seq and the timestamp of its completion marker function in $marker\_tr$.*

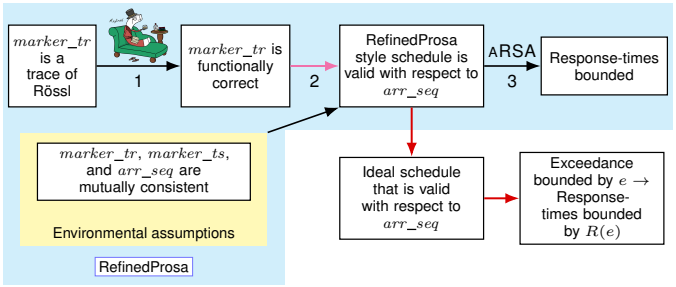

Fig. 2: Proof pipeline from concrete execution assumptions to exceedance analysis. The blue box shows the RefinedProsa part of the proof. The pink arrow shows parts of the proof that were slightly modified in our approach while the red arrows indicate completely new parts that our proof adds.

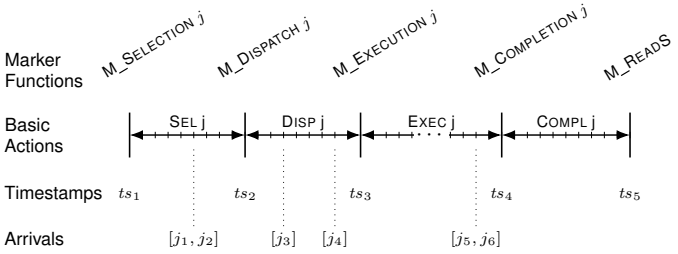

Fig. 3: An example trace of basic actions and corresponding marker functions and timestamps. The trace represents a part of Rössl execution in which job j is selected (SEL j), its callback dispatched (DISP j), the callback executes (EXEC j), after which control is returned to Rössl (COMPL j).

While the proof proceeds through several abstraction steps, the final theorem is stated directly over executions of the concrete implementation. Next, we give a short overview of how RefinedProsa proves this theorem.

RefinedProsa's proof outline. At a high level, RefinedProsa proves response-time bounds for Rössl through a three-stage proof structure shown in the blue box in Fig. 2. 1) A verification tool for *C* called RefinedC [37] is used to prove invariants on the execution traces of Rössl. 2) These execution traces along with the environment driving those executions are abstracted into a system model suitable for response-time analysis. 3) This system model is analyzed using a mechanized response-time analysis from the Prosa library [32].

RefinedProsa uses RefinedC to prove basic *functional* properties of marker-function traces generated by Rössl. These properties capture correct scheduler decisions: Rössl always selects the highest-priority pending job. They also include invariants on how often overheads can occur during Rössl 's execution, *e.g.*, a selection overhead occurs at most once per iteration of Rössl 's scheduling loop (Line 9 in Fig. 1).

Next, to simplify reasoning about response times, RefinedProsa merges consecutive basic actions into more abstract

classes of overhead and lifts the functional properties of the basic action traces to relevant properties of the more abstract trace. An example of a basic action trace and its corresponding abstract trace is shown in the 2nd and 3rd lines of Fig. 4. This abstract trace along with the trace of arrivals forms the system model that RefinedProsa analyzes to bound response times.

For the response-time analysis of this system model, RefinedProsa uses the ARSA framework from the Prosa library [8]. ARSA is a framework for proving response-time bounds for systems subject to processor blackouts. To apply ARSA, RefinedProsa models all the scheduler overheads as processor blackouts and uses the properties established for abstract traces to prove the assumptions required by ARSA, which, in addition to the usual priority policy compliance and work conservation assumptions, include bounds on the total blackout experienced by the system in any interval of a given length.

To prove these properties, RefinedProsa combines the verified properties of Rössl and the assumed environmental properties. On top of these properties, RefinedProsa uses some standard real-time modeling techniques—*e.g.*, release jitter is used to lift the correctness of scheduler decisions to the system model properties required by ARSA.

Unfortunately, these proofs rely extensively on local WCETs and thus do not transfer to the exceedance setting. A more detailed account of why RefinedProsa's techniques do not directly apply is given in the supplementary material [4]. Here, we instead focus on how our method accounts for the various complexities of Rössl and its timing behavior.

A. Analyzing Rössl

The concrete implementation of Rössl introduces several complications that must be accounted for in its response-time analysis. One important issue is that jobs do not become immediately visible to the scheduler when they arrive. Since Rössl polls for new arrivals only at specific points during execution, there may be a delay between a job physically arriving and the scheduler observing it. As a result, Rössl makes scheduling decisions based on the set of currently visible jobs, whereas the system model analyzed by exceedance analysis defines correctness of scheduling decisions with respect to the set of all jobs that have already arrived.

For example, consider the first row of Fig. 4, which shows a trace of basic actions produced by Rössl in a system with two tasks, τ_1 and τ_2, where τ_1 has higher priority than τ_2, and a single registered socket. Here, τ_2 arrives immediately after a read operation has started and is thus not observed during that polling phase. Consequently, the scheduler state remains empty, selection fails, and Rössl executes an idling phase.

In the next loop iteration, τ_2 is read and becomes visible to the scheduler. By the time the scheduler performs its next selection decision, the higher-priority task τ_1 has also already arrived. However, since τ_1 arrived too late to be read during that iteration, it is still absent from the scheduler state, and thus τ_2 is selected for execution first. Thus, although scheduler decisions are always correct w.r.t. the set of pending jobs in the state of the scheduler, they are not consistent w.r.t. the set of pending jobs in the environment.

In the presence of exceedance, further possibilities arise, as illustrated in the second row of Fig. 4. In this schedule, the basic actions executed during the idling phase exceed their NETs, delaying the next polling phase. This delay is large enough that both τ_2 and τ_1 are read before the next selection decision. Consequently, both jobs become simultaneously visible to the scheduler, and since τ_1 has higher priority, it is selected first—exactly the opposite of what happens without exceedance.

Crucially, the arrival pattern in both executions is identical; only the exceedance differs. Nevertheless, the effect on the response time of τ_2 is far larger than the exceedance itself. Rather than merely stretching delays, exceedance changes which jobs interfere with one another by altering which arrivals become simultaneously visible to the scheduler.

