Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel

Daniel B. de Oliveira[1,2,3], Rômulo S. de Oliveira[2], and Tommaso Cucinotta[3]

[1]RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.
[2]Department of Systems Automation, UFSC, Florianópolis, Brazil.
[3]RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.
Email: bristot@redhat.com, romulo.deoliveira@ufsc.br, tommaso.cucinotta@santannapisa.it

***Abstract*—This article proposes an automata-based model for describing and validating the behavior of threads in the Linux PREEMPT_RT kernel, on a single-core system. The automata model defines the events and how they influence the timeline of threads' execution, comprising the preemption control, interrupt handlers, interrupt control, scheduling and locking. This article also presents the extension of the Linux trace features that enable the trace of the kernel events used in the modeling. The model and the tracing tool are used, initially, to validate the model, but preliminary results were enough to point to two problems in the Linux kernel. Finally, the analysis of the events involved in the activation of the highest priority thread is presented in terms of necessary and sufficient conditions, describing the delays occurred in this operation in the same granularity used by kernel developers, showing how it is possible to take advantage of the model for analyzing the thread wake-up latency, without any need for watching the corresponding kernel code.**

I. INTRODUCTION

Real-time Linux has been successfully used throughout a number of academic and industrial projects as a fundamental building block of real-time distributed systems, from distributed and service-oriented infrastructures for multimedia [1], robotics [2], sensor networks [3] and factory automation [4], to the control of military drones [5] to distributed high-frequency trading systems [6], [7]. This is possible thanks to a set of operations that ensure the deterministic operation of Linux, while reducing the operating system noise. These operations, however, require in-kernel synchronization that can cause non-negligible delays, even for non-explicitly related tasks [8]. The synchronization is necessary because of the non-atomic nature of a sophisticated operating system like Linux. The understanding of the synchronization primitives, and how they affect the timing behavior of a thread, are fundamental for the development of real-time software for Linux.

However, the amount of effort required to understand all these constraints is not negligible. It might take years for a newcomer to understand the internals of the Linux kernel. The complexity of Linux is indeed a barrier, not only for researchers but for developers as well. Inside the kernel, scheduling operations interact with low-level details of the underlying processor and memory architectures, where complex locking protocols and "hacks" are used. The challenge is then, to describe such operations, using a level of abstraction that removes the complexity due to the kernel code. The description must use a format that facilitates the understanding of Linux dynamics for real-time researchers, without being too far from the way developers observe and improve Linux.

The developers of Linux observe and debug the timing properties of Linux using the tracing features present in the kernel. They interpret a chain of events, trying to identify the states that cause "*latencies*" in the activation of the highest priority thread, and then try to change kernel algorithms to avoid such delays. For instance, they use `ftrace` [9] or `perf`[1] to trace kernel events like interrupt handling, wakeup of a new thread, context switch, etc.. While `cyclictest` measures the "*latency*" of the system.

The notion of *events*, *traces* and *states* used by developers are common to Discrete Event Systems (DES). The admissible sequences of events that a DES can produce or process can be formally modeled through a *language*. The *language* of a DES can be modeled in many formats, like regular expressions, Petri nets and automata.

Paper Contributions: This article proposes an automata-based model describing the possible interleaving sequences of kernel events in the code path handling the execution of threads, IRQs and NMIs in the kernel, on a single-core system. The model covers also kernel code related to locking mechanisms, such as mutexes, read/write semaphores and read/write locks, including the possibility of nested locks, as for example in the locking primitives own code.

This article also presents the extension of the kernel tracing mechanism used to capture traces of the kernel events used in the model, to enable validation of the model by applying a modified `perf` tool running in user-space against traces captured from the live running system. Two problems were found in the Linux kernel code, regarding scheduler and tracing, by using our model. Finally, this paper demonstrates how the model can improve the understanding of Linux properties in logical terms, in the same granularity used by developers, but without the need of reading the kernel code.

II. RELATED WORK

This section presents prior literature relevant to the work being presented in this paper, spanning across two major areas: use of automata in real-time systems analysis, and formal software verification techniques successfully applied to the verification of kernel code in operating systems.

---

[1] More information at: http://man7.org/linux/man-pages/man1/perf.1.html.

a) Automata-based real-time systems analysis: Automata and discrete-event systems have been extensively used to verify timing properties of real-time systems. For example, in [10], a methodology based on timed discrete event systems is presented to ensure that a real-time system with multiple-period tasks is reconfigured dynamically using a safe execution sequence. In [11], the Kronos tool is used for checking properties of models based on timed automata.

In [12], parametric timed automata are used for the symbolic computation of the region of the parameters' space guaranteeing schedulability of a given real-time task set, under fixed priority scheduling. Additionally, some authors [13] considered composability of automata-based timing specifications, so that timing properties of a complex real-time system can be verified with reduced complexity.

In [14], the TIMES tool is used with an automata-based formalism to describe a network of distributed real-time components for analyzing their temporal behavior from the viewpoint of schedulability. Similar is the approach of UPPAAL [15].

Compared to the work being presented here, the mentioned methodologies focus on modeling the timing behavior of the applications, and their reciprocal interferences due to scheduling, neglecting the exact sequence of steps executed by an operating system kernel and its process scheduler, in order to let, for example, a higher-priority task preempt a lower-priority one. These details can be fundamental to ensure the build of an accurate formal model of the possible interferences among tasks, as shown in this paper.

b) Formal methods for OS kernels: An area that is particularly challenging is the one of verification of an operating system kernel and its various components. Some works that addressed this problem include the BLAST tool [16], where control flow automata have been used, combining existing techniques for state-space reduction based on abstraction, verification and counterexample-driven refinement, with *lazy abstraction*. Interestingly, the authors applied the technique to the verification of safety properties of OS drivers for the Linux and Microsoft Windows NT kernels.

Chaki et al. [17] proposed MAGIC, a tool for automatic verification of sequential C programs against finite state machine specifications. The tool can analyze a direct acyclic graph of C functions, by extracting a finite state model from the C source code, then reducing the verification to a Boolean satisfiability (SAT) problem. Interestingly, MAGIC has been used to verify correctness of a number of functions in the Linux kernel involved in syscalls handling mutexes, sockets, and packet sending. The tool has also been extended later to handle concurrent software systems [18], albeit the authors focus on verifying correctness and deadlock-freedom in presence of message-passing based concurrency, forbidding the sharing of variables. Authors were able to find a bug in the Micro-C/OS source code, albeit when they notified developers the bug had already been found and fixed in a newer release.

Another remarkable work is the `lockdep` mechanism [19] built into the Linux kernel, capable of identifying errors in using locking primitives that might eventually lead to deadlocks. The mechanism includes detection of mistaken order of acquisition of multiple (nested) locks throughout multiple kernel code paths, and detection of common mistakes in handling spinlocks across IRQ handler vs process context, e.g., acquiring a spinlock from process context with IRQs enabled as well as from an IRQ handler.

In [20], a formal memory model is introduced to automate verification of consistency properties of core kernel synchronization operations for a number of different architectures and associated memory consistency models.

In [21], a model of an RT system involving Linux is presented, with two OS domains: a real-time and a non-real-time one. These are abstracted as a seven and three states model, respectively. The model, however, is a high-level one and does not consider the internal details of the Linux kernel.

To the best of our knowledge, none of the above techniques ventured into the challenging goal of building a formal model for the understanding and validation of the Linux PREEMPT_RT kernel code sections responsible for such low-level operations such as task scheduling, IRQ and NMI management, and their delicate interplay, as done in this paper.

The only exception is the work in [22], where the idea of building an automata-based model for the Linux kernel was sketched out, presenting a very preliminary model focusing on IRQ and NMI only. The present paper presents a much more complete model, encompassing kernel events related to NMI, IRQ, threads management and locking code in the Linux PREEMPT_RT kernel, describing internals of our modifications to the `perf` tool, discussing its performance and overheads, and presenting two major results obtained applying the technique, that allowed us to track down problems in the scheduler and tracing code paths within the kernel, discussed with and confirmed by main kernel developers.

Finally work exists that tries to combine theoretical analytical real-time system models with empirical worst-case estimations based on a Linux OS [23]. There, the author introduced an "overhead-aware" evaluation methodology for a variety of considered analysis techniques, with multiple steps: first, each scheduling algorithm to be evaluated is implemented on the LITMUS RT platform, then hundreds of benchmark task sets are run, gathering average and maximum values for what the authors call scheduling overheads, then these figures are injected into overhead-aware real-time analysis techniques. Now, the key comparison point with the present work, is that we aim at explaining at a finer-grained level of detail what these scheduling overheads are, where they originate from and why, when referring to the Linux kernel, and specifically to its PREEMPT_RT variant. The discussion around outliers in [23], along with the explicit admission of the need for removing manually some of them, witnesses the need for a more insightful model that provides more accurate information of said overheads. Our automata-based model, that will be detailed in the next sections, sheds some light exactly into this direction.

III. BACKGROUND

We model the succession of events in the Linux kernel over time as a Discrete Event System. A DES can be described in various ways, for example using a *language* (that represents

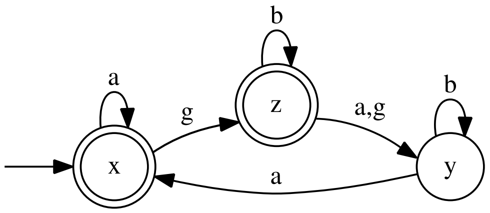

Fig. 1: State transitions diagram (based on Fig. 2.1 from [24]).

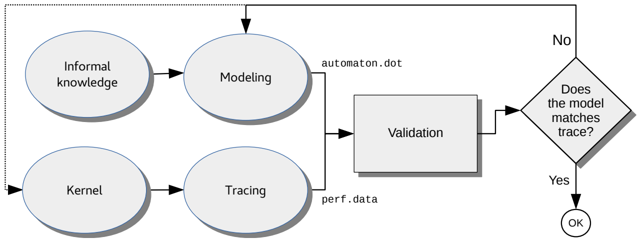

Fig. 2: Modeling Phases.

the valid sequences of events that can be observed during the evolution of the system). Informally speaking, an automaton is a formalization used to model a set of well-defined rules that define such a language.