Our solution. At a high level, these issues arise from implementation-specific timing effects whose impact is further amplified by execution-time exceedance of both tasks and basic actions. Our solution adds two more steps on top of the RefinedProsa proof as shown outside the blue box in Fig. 2. First, we eliminate all implementation-specific timing effects (like the one described above) from the analysis model by abstracting RefinedProsa schedules into *ideal schedules* in which all execution is represented uniformly as workload of tasks in the augmented task set $\mathcal{M}$ from Sec. II. An example of such an abstracted schedule is shown in the bottom row of Fig. 4. Second, the resulting ideal schedule is analyzed using the mechanized exceedance analysis from Sec. III-B below.

Defining the abstraction. As explained in Sec. II, our abstraction to ideal schedules accounts for scheduler overheads in two ways. First, overheads immediately surrounding job execution are folded directly into the job's execution and accounted for through task cost inflation. Second, remaining scheduler overheads are modeled as jobs of pseudo tasks.

Specifically, scheduler overheads immediately preceding and following job execution are absorbed into the job itself. Thus, even if a higher-priority job arrives during the execution of these overheads but is not yet observed by the scheduler, this no longer causes a priority-policy violation in the abstract model. Next, iterations of the idling loop are modeled as jobs of the lowest-priority task τ^{idle}. Thus, even if a job arrives during the iteration of the idling loop but is not yet observed by the scheduler, this no longer violates work conservation in the abstract model. Finally, all read overheads are modeled as jobs of τ^{read}. As a result, delays caused by polling and reading are represented simply as ordinary higher-priority interfering workload. This is significantly simpler than RefinedProsa's handling of delays caused by reading overheads, which involves a separate fixed-point equation for bounding the maximum number of jobs read in one polling phase.

However, defining the abstraction algorithm alone is not sufficient: as explained in Sec. II, a major part of our proof is showing that this abstraction correctly accounts for all scheduler overheads and that the resulting ideal schedules

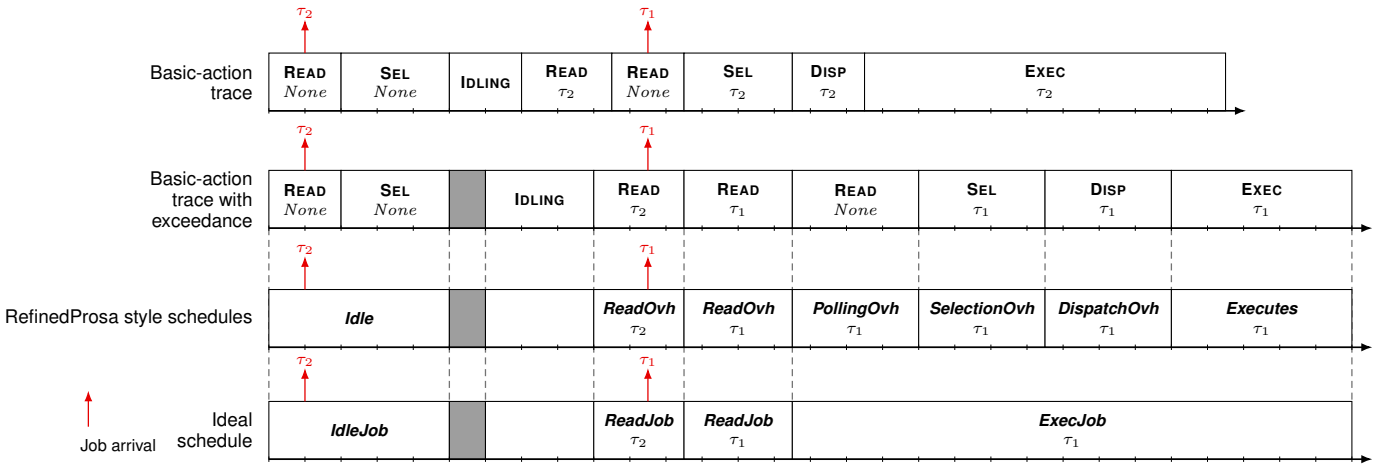

Fig. 4: Trace of basic actions with and without exceedance. The figure also shows the corresponding RefinedProsa abstraction of the trace and our new abstraction.

satisfy the assumptions required by response-time analysis, namely priority-policy compliance and work conservation.

These properties are proved by combining: (i) invariants proved about basic-action traces during the C-level verification of Rössl, (ii) properties of the abstraction algorithm itself, and (iii) the parameters of $\mathcal{M}$ defined in Sec. II.

For example, one of the invariants we establish on Rössl traces is that a selection overhead occurs at most once per scheduling-loop iteration. Using this, we show that our abstraction algorithm can attribute every selection overhead either to the execution of the selected job or to the idling loop iteration that follows.

Another example is the proof of priority-policy compliance. Recall from Sec. II that τ^{read} is assigned the highest priority. Combined with the verified properties that reading overheads always occur before job execution and that polling stops only when no further messages remain to be read, this allows us to prove priority-policy compliance for reading overheads. The remaining proof obligations are established similarly.

The abstraction to ideal schedules does not *per se* eliminate reorderings of job execution caused by execution-time exceedance. If a job or scheduler action executes longer than expected, newly arriving higher-priority jobs may still become visible and execute first. However, after the abstraction, such reorderings arise purely from NET exceedances and are no longer tied to any system-specific mechanisms such as polling or scheduler visibility. Consequently, the reordering issue becomes precisely the problem targeted by exceedance analysis.

B. Mechanizing Exceedance Analysis

Prior exceedance-analysis results [47] were developed using pen-and-paper proofs over idealized system models. In contrast, RefinedProsa derives guarantees from machine-checked proofs. To adapt RefinedProsa to exceedance analysis, we thus need a way to mechanize the analysis from Sec. II-B and connect it to the properties of ideal schedules established by RefinedProsa.