The evolution of a DES is described with all possible sequence of events $e_1, e_2, e_3, ...e_n$, $e_i \in E$, defining the language $\mathcal{L}$ that describes the system. There are many possible ways to describe the language of a system. For example, it is possible to use regular expressions. For complex systems, more flexible modeling formats, like automaton, were developed. Automata are characterized in the typical directed graph or state transition diagram representation. For example, consider the event set $E = \{a, b, g\}$ and the state transition diagram in Figure 1, where nodes represent system states, labeled arcs represent transitions between states, the arrow points to the initial state and the nodes with double circles are *marked states*, i.e., safe states of the system. Formally, a deterministic automaton, denoted by G, is a tuple $G = \{X, E, f, \Gamma, x_0, X_m\}$ where: X is the set of states; E is the set of events; $f : X \times E \to X$ is the transition function, defining the state transition between states from X due to events from E; $\Gamma : X \implies 2^E$ is the active (or feasible) event function, i.e., $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined in the state x; x_0 is the initial state and $X_m \subseteq X$ is the set of marked states.

For instance, the automaton G represented in Figure 1 can be described by: $X = \{x, y, z\}$, $E = \{a, b, g\}$, $f(x, a) = x$, $f(x, g) = z$, $f(y, a) = x$, $f(y, b) = y$, $f(z, b) = z$, $f(z, a) = f(z, g) = y$, $\Gamma(x) = \{a, g\}$, $\Gamma(y) = \{a, b\}$, $\Gamma(z) = \{a, b, g\}$, $x_0 = x$ and $X_m = \{x, z\}$. The automaton starts from the initial state x_0 and moves to a new state $f(x_0, e)$ upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$. This process continues based on the transitions for which f is defined.

Informally, following the graph of Figure 1 it is possible to see that the occurrence of event a, followed by event g and a will lead from the initial state to state y. The language $\mathcal{L}(G)$ generated by an automaton $G = \{X, E, f, \Gamma, x_0, X_m\}$ consists of all possible chains of events generated by the state transition diagram starting from the initial state.

One important language generated by an automaton is the *marked language*. This is the set of words in $\mathcal{L}(G)$ that lead to marked states. The marked language is also called the language recognized by the automaton. When modeling systems, a marked state is generally interpreted as a possible final or safe state for a system.

Automata theory also enables operations among automata. An important operation is the *parallel composition* of two or more automata that are combined to compose a single, augmented-state, automaton. This allows for merging two or more automata models into one single model, constituting the standard way of building a model of the entire system from models of individual components [24].

A. *Monolithic vs. modular modeling*

In modeling complex systems using automata, there are two possible approaches, the *monolithic* and the *modular* one [25].

In the monolithic approach the system is modeled as a single automaton. Although this approach is good for simple systems, it is not efficient in the modeling of complex systems, as the number of states increases exponentially. In the modular approach, rather than specifying a single automaton, the system is modeled as a set of independent sub-systems, where each sub-system has its own alphabet. For systems composed of many independent sub-systems, with several specifications, the modular approach turns out to be more efficient.

In the modular approach, a *generator* of events of each sub-system is modeled independently. The synchronization rules of each sub-system are then stated as a set of *specification* automata. Each *specification* synchronizes the actions of two or more *generators*. The parallel composition of all the generators and specifications creates the model of the system and its synchronizations.

IV. MODELING

Following the approach presented in Figure 2, the knowledge about Linux tasks is modeled as an automaton using the modular approach. The main sources of information, in order of importance, are the observation of the system's execution using various tracing tools [9], the kernel code analysis, academic documentation about Linux and real-time systems [8], and hardware documentation [26].

At the same time, we observe a real system running. The development of the model uses the Linux vanilla kernel with the PREEMPT_RT patchset applied. This work is based on the *fully-preemptive* mode only, that is the mode utilized by the real-time Linux community. The configuration options of this kernel are based on the configuration of the Red Hat Enterprise Linux for Real Time, an enterprise version of Linux with the PREEMPT_RT patchset, with kernel version *v4.14.15-rt13*. However, the kernel was configured to run on a single CPU.

During the development of the model, the abstractions from the kernel are transformed into automata models. Initially, the identification of the system is made using the `tracepoints` already available. However, the existing tracepoints were not enough to explain the behavior of the system satisfactorily. For

TABLE I: Automaton and Kernel events relation. Events in bold font were added to the kernel.

Kernel event	**Automaton event**	**Description**
IRQ related tracepoints		
hw_local_irq_disable	**irq:local_irq_disable**	Begin IRQ handler
hw_local_irq_enable	**irq:local_irq_enable**	Return IRQ handler
local_irq_disable	**irq:local_irq_disable**	Mask IRQs
local_irq_enable	**irq:local_irq_enable**	Unmask IRQs
nmi_entry	**irq_vectors:nmi**	Begin NMI handler
nmi_exit	**irq_vectors:nmi**	Return NMI Handler
Preemption/Scheduler related events		
preempt_disable	**sched:sched_preempt_disable**	Disable preemption
preempt_enable	**sched:sched_preempt_enable**	Enable preemption
preempt_disable_sched	**sched:sched_preempt_disable**	Disable preemption to call the scheduler
preempt_enable_sched	**sched:sched_preempt_enable**	Enables preemption returning from the scheduler
schedule_entry	**sched:sched_entry**	Begin of the scheduler
schedule_exit	**sched:sched_exit**	Return of the scheduler
sched_need_resched	**sched:set_need_resched**	Set need resched
State of the thread related events		
sched_waking	sched:sched_waking	Activation of a thread
sched_set_state_runnable	**sched:sched_set_state**	Thread is runnable
sched_set_state_sleepable	**sched:sched_set_state**	Thread can go to sleepable
Context switch related events		
sched_switch_in	sched:sched_switch	Switch in of the thread under analysis
sched_switch_suspend	sched:sched_switch	Switch out due to a suspension of the thread under analysis
sched_switch_preempt	sched:sched_switch	Switch out due to a preemption of the thread under analysis
sched_switch_blocking	sched:sched_switch	Switch out due to a blocking of the thread under analysis
sched_switch_in_o	sched:sched_switch	Switch in of another thread
sched_switch_out_o	sched:sched_switch	Switch out of another thread
Mutex related events		
mutex_lock	**lock:rt_mutex_lock**	Requested a RT Mutex
mutex_blocked	**lock:rt_mutex_block**	Blocked in a RT Mutex
mutex_acquired	**lock:rt_mutex_acquired**	Acquired a RT Mutex
mutex_abandon	**lock:rt_mutex_abandon**	Abandoned the request of a RT Mutex
Read/Write Lock/Semaphore related events		
write_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as writer
write_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as writer
write_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as writer
write_abandon	**lock:rwlock_abandon**	Abandoned a R/W Lock or Sem as writer
read_lock	**lock:rwlock_lock**	Requested a R/W Lock or Sem as reader
read_blocked	**lock:rwlock_block**	Blocked in a R/W Lock or Sem as reader
read_acquired	**lock:rwlock_acquired**	Acquired a R/W Lock or Sem as reader
read_abandon	**lock:rwlock_abandon**	Abandon a R/W Lock or Sem as reader

example, the `sched:sched_waking` `tracepoint` is sufficient to inform the activation of a thread. Although it includes the `prio` field to vehicle the priority of the just awakened thread, this is not enough to determine whether the thread has the highest priority or not. For instance, the `SCHED_DEADLINE` does not use the `prio` field, but the thread's absolute deadline. When a thread becomes the highest priority one, the flag `TIF_NEED_RESCHED` is set for the current running thread. This causes invocation of the scheduler at the next scheduling point. Hence, the event that most precisely defines that another thread has the highest priority task is the event that sets the `TIF_NEED_RESCHED` flag. Since the standard set of Linux's tracepoints does not include an event to notify the setting of `TIF_NEED_RESCHED`, a new tracepoint needed to be added. In such cases, new `tracepoints` were added to the kernel. These new `tracepoints` are highlighted in Table I.

A. Events

Table I presents the events used in the automata modeling and their related kernel events. When a kernel event refers to more than one automaton event, the extra fields of the kernel event are used to distinguish between automaton events.

Linux kernel evolves very fast. For instance, in a very recent release (4.17), around 1.559.000 lines were changed (690000 additions, 869000 deletions) [27]. This makes natural the rise of the question: *How often do the events and abstractions utilized in this model change?* Despite the continuous evolution of the kernel, some principles stay stable for a long time. IRQs and the possibility of masking them are present in Linux since its very early days. The fully preemptive model, and the functions to disable preemption are present since the early days of the PREEMPT_RT, dating back to year 2005 [28]. It is worth noting that the scheduling and locking related events are implementation independent. For instance, the model does not refer to any detail about how specific schedulers' implementations define which thread to pick next (highest priority, earliest deadline). The same is valid for locking: the model is independent from details about the specific implementation of locking primitives. These might even change, but the events and their effects in the timeline of threads stay invariable. The abstractions used in this paper was discussed with the main Linux kernel developers and maintainers of the real-time, scheduling and tracing sub-systems [29], [30].

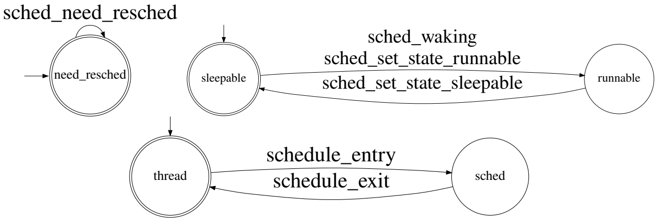

Fig. 3: Examples of generators: *G05* Need Resched (on top, left), *G01* Sleepable and Runnable (on top, right) and *G04* Scheduling Context (bottom).

B. Modeling

The automata model have been developed using the Supremica IDE [31]. Supremica is an integrated environment for verification, synthesis, and simulation of discrete event systems using finite automata. Supremica allows exporting the result of the modeling in the DOT format that can be plotted using `graphviz` [32], for example.

The model was developed using the modular approach. All modules were developed manually. The generators are the system's events of Table I modeled as a set of independent sub-systems. Each sub-system has a private set of events. Similarly, each specification is modeled independently, but using the events of the sub-systems of the generators it aims to synchronize.