Our key insight is to adopt a perspective shift whereby exceedance is modeled as a form of *blackout*, *i.e.* as an interval of restricted processor supply. This perspective shift is useful

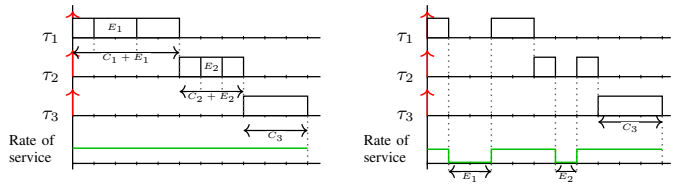

Fig. 5: Schedule with overruns and schedule with blackout. Red arrows represent task arrivals.

because it enables us to take ARSA—an abstract response-time analysis framework for systems with blackout intervals—and repurpose it for mechanizing exceedance analysis.

Concretely, we interpret every task exhibiting execution-time overrun as a task suffering from a lack of processor supply. While fundamentally there is a difference between an increase in demand and a decrease in supply, from the perspective of schedulability analysis, both phenomena have the same effect: they introduce the same additional delay into the schedule and hence give rise to the same cascading scheduling effects. Thus, for the purpose of response-time analysis, an execution-time overrun is indistinguishable from a blackout period during which the processor is unavailable.

The idea behind this model is illustrated in Fig. 5. In the schedule on the left, the tasks τ_1, τ_2, and τ_3 are all released at time 0. Since τ_1 has higher priority, it executes first. τ_1 has a nominal execution time of C_1 but runs for $C_1 + e_1$. Therefore, out of the total execution time, e_1 units in the middle of the execution are modeled as a blackout. Next, task τ_2, which has nominal execution time C_2, runs for $C_2 + e_2$. Finally, task τ_3 runs for time C_3. The response time of τ_3 is thus $C_1 + e_1 + C_2 + e_2 + C_3$. Now, in the schedule on the right, the periods of overrun are instead interpreted as periods of blackout. However, in both cases, the effect on task τ_3 is the same: it is delayed by an additional $e_1 + e_2$ units beyond what is accounted for by the nominal execution times.

This modeling reduces a schedule with exceedance to a schedule with restricted processor supply; such a schedule is in turn amenable to analysis using ARSA, which provides response-time bounds for all tasks when standard scheduling

assumptions hold and the total blackout is bounded. To apply aRSA, one must show two things: first, that standard scheduling properties such as priority compliance and work conservation hold; and second, that the total blackout within the busy interval of any job is bounded. The crucial point is that in our setting this second obligation lines up exactly with exceedance analysis: the total blackout in a busy interval is precisely the cumulative exceedance in that interval. Once these conditions are established, aRSA yields response-time bounds together with machine-checked correctness guarantees.

We thus obtain a fully mechanized version of exceedance analysis. Combining this mechanized exceedance analysis with the ideal schedules defined above, we prove Theorem 1.

C. Threats to Validity

An important aspect of Theorem 1 is understanding precisely what is verified and what assumptions remain. The *trusted computing base* (TCB) on which the theorem depends is summarized in Table II.

IV. Case Study

We conducted a case study exploring how to deploy Rössl on a commodity platform such that the execution environment matches the verified model (Sec. IV-A), obtain all parameters necessary for analysis (Sec. IV-B), and derive concrete robustness metrics (Sec. IV-C). Additionally, we ran controlled experiments to compare observed response times with those predicted by Theorem 1 (Secs. IV-D and IV-E).

The workload. We configured Rössl to schedule seven callbacks, each performing a matrix multiplication of a distinct size. We chose matrix multiplication as a representative workload because it captures the compute-heavy kernels common in signal processing and ML inference, and in some control applications. The matrix dimensions ranged from 120×120 to 168×168 in increments of 8. The corresponding callback tasks are named `MATRIX_120` through `MATRIX_168`. Callbacks processing smaller matrices were assigned higher priorities. For simplicity, all callbacks were triggered periodically via UDP messages, with periods as indicated in Table III.

A. Linux Setup for Rössl

A key practical advantage of our approach is that deploying Rössl does not require specialized hardware or OS support. Accordingly, we performed all experiments on a COTS 64-bit ARM platform (a Raspberry Pi 5) running a mainline Linux kernel (version 6.12, with the PREEMPT_RT option enabled).

Our proof models Rössl as running (exclusively) on a uniprocessor. Accordingly, we configured Linux to fully dedicate one core of the Raspberry Pi 5 to Rössl. Specifically, we isolated a core from regular OS use by booting Linux with the `isolcpus=nohz,domain,managed_irq` command-line parameter. The `nohz` option suppresses periodic scheduler ticks, the `domain` flag removes the core from all load-balancing domains (*i.e.*, it stops Linux from moving other processes to the isolated core), and `managed_irq` prevents interrupts from being routed to the isolated core, to the extent possible on the hardware platform.

This configuration significantly reduces OS noise and limits kernel-induced interference, but it does not completely isolate the core. For example, we found that certain interrupts were still delivered to the core reserved for Rössl. More importantly, hardware interference channels such as shared caches and other elements of the Raspberry Pi 5's memory subsystem remain unmitigated (when running a current mainline Linux kernel).

This imperfect isolation is not problematic; rather, it highlights a strength of our NET-based, exceedance-aware approach: Rössl does *not* require a perfectly isolated core for Theorem 1 to hold. Since our model is capable of capturing small, transient disruptions—such as the occasional interrupt, a spike in LLC misses, or increased contention for memory bandwidth—as execution-time exceedance, small model-fidelity mismatches can simply be treated as cases of temporal disturbance. As we show below, Corollary 1 allows quantifying the degree to which such disturbances can be tolerated without violating any deadlines.

In addition to configuring Linux to reserve a core for Rössl, we used the marker functions in Rössl's source code to inject *compiler optimization barriers* (via inline assembly blocks). Such barriers prevent the compiler from reordering instructions across basic action boundaries. Since basic actions form the atoms of our timing model, the optimization barriers ensure that our measurement setup captures the appropriate NETs required by our analysis, as discussed next.

B. Parameter Estimation

To apply Corollary 1 (and hence Theorem 1), we must first obtain the NETs and arrival curves for all callbacks, as well as the NETs for all basic actions listed in Table I. To this end, we compiled Rössl in an instrumented *profiling mode* and drove it with a representative workload to measure all required parameters under nominal operating conditions.