Examples of generators are shown in Figure 3. The Need resched generator (*G05*) contains only one event and one state. The Sleepable or Runnable generator (*G01*) has two states. Initially, the thread is in the `sleepable` state. The events `sched_waking` and `sched_set_state_runnable` cause a state change to `runnable`. The event `sched_set_state_sleepable` returns the task to the initial state. The Scheduling Context (*G04*) models the call and return of the main scheduling function of Linux, which is `__scheduler()`.

Table II shows statistics information about the Generators and Specifications that compose the model. The final model is generated from the parallel composition modular models. The parallel composition is done via `Supremica` tool. The final model has 34 events, 13906 states and 31708 transitions. The

```
Reference model: isorc.dot
+---->  +=thread of interest - .=other threads
| +-> T=Thread - I=IRQ - N=NMI
| |
| |     TID |   timestamp   | cpu |           event               | state   | safe?
. T       8    436.912532    [000]               preempt_enable ->       q0 safe
. T       8    436.912534    [000]            local_irq_disable ->    q8102
. T       8    436.912535    [000]              preempt_disable ->   q19421
. T       8    436.912535    [000]                 sched_waking ->      q99
. T       8    436.912535    [000]           sched_need_resched ->   q14076
. T       8    436.912535    [000]             local_irq_enable ->    q1965
. T       8    436.912536    [000]               preempt_enable ->   q12256
. T       8    436.912536    [000]        preempt_disable_sched -> q18615,q23376
. T       8    436.912536    [000]               schedule_entry -> q16926,q17108,q2649,q7400
. T       8    436.912537    [000]            local_irq_disable -> q11700,q14046,q21391,q23792
. T       8    436.912537    [000]           sched_switch_out_o -> q10337,q20018,q21933,q7672
. T       8    436.912537    [000]              sched_switch_in -> q10268,q20126
+ T    1840    436.912537    [000]             local_irq_enable ->   q20036
+ T    1840    436.912538    [000]                schedule_exit ->   q21033
```

Fig. 4: Example of the `perf thread_model` output: a thread activation.

TABLE II: Automata models.

Name	States	Events	Transitions
G01 Sleepable or runnable	2	3	3
G02 Context switch	2	4	4
G03 Context switch other thread	2	2	2
G04 Scheduling context	2	2	2
G05 Need resched	1	1	1
G06 Preempt disable	3	4	4
G07 IRQ Masking	2	2	2
G08 IRQ handling	2	2	2
G09 NMI	2	2	2
G10 Mutex	3	4	6
G11 Write lock	3	4	6
G12 Read lock	3	4	6
S01 Sched in after wakeup	2	3	5
S02 Resched and wakeup sufficency	3	10	18
S03 Scheduler with preempt disable	2	4	4
S04 Scheduler doesn't enable preemption	2	6	6
S05 Scheduler with interrupt enabled	2	4	4
S06 Switch out then in	2	20	20
S07 Switch with preempt/irq disabled	3	10	14
S08 Switch while scheduling	2	8	8
S09 Schedule always switch	3	6	6
S10 Preempt disable to sched	2	3	4
S11 No wakeup right before switch	3	5	8
S12 IRQ context disable events	2	27	27
S13 NMI blocks all events	2	34	34
S14 Set sleepable while running	2	6	6
S15 Don't set runnable when scheduling	2	4	4
S16 Scheduling context operations	2	3	3
S17 IRQ disabled	3	4	4
S18 Schedule necessary and sufficient	7	9	22
S19 Need resched forces scheduling	7	27	59
S20 Lock while running	2	16	16
S21 Lock while preemptive	2	16	16
S22 Lock while interruptible	2	16	16
S23 No suspension in lock algorithms	3	10	19
S24 Sched blocking if blocks	3	10	20
S25 Need resched blocks lock ops	2	15	17
S26 Lock either read or write	3	6	6
S27 Mutex doesn't use rw lock	2	11	11
S28 RW lock does not sched unless block	4	11	22
S29 Mutex does not sched unless block	4	7	16
S30 Disable IRQ in sched implies switch	5	6	10
S31 Need resched preempts unless sched	3	5	11
S32 Does not suspend in mutex	3	5	11
S33 Does not suspend in rw lock	3	8	16
Model	13906	34	31708

complete model has only one final state, has no forbidden states, it is deterministic and non-blocking.

The complete model exposes the complexity of Linux. At a first glance, the number of states seems to be excessively high. But, for instance, as it is not possible to mask NMIs, these can take place in all states, doubling the number of states, and adding two more transitions for each state. The complexity, however, can be simplified if analyzed at the generators and specifications level. By breaking the complexity into small specifications, the understanding of the system becomes more natural. For instance, the most complex specification has only seven states. The modular modeling approach can provide a simple view of small parts of the system, facilitating the understanding by humans, while providing the entire picture of the system, making the validation of the trace more efficient.

C. Model Validation

The `perf` tracing tool was extended to automate the validation of the model against the execution of the real system. The `perf` extension is called `thread_model`. The `perf thread_model` has two operation modes. In `record` mode, the `tracepoints` presented in Table I are enabled, and recorded into a `perf.data` file. This phase involves both the Linux kernel tracing features and `perf` itself in user-space. In the kernel, `tracepoints` are enabled, recording the events in the trace buffer. This operation is done using lock-free primitives, that do not generate events involved in the model. Hence, the kernel part does not influence the model validation. Due to the high granularity of data, a typical 30 seconds trace of the system running `cyclictest` as workload, generates around 27000000 events, amounting 2.5 GB of data. To avoid having to collect the trace buffer data very frequently, a 3 GB trace buffer was allocated. The high number of events is due to background activities from Linux. For example, the periodic scheduler tick, RCU activities, network and disk operations, and so on. The user-space side periodically collects the trace from the trace-buffer, saving the data to a file. This generates additional events that are analyzed as any regular process.

After recording, the analysis of the data is done using the `perf thread_model report` mode. This is the core of the validation tool. The `report` mode has three basic arguments:

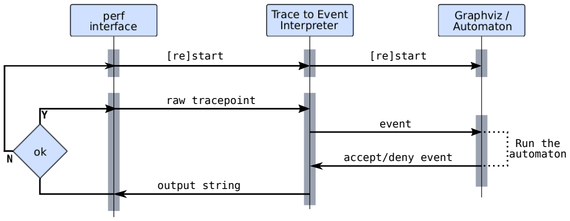

Fig. 5: `perf task_model` structure.

The model exported by `Supremica` in the `.dot` format; the `perf.data` file containing the trace; and the `pid` of the thread to analyze. The modules of the tool are presented in Figure 5. The `perf interface` is part of `perf`. The `Grapviz` library is used to parse the `.dot` file. Between these two components, the `Trace to Event Interpreter` was developed. When starting, the `perf` interface opens the trace file, and the `Grapviz` is used to open the model. The `Trace to Event Interpreter` is initialized, setting initial states and data. After the initial setup, `perf` handles the `raw tracepoints` to the `Trace to Event Interpreter`, that translates the `trace` to *event*, that is then tried in the automaton. If the automaton accepts the event, the regular output is printed. If not, an error message is printed. Either way, the trace continues to be parsed and evaluated until the end of the trace file.

The validation is done using the complete model. The advantage of using the complete model is that one kernel transition generates only one transition in the model. Hence the validation of each event is done in linear time (*O(1)*) for each event. This is a critical point, given the number of states in the model, and the amount of data from the kernel. On the adopted platform, each GB of data is evaluated in nearly 8 seconds. One example of output provided by `perf thread model` is shown in Figure 4.

It is then possible to use the `Supremica` simulation mode to identify the state of the automata, and the raw trace to determine the events generated and unexpected event. If the problem is in some automaton, it should be adapted to include the behavior presented by the kernel. However, it could be a problem in the kernel code or in the `perf` tool. Not surprisingly, kernel bugs were found in the scheduler and perf/trace. The first was an inefficiency bug in the kernel `schedule()` function. The fix suggested by the authors was accepted and is already included in all PREEMPT_RT versions under support [33]. The second is a bug in the trace-subsystem, which is dropping events due to a problem in the detection of nesting of events [2]. The second problem was acknowledged by developers, but no solution was found yet.

The source code of the model in the format used by `Supremica`, the kernel patch with kernel and `perf` modifications and more information about how to use the model and and reproduce the experiments are available at this paper's *Companion Page* [34].

V. APPLICATIONS OF THE MODEL: ANALYSIS OF ACTIVATION OF THE HIGHEST PRIORITY THREAD

This section analyzes the models related to the activation of the highest priority thread. This behavior is important because it is part of the principal metric utilized by the PREEMPT_RT developers, the *latency*.

The generators that act during the activation of a thread are described first, followed by the specifications. Then, specifications and generators are used to explain the possible paths, and how they influence the activation delay.

A. Generators

The model considers three types of tasks: 1) NMI; 2) IRQs and 3) Threads. The generator *G09* in Figure 6 show the events that represent the execution of an NMI. The NMI can always take place, hence interfering in the execution of threads and IRQs. The second type of tasks are IRQs. Before starting the handling of an IRQ, the processor masks interrupts to avoid reentrancy in the interrupt handler. Although it is not possible to see actions taken by the hardware from the operating system point of view, the `irqsoff` tracer of the Linux kernel has a hook in the very beginning of the handler, that is used to take note that IRQs were masked [22]. In such a way to reduce the number of events and states, the events that inform the starting of an interrupt handler were suppressed, and the notification of interrupts being disabled by the hardware prior to the execution of the handler are used as the events that notify the start of the interrupt handler. The same is valid for the return from the handler. The last action in the return from the handler is the unmask of interrupts. This is used to identify the end of an interrupt handler. A thread can also postpone the start of the handler of an interrupt using the `local_irq_disable()` and `local_irq_enable()` like functions. The generator *G07* models the masking of the interrupts by a thread. The generator *G08* models the masking of the interrupts by the hardware to handle a hardware interrupt. These are presented in Figure 7.

A thread starts running after the scheduler completes execution. The scheduler context starts with the event `schedule_entry`, and finishes with the event `schedule_exit`, as modeled in generator *G04* (Figure 3).

The context switch operation changes the context from one thread to another. The model considers two threads. One is the thread under analysis, and the other represents all other threads in the system. On Linux, there is always one thread ready to run. That is because the idle state runs as if it was a thread, the lowest priority thread. In the initial state of the automata, any other thread is running. The context switch operations from or to the other threads are presented in Figure 8.