Arrival curves. As Rössl receives all its input via sockets, message arrivals can be observed via Linux's standard packet capture (*pcap*) interface, which exposes both packet timing and payloads (from which message IDs can be inferred). Given a trace of packet arrival times and message IDs, arrival curves for all callbacks can be readily extracted using standard methods (*e.g.*, see the detailed overview by Stark [39]). Ye et al. [44] provide Rust and Python libraries for this purpose.

NETs. To measure the NETs of callbacks and basic actions, we instrumented Rössl to record timestamps obtained from the *ARM system counter*, a globally visible, monotonic 54 MHz clock shared across all cores that can be read with minimal overhead. More specifically, recall that marker functions at designated points in Rössl's C implementation indicate the start and end of basic actions, including the execution of callbacks. When Rössl is compiled in profiling mode, each invocation of a marker function logs a fixed-size event comprising the marker ID, the corresponding job ID (if any), and a timestamp. Each such event is written to a preallocated single-producer, single-consumer ring buffer, which a separate collector thread asynchronously drains to a log file. Tracing in Rössl's main loop

TABLE II: Trusted components and assumptions underlying the end-to-end theorem.

Trusted component	Assumption	Role in the theorem
Rocq kernel	The proof-checking kernel of Rocq is sound.	Our proofs are machine-checked by the Rocq kernel.
RefinedC semantics of C	The formal C semantics used by RefinedC faithfully models the actual semantics of C.	The implementation-level correctness theorem for Rössl is proved with respect to these semantics.
Operating-system calls	The operating system faithfully implements the POSIX behavior of the `read` system call used by Rössl.	If a message is visible to Rössl, `read` is assumed to fully and correctly read it from the sockets.
Marker placement	Execution eventually progresses from each marker function to the next; in particular, execution does not diverge between consecutive markers.	This ensures that marker function traces accurately account for executions of Rössl.
Arrival trace	The concrete arrival trace arr_seq faithfully represents the external arrivals observed by Rössl.	This ensures that the environment is correctly captured by the arr_seq.
Arrival curves	The observed environment respects the specified task arrival curves.	This is a concrete analogue of the standard arrival-curve assumption used by schedulability analysis.

(Fig. 1) is thus limited to lock-free constant-time operations and performs no system calls, thereby minimizing probe effects.

From the recorded log file, we extract the observed execution times using the start and end timestamps of all basic actions and callback executions. As these observations can be affected by OS noise (*e.g.*, spurious interrupts), we apply a standard *Hampel* outlier filter to exclude likely anomalous observations. For each task and basic action, we then define the corresponding NETs as the maximum of the filtered observations. This highlights a key practical advantage of our exceedance-based approach: because it requires only NETs rather than strict WCETs, we can apply standard statistical outlier filtering without having to perfectly separate valid observations from noise. Any discarded extremes and additional overruns are accounted for explicitly as execution-time exceedance in the subsequent analysis.

Ground-truth exceedance. The instrumented setup described so far suffices to use Rössl and to derive all necessary inputs for Theorem 1 and Corollary 1 in practice. For our experiments comparing our analysis with observed ground-truth exceedance amounts (Secs. IV-D and IV-E), we also required more detailed traces that include precise message arrival times on the same timeline as the marker-function log.

For this purpose, we additionally instrumented the kernel's UDP receive path using eBPF to observe packets at the enqueue point without modifying the kernel. Since the ARM system counter is not directly accessible from eBPF, we loaded a small kernel module to expose it to eBPF programs via a custom API. Events captured in kernel space were forwarded to user space via Linux's *perf* interface and appended to a log file. To reiterate, this instrumentation of the kernel was required solely to enable our validation experiments discussed below and is not needed for regular use of Rössl and the supporting analysis.

C. Applications of Exceedance Analysis

Profiling our matrix-multiplication workload as discussed above resulted in the NETs reported in Table III. (The NETs of basic actions have been omitted due to space constraints.)

We imposed *implicit deadlines*: every callback invocation was required to complete before the next invocation of the same callback. Applying Theorem 1 with $e = 0$ verified that the system is *schedulable under nominal conditions*: in the absence of execution-time exceedance, every callback invocation is guaranteed to meet its deadline. The natural next question for a system designer to ask is then: "How much margin does the system have before deadline misses become possible?"

Tolerable exceedance. Our analysis can provide the desired insights. More specifically, applying Corollary 1 to each callback allowed us to determine the callback-specific *tolerable exceedance threshold* (TET) $\hat{e}$, which is the maximum amount of exceedance a callback invocation can tolerate in its busy window while still guaranteeing the absence of deadline misses. The TET $\hat{e}$ is hence a direct measure of the system's robustness against transient temporal disturbances.

The results, reported in Table III, show that higher-priority callbacks tend to have larger TETs, as one might intuitively expect. However, the specific values are not easily anticipated "at a glance" from workload parameters alone, and the results are also not monotonic in callback priority, which highlights the value of rigorously computing TETs via Corollary 1.

Recovery time. Another important question is: "How long does tolerable exceedance affect the system?" In other words, if the system experiences execution-time exceedance that remains below a callback's TET, for how long can the resulting transient effects linger? Conversely, after what duration is TET-bounded exceedance guaranteed to have "dissipated"?

Our analysis can answer this question by computing the busy-window bound $L(e)$ for the lowest-priority callback `MATRIX_168`, with e set to the TET of the callback under analysis. For example, to bound the maximum *recovery time* for the highest-priority callback `MATRIX_120`, we compute the value $L(e)$ for `MATRIX_168` with $e = 97.85\,\text{ms}$, the TET of `MATRIX_120`. The solution, $352.40\,\text{ms}$ in the case of `MATRIX_120`, bounds the duration after which a busy window of the lowest-priority callback is guaranteed to have ended, at which point the system as a whole becomes quiet (*i.e.*, there is no backlog remaining, at any priority level).

Necessary slowdown. The recovery time is an easy-to-understand metric that is intuitively meaningful to system designers. In comparison, the absolute TET value may be less immediately informative. We therefore derive a second, more directly interpretable perspective on Corollary 1.

TABLE III: The evaluated matrix multiplication workload.