The context switch generator for the thread under analysis is slightly different. In the initial state, the thread is not running. After starting running, the thread can leave the processor in three different modes: 1) *suspending* the execution waiting for another activation; 2) *blocking* in a locking algorithm like Mutex, or read/write semaphores; or 3) suffering a *preemption* from a higher priority thread, as shown in Figure 9.

The thread is activated with the `sched_waking` event in the generator *G01*, the notification of a new highest priority

---

[2] http://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1811261.html

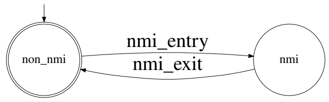

Fig. 6: *G09* NMI generator.

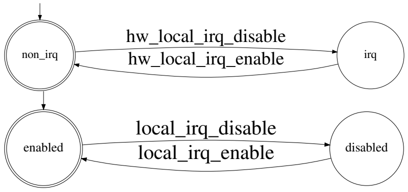

Fig. 7: *G08* IRQ Handling (Top); *G07* IRQ Masking (Bottom) generators.

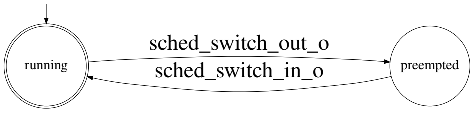

Fig. 8: *G03* Context switch other thread generator.

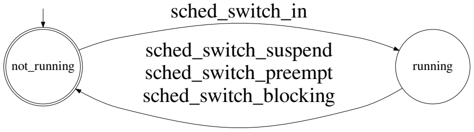

Fig. 9: *G02* Context switch generator.

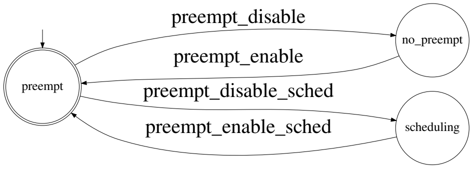

Fig. 10: *G06* Preempt disable.

thread, with `set_need_resched` event in the generator *G05*, as shown in Figure 3.

The last involved generator is about preemption. In the initial state, the preemption is enabled. But it can be disabled for two main reasons: first, to guarantee that the current thread will not be de-scheduled; second, to avoid reentrancy in the scheduler code when already executing the scheduler. In the first case, the `preempt_disable` and `preempt_enable` events are generated, the second case generates the events `preempt_disable_sched` and `preempt_enable_sched`. These two possibilities are modeled in the *G06*, as shown in Figure 10.

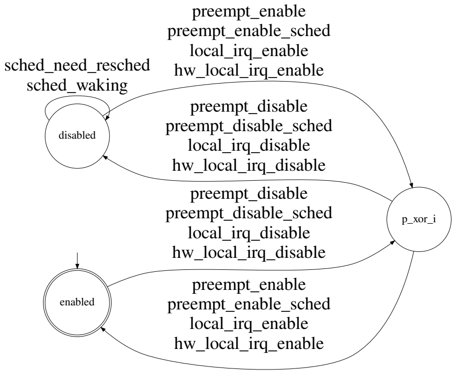

Fig. 11: *S02* Wakeup and Need resched takes place with IRQ and preemption disabled.

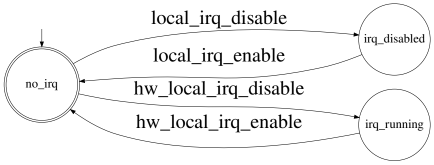

Fig. 12: *S17* IRQ disabled.

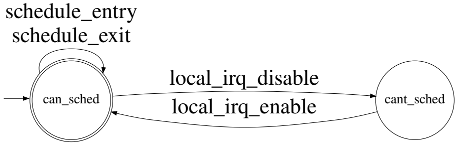

Fig. 13: *S05* Scheduler called with interrupts enabled.

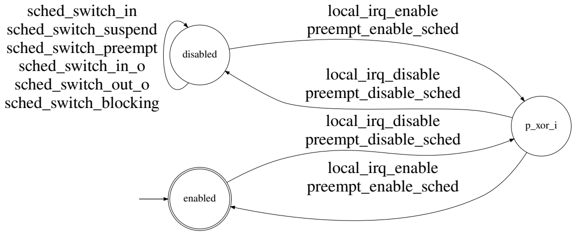

Fig. 14: *S07* Switch with interrupts and preempt disabled.

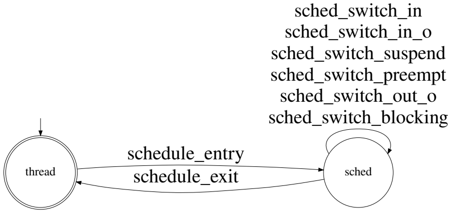

Fig. 15: *S08* Switch while scheduling.

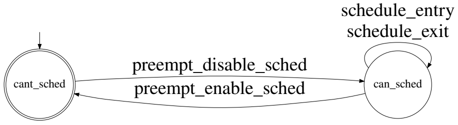

Fig. 16: *S03* Scheduler called with preemption disabled.