Callback (χ)	Period (ms)	*NetCB* (ms)	TET ($\hat{e}$) (ms)	Slowdown ($\hat{s}$)	Recovery (ms)
120	125	7.27	97.85	384.10%	352.40
128	100	10.47	62.39	229.73%	198.59
136	125	10.55	66.36	186.76%	295.55
144	200	12.56	110.95	198.97%	365.51
152	100	14.88	24.39	120.53%	160.59
160	200	17.38	63.80	130.48%	293.00
168	200	19.83	63.80	127.84%	292.99

Given a job j, recall from Eq. (1) that $\max\big(0, c(j) - C^{ovh}(\tau(j))\big)$ is the amount of exceedance exhibited by j, where $c(j)$ is its actual execution time and $\tau(j)$ is j's corresponding task. The job's individual *slowdown factor*

$$sl(j) \triangleq 1 + \frac{\max\big(0, c(j) - C^{ovh}(\tau(j))\big)}{C^{ovh}(\tau(j))}$$

thus expresses the increase in processor demand in relative terms (*e.g.*, a slowdown factor of 200% means that the job exhibits twice its nominal cost). From Eq. (1), we can trivially restate the total exceedance in terms of slowdown factors:

$$e = \sum_{j \in J} (sl(j) - 1) \cdot C^{ovh}(\tau(j)). \tag{4}$$

There are many ways to distribute the total amount of exceedance among the jobs in J, some more extreme than others. We focus on the *maximum slowdown* $\hat{s}$; formally, let $\hat{s}$ be the least value such that $sl(j) \leq \hat{s}$ for every job $j \in J$.

Now, what is the *least* $\hat{s}$ that allows the total exceedance to reach a callback's TET? From Eq. (4) and $\hat{s}$, we have:

$$\hat{e} = \sum_{j \in J} (sl(j) - 1) \cdot C^{ovh}(\tau(j)) \leq (\hat{s} - 1) \cdot \sum_{j \in J} C^{ovh}(\tau(j)).$$

Solving for $\hat{s}$, we obtain $\hat{s} \geq 1 + \big(\hat{e} / \sum_{j \in J} C^{ovh}(\tau(j))\big)$, where, J contains all higher-or-equal-priority jobs permitted by the arrival curves during a complete busy window of length $L_i(\hat{e}_i)$, plus one potential blocker: the lower-priority callback with the largest inflated NET, or the idling task if its NET is larger. The necessary slowdown $\hat{s}$ gives us an immediately interpretable boundary condition for system robustness: as long as no job exhibits a slowdown greater than $\hat{s}$, the callback under analysis is guaranteed to meet its deadline.

Table III shows that the necessary slowdown metric generally increases with callback priority, as is expected under fixed-priority scheduling. For example, the callback `MATRIX_128` is not at risk of missing a deadline unless some job executes for more than 2.2 times its nominal cost, even if jobs arrive at the maximum rate permitted by the arrival curves. In contrast, the schedulability of the lowest-priority callback `MATRIX_168` is guaranteed only while every job's cost increase remains below $\approx 27.8\%$. Overall, the necessary slowdown metric provides an easy-to-check, directly interpretable assessment of a callback's temporal safety margin that is intuitively meaningful.

D. Evaluating the Tightness of the Analysis

To assess the tightness of our analysis, we added a synthetic spin loop to the callbacks that allowed us to artificially inject arbitrary amounts of exceedance in a controlled manner. We

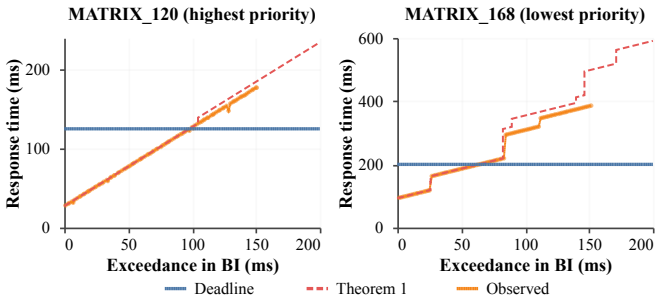

Fig. 6: Maximum observed and predicted response times.

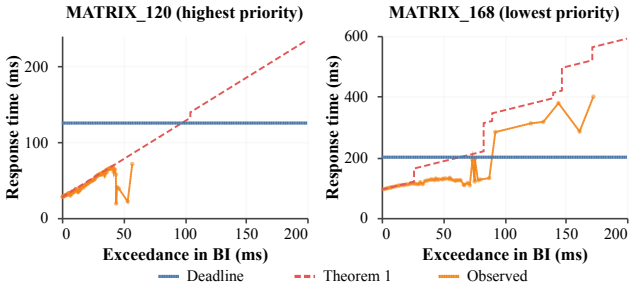

Fig. 7: Observed maximum response times for `MATRIX_120` (left) and `MATRIX_168` (right) under memory stress.

varied the amount of total exceedance e from 0 to 150 ms, which we injected into selected callback invocations such that the resulting schedule closely approximated the worst-case scenario analyzed in Sec. II-B. Fig. 6 shows the results for the highest- and lowest-priority callbacks.

The two plots, which compare the *observed* maximum response times with the *predicted* bound from Theorem 1 as the amount of total exceedance e varies, reveal several key observations. First, as expected given that the analysis has been verified, the graphs confirm that the derived bounds are sound, as all observed maxima remain within the predicted limits.

Second, Fig. 6 reveals that there is reassuringly little pessimism in our analysis: the observed response times largely track the predicted bounds at all priority levels.

Third, a small gap appears nonetheless for larger exceedance amounts in both subplots of Fig. 6, where the observed maxima do not quite reach the predicted bounds. We traced this discrepancy to the modeling of *self-interference*. Since Rössl cannot guarantee that all callbacks will be executed in the order in which messages are sent (packets may be reordered in the network or read out of order from different sockets), the analysis must account for the possibility that a callback invocation is delayed by conceptually later but reordered invocations of the same callback. As exceedance increases, the theoretical busy window grows large enough to admit multiple arrivals of the same task, all of which are conservatively treated as possibly interfering, thereby inflating the response-time bound. In practice, however, this reordering window is very small, and such reordering did not manifest in our experiments. Modeling the race window for message reordering explicitly could enable a (slightly) tighter bound, which we leave to future work.

E. Execution-Time Exceedance Due to Memory Stress

A common concern in practice is that substantial execution-time exceedance may arise from cross-core interference via shared hardware components in the platform's memory hierarchy (such as caches and memory bank controllers). To explore the effects of such memory interference, we repeated the prior experiment comparing observed and predicted maximum response times, but this time using the well-known `stress-ng` load-testing tool to induce contention in the memory subsystem. That is, instead of injecting exceedance artificially at controlled times, we let `stress-ng` induce exceedance in a largely uncontrolled manner as a byproduct of cross-core interference.

To emulate transient exceedance, `stress-ng` was activated according to a *Poisson* process with a mean inter-arrival time of 20 seconds. At each activation of the stressor, its runtime was chosen uniformly at random from 10 to 20 seconds.

The results are shown in Fig. 7. The major differences compared to Fig. 6 are due to two key effects. First, since exceedance arises "naturally" in this experiment, we cannot report samples across the full range: smaller amounts of exceedance are much more likely to manifest than larger amounts, thereby limiting the range of practically observable values of e to some workload- and hardware-specific range.

Second, in Fig. 7, the observed response times do not come as close to reaching the predicted maxima as in Fig. 6, especially for the highest-priority callback in the left panel of Fig. 7. The reason is that memory stress tends to manifest more gradually throughout busy windows, affecting all callback invocations and basic actions more or less equally (rather than as one extreme delay at a critical moment that maximizes its impact, as in our previous experiment). In other words, memory stress does lead to execution-time exceedance, but it does not necessarily lead to *worst-case* execution-time exceedance.

The conclusion from this experiment is twofold. On the one hand, the proposed analysis is *provably and demonstrably sound for worst-case exceedance patterns*, and thus safely over-approximates *all* sources of execution-time exceedance, no matter how they manifest. On the other hand, in scenarios in which *only* cross-core memory interference is a concern, more refined models that can capture the fact that exceedance due to memory stress arises gradually over time could yield considerably tighter bounds, which indicates an interesting future direction for NET-based reasoning.

F. Multiple Exceedance Sources and Multicore Deployment

A strength of the general, unrestricted model of exceedance underlying our analysis is that it is indifferent to the source of exceedance: any NET overrun is treated alike, irrespective of its cause. In particular, it transparently supports multiple disturbance sources affecting jobs simultaneously. To exhibit this capability, as well as two other features of our analysis—support for arbitrary arrival processes and multicore deployments—we extended the experimental setup from Sec. IV-E as follows.

In addition to the seven-task workload considered before, which was pinned to core 2, we added a second, independent Rössl instance on core 3. This second instance ran a slightly

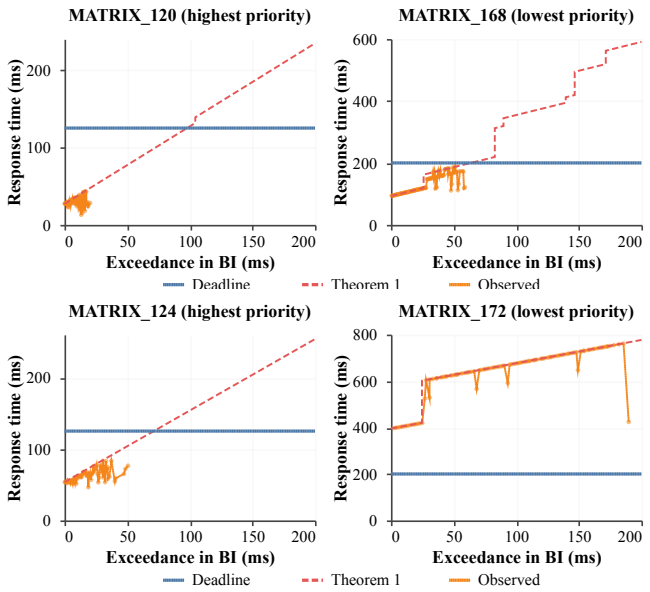

Fig. 8: Observed maximum response times in the multicore deployment. The top row shows the highest- and lowest-priority periodic callbacks on core 2; the bottom row shows the highest- and lowest-priority bursty callbacks on core 3.

modified workload also comprising seven matrix-multiplication tasks (with different matrix sizes). Importantly, the new tasks on core 3 were triggered to release jobs in *bursts* of four, with inter-burst separations of 400, 500, or 800 ms chosen such that the system is not permanently overloaded. These bursty activations highlight that our analysis supports arbitrary arrival processes, including the simultaneous arrival of multiple jobs.

As before, we first performed a benchmark run to collect NETs, and then added transient memory stress with `stress-ng`, which affected both cores alike. Additionally, we exposed core 3 to *interrupt storms* as a second source of exceedance with a custom kernel module that sent inter-processor interrupts (IPIs) from core 0 to core 3. Upon receiving each IPI, core 3 executed a 35 μs spin loop, thereby delaying the local Rössl instance.

Interrupt storms were exponentially distributed with a mean separation of 10 seconds. Each storm lasted for a duration chosen uniformly at random between 0.5 and 1 seconds, and generated IPIs at a rate of 6,000 interrupts per second. Since the IPIs targeted only core 3, they created a localized source of exceedance in addition to the memory interference experienced by both schedulers. Tasks on core 3 thus exhibited NET exceedance due to the combined effects of multiple disturbance sources. To avoid permanent overload, we reduced the per-activation runtime of `stress-ng` to 2–4 seconds.

Figure 8 compares the maximum observed response times with the predicted bound for the highest- and lowest-priority tasks on each core. In all four plots, the observed maxima remain below the bound throughout the analyzed range. The measurements on core 3 span a substantially larger range of observed exceedance than those on core 2, as expected due to the interrupt storms directed exclusively at core 3.

The experiment demonstrates that Rössl supports *partitioned*

multicore deployments, where multiple independent scheduler instances execute concurrently on different cores. Each instance is analyzed separately, while both global interference through shared resources and core-local disturbances are jointly and uniformly accounted for as general NET exceedance.