B. Specification

In Figure 11, the specifications *S02* shows the sufficient condition for the occurrence of both `sched_waking` and `sched_need_resched`: they can occur only with both preemption and IRQs disabled. By disabling both interrupts and preemption, the automaton moves to the state *disabled*, where it is possible to execute `sched_waking` and `sched_need_resched`. The automaton *S02* allows the sequence of events "`local_irq_disable`", "`hw_local_irq_disable`", giving the impression that it does not enforce both IRQ and preemption to be disabled. In fact, the specification *S02* does not forbid this sequence. This sequence is forbidden in the specification *S17* IRQ disabled, in Figure 12. The specification *S17* is a classical mutual exclusion. Interrupts are disabled either by hardware or by software, but never by both. This specification, along with

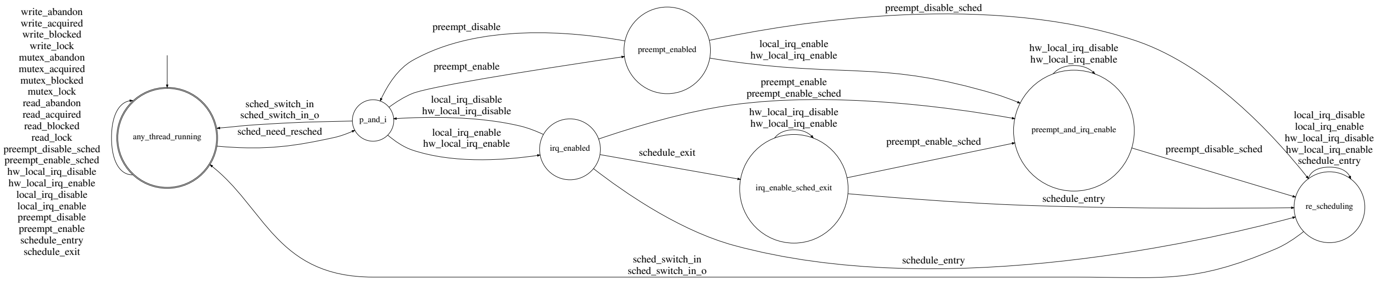

Fig. 17: *S19* Need resched forces scheduling.

the generator of the preemption disabled (*G06*), gives the properties needed to the specification *S02* to have both IRQs and preemption disabled in the `disabled` state.

The context switch of threads also depends on two main specifications: *S07* both preemption and IRQ should be disabled. However, with a slight difference of the specification *S02*: interrupts disabled in the thread context (not because of an IRQ), and preemption disabled during a scheduler call. Moreover, the context switch only happens inside the scheduling context, because of the specification *S08*. These specifications are presented in Figure 14 and 15, respectively. The scheduler execution has two main specifications as well: The specification *S03*, in Figure 16, restricts the execution of the scheduler for a non-preemptive section. However, the scheduler is always called with interrupts enabled, as modeled in the specification *S05* in Figure 13.

The main goal of the PREEMPT_RT is to schedule the highest priority thread as soon as possible. In the terms used in the model, the goal of the PREEMPT_RT developers is to cause `sched_switch_in` or `sched_switch_in_o` events after the occurrence of `set_need_resched` as soon as possible. The specification *S19*, in Figure 17, models this property.

The specifications explained so far described the sufficient conditions for these events. Given the sufficient conditions, the specification *S19* provides the necessary conditions to context switch to the highest priority thread.

In the initial state, the system runs without changing the state, unless `set_need_resched` takes place. Once `set_need_resched` occurs, the initial state will be possible only after the context switch in of a thread. Hence, `set_need_resched` is a necessary condition to cause a preemption, causing a context switch. When `set_need_resched` occurs, preemption and interrupts are known to be disabled (*S02*). Before returning to the initial state, the set of events that can happen are limited for those that deal with IRQ/IRQ masking, preemption and scheduling.

The return to the initial state is possible from two states: in the state *p_and_i*, and in the *re_scheduling*. The first case takes place when `set_need_resched` occurs in the scheduler execution. For instance, the sequence "`preempt_disable_sched`", "`schedule_entry`", "`local_irq_disable`" satisfies the specification *S02* for the `set_need_resched` and *S03*, *S05*, *S07* and *S08* for the context switch. This case represents the best case, where all sufficient conditions occurred before the necessary one.

If this is not the case, the return for the initial state can happen through a sole state, the *re_scheduling*. From the state *p_and_i* until *re_scheduling*, the calls to the scheduler function are enabled anytime sufficient conditions are met. However, this implies that preemption was disabled to call the scheduler (*S03*), which is the case of a thread running on the way to enter in the scheduler, or already in scheduling context (*G04*). This case, however, is not the point of attention for Linux developers. The point of interest for developers is in the cyclic part of the specification *S17*, between states *p_and_i*, *preempt_enabled*, and *irq_enabled*, in which either or both IRQs and preemption stays disabled, not allowing the progress of the system. Moreover, in the states in which IRQs are enabled, like *irq_enabled* and *preempt_and_irq_enable*, interrupt handlers can start running, postponing the context switch. Finally, NMIs can take place at any time, contributing to the delay. These operations that postpone the occurrence of the context switch are part of the *latency* measured by practitioners. The latency measurements, however, does not clarify the cause of the latency: the kernel is evaluated as a black box. By modeling the behavior of tasks on Linux, this work opens space for the creation of a novel