V. Related Work

This work lies at the intersection of two research directions: the formal verification of real-time system implementations and schedulability analysis without trusted WCET assumptions.

Formal verification. Given the critical nature of real-time systems, formal verification is being increasingly adopted by the community. Existing approaches range from model checking [14, 16, 31] to interactive theorem proving [2, 7, 9, 11, 17, 18, 26, 41], and target either schedulability theory and response-time analysis [2, 9, 11], scheduler implementations [3, 17, 18, 41], or operating-system kernels [13, 21]. While verification of schedulability analyses eliminates errors in the underlying theory, implementation mismatches may still lead to runtime bugs [40], motivating approaches that verify the concrete implementation itself. Among such approaches, some build on verified operating systems while others, including our work and RefinedProsa [3], trust the underlying OS to improve applicability. The underlying verification techniques also differ significantly in their trusted computing base. For example, approaches based on SPIN [14], VCC/Z3 [16], or UPPAAL [31] ultimately trust the corresponding verification engine, whereas interactive theorem proving provides stronger guarantees since correctness depends only on a comparatively small trusted proof kernel such as those of Rocq or Isabelle.

Schedulability analysis without WCETs. Classical schedulability analyses [20, 24] traditionally rely on strict WCET assumptions. More recently, however, significant effort has gone into relaxing or removing this assumption.

One line of work studies probabilistic real-time systems [15], where execution times are modeled probabilistically rather than through strict WCET bounds [10, 28, 43]. While such approaches can provide strong probabilistic guarantees, they also introduce substantial conceptual and technical challenges [15].

Another related direction is sensitivity analysis [5, 6, 12, 22, 23, 33–35, 45, 46], which studies how WCETs may be scaled while preserving schedulability. Unlike exceedance analysis, however, sensitivity analysis typically considers permanent overload and analyzes the effect of scaling individual tasks rather than cumulative transient exceedance across the task set.

Exceedance analysis [47] approaches the problem from a different perspective by characterizing how response times degrade as execution exceeds nominal runtime estimates. While this does not provide probabilistic guarantees as strong as those obtainable from fully stochastic analyses, it yields a comparatively simple characterization of robustness to execution-time overruns [29]. However, existing exceedance-analysis results are limited to pen-and-paper proofs and do not reason about concrete implementations. Our work strengthens this line of work by mechanizing exceedance-aware response-time analysis and connecting it directly to a verified scheduler implementation.

VI. Conclusion, Discussion, and Future Work

We have formally verified exceedance-aware response-time bounds for a concrete system by connecting implementation-level execution traces to mechanized exceedance analysis through multiple abstraction layers. Our work also strengthens prior exceedance theory by mechanizing the analysis itself. Evaluation with Linux on a Raspberry Pi platform shows that the approach is practical on commodity hardware.

Exceedance analysis provides meaningful robustness guarantees under transient execution-time overruns. Crucially, our intention is *not* for users to first determine a single exceedance bound and then ensure that all busy windows comply. Rather, given a workload's timing requirements, our approach yields useful robustness metrics like the TET (Sec. IV-C), which indicates how much exceedance a system can bear before missing deadlines. Provided the TET is deemed "sufficiently high"—which is a judgement call taking into account an application's specific context, requirements, economic constraints, and applicable engineering experience—the system can be considered safe to deploy. In other words, exceedance analysis characterizes a system's *safety margin* against *all* temporal disturbances that manifest as NET overruns. How large a safety margin is desirable (or can be afforded) remains an engineering decision that must be made on a case-by-case basis.

Orthogonally, as the actual exceedance a system faces is discovered only at runtime, online monitoring and mitigation techniques naturally lend themselves as complementary deployment mechanisms. Specifically, metrics provided by a design-time exceedance analysis can inform the configuration of runtime mechanisms. For example, if a system has a fail-operational "degraded mode," the TET indicates when it may become necessary to trigger said mode. We leave the integration of exceedance analysis with runtime mitigations to future work.

Obtaining verified exceedance analyses for more systems besides Rössl would also be a worthwhile future direction. We believe our methodology is broadly applicable and thus, in principle, expect it to transfer well to other schedulers. However, an end-to-end proof is necessarily specific to a concrete system (such as Rössl). Applying the methodology to another scheduler hence first requires verifying its implementation.

There are also several promising avenues for further analytical improvements and extensions. While our work already applies to partitioned multiprocessor systems (with a separate Rössl instance per core), the challenge of supporting *global* multiprocessor systems (where callbacks are dispatched dynamically across multiple cores by a single instance), or analogously, *multithreaded executors*, remains wide open. Additionally, the literature on exceedance analysis has to date dealt only with uncertainty in execution-time bounds, but if arrival curves are measured rather than analytically derived, then those may be uncertain, too. It would thus be desirable to obtain a theory of "arrival-bound exceedance" that combines well with existing results on NET exceedance. Last but not least, as already noted in Sec. IV-E, there is an opportunity for disturbance-specific exceedance models that trade generality for tightness.

References

[1] "The Rocq Prover," https://rocq-prover.org.

[2] K. Bedarkar, M. Vardishvili, S. Bozhko, M. Maida, and B. B. Brandenburg, "From intuition to Coq: A case study in verified response-time analysis of FIFO scheduling," in *Proceedings of the 43rd IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 197–210.

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

[4] K. Bedarkar, B. Ye, D. Garg, D. Dreyer, and B. B. Brandenburg, "Supplementary material for 'From estimates to guarantees: Verified exceedance analysis on commodity hardware'," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.22795171

[5] E. Bini, M. Di Natale, and G. C. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," *Real-Time Systems*, vol. 39, no. 1–3, pp. 5–30, 2008.

[6] L. Bougueroua, L. George, and S. Midonnet, "Temporal robustness of real-time architectures specified by estimated WCETs," *International Journal on Advances in Software*, pp. 359–371, 2009.

[7] M. Boyer, P. Roux, H. Daigmorte, and D. Puechmaille, "A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus," in *Proceedings of the 33rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2021, pp. 14:1–14:21.

[8] S. Bozhko, "Rigorous and general response-time analysis for uniprocessor real-time systems," Ph.D. dissertation, Saarland University, Germany, 2026, in preparation.

[9] 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.

[10] L. Carnevali, L. Santinelli, and G. Lipari, "Towards probabilistic modeling and analysis of real-time systems," in *Proceedings of the 15th European Performance Engineering Workshop (EPEW)*, 2018, pp. 157–172.

[11] F. Cerqueira, F. Stutz, and B. B. Brandenburg, "PROSA: A case for readable mechanized schedulability analysis," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 273–284.

[12] J. Chen, C. Du, P. Han, and Y. Zhang, "Sensitivity analysis of strictly periodic tasks in multi-core real-time systems," *IEEE Access*, vol. 7, pp. 135 005–135 022, 2019.

[13] S. Cheng, J. Woodcock, and D. D'Souza, "Using formal reasoning on a model of tasks for FreeRTOS," *Formal Aspects of Computing*, vol. 27, no. 1, pp. 167–192, 2015.

[14] D. D. Cofer and M. Rangarajan, "Formal verification of overhead accounting in an avionics RTOS," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002, pp. 181–190.

[15] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for real-time systems," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, pp. 04:1–04:53, 2019.

[16] S. Divakaran, D. D'Souza, A. Kushwah, P. Sampath, N. Sridhar, and J. Woodcock, "Refinement-based verification of the FreeRTOS scheduler in VCC," in *Proceedings of the 17th International Conference on Formal Engineering Methods (ICFEM)*, 2015, pp. 170–186.

[17] X. Guo, M. Lesourd, M. Liu, L. Rieg, and Z. Shao, "Integrating formal schedulability analysis into a verified OS kernel," in *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, 2019, pp. 496–514.

[18] X. Guo, L. Rieg, and P. Torrini, "A generic approach for the certified schedulability analysis of software systems," in *Proceedings of the 27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021, pp. 83–92.

[19] S. Hahn, M. Jacobs, N. Hölscher, K.-H. Chen, J.-J. Chen, and J. Reineke, "LLVMTA: An LLVM-based WCET analysis tool," in *Proceedings of the 20th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2022, pp. 2:1–2:17.

[20] P. K. Harter, Jr., "Response times in level-structured systems," *ACM Transactions on Computer Systems*, vol. 5, no. 3, pp. 232–248, 1987.

[21] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[22] P. Kumar and L. Thiele, "Quantifying the effect of rare timing events with settling-time and overshoot," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, 2012, pp. 149–160.

[23] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS)*, 1989, pp. 166–171.

[24] 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.

[25] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[26] M. Maida, S. Bozhko, and B. B. Brandenburg, "Foundational response-time analysis as explainable evidence of timeliness," in *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS)*, 2022, pp. 19:1–19:25.

[27] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys*, vol. 52, no. 3, pp. 56:1–56:38, 2019.

[28] F. Marković, J. Carlson, R. Dobrin, B. Lisper, and A. Thekkilakattil, "Probabilistic response time analysis for fixed preemption point selection," in *Proceedings of the 13th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2018, pp. 1–10.

[29] F. Marković, G. von der Brüggen, M. Günzel, J. Chen, and B. B. Brandenburg, "A distribution-agnostic and correlation-aware analysis of periodic tasks," in *Proceedings of the 45th IEEE Real-Time Systems Symposium (RTSS)*, 2024, pp. 215–228.

[30] E. J. Maroun, E. Dengler, C. Dietrich, S. Hepp, H. Herzog, B. Huber, J. Knoop, D. Wiltsche-Prokesch, P. Puschner, P. Raffeck, M. Schoeberl, S. Schuster, and P. Wägemann, "The Platin multi-target worst-case analysis tool," in *Proceedings of the 22nd International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2024, pp. 2:1–2:14.

[31] M. Mikucionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard, "Schedulability analysis using Uppaal: Herschel-Planck case study," in *Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2010, pp. 175–190.

[32] "PROSA: A foundation for formally proven schedulability analysis," http://prosa.mpi-sws.org/, 2026.

[33] S. Punnekkat, R. I. Davis, and A. Burns, "Sensitivity analysis of real-time task sets," in *Proceedings of the Third Asian Computing Science Conference (ASIAN)*, R. K. Shyamasundar and K. Ueda, Eds., 1997, pp. 72–82.

[34] R. Racu, A. Hamann, and R. Ernst, "A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, 2006, pp. 3–12.

[35] ——, "Sensitivity analysis of complex embedded real-time systems," *Real-Time Systems*, vol. 39, no. 1–3, pp. 31–72, 2008.
[36] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 34:1–34:25, 2012.
[37] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "RefinedC: Automating the foundational verification of C code with refined ownership types," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021, pp. 158–174.
[38] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 2–13.
[39] T. Stark, "Real-time execution management in the ROS 2 framework," Ph.D. dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Saarbrücken, 2022.
[40] H. Teper, D. Kuhse, M. Günzel, G. von der Brüggen, F. Howar, and J. Chen, "Thread carefully: Preventing starvation in the ROS 2 multithreaded executor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3588–3599, 2024.
[41] F. Vanhems, V. Rusu, D. Nowak, and G. Grimaud, "A formal correctness proof for an EDF scheduler implementation," in *Proceedings of the 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 281–292.
[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, 2008.
[43] M. H. Woodbury and K. G. Shin, "Evaluation of the probability of dynamic failure and processor utilization for real-time systems," in *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS)*, 1988, pp. 222–231.
[44] B. Ye, F. Marković, and B. B. Brandenburg, "Framework-agnostic model inference for intra-thread real-time tasks," in *Proceedings of the 32nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2026, pp. 68–81.
[45] R. Yerraballi, R. Mukkamala, K. Maly, and H. M. Abdel-Wahab, "Issues in schedulability analysis of real-time systems," in *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, 1995, pp. 87–92.
[46] F. Zhang, A. Burns, and S. K. Baruah, "Sensitivity analysis for EDF scheduled arbitrary deadline real-time systems," in *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2010, pp. 61–70.
[47] M. Zini, F. Marković, D. Casini, A. Biondi, and B. B. Brandenburg, "In search of butterflies: Exceedance analysis for real-time systems under transient overload," in *Proceedings of the 45th IEEE Real-Time Systems Symposium (RTSS)*, 2024, pp. 229–242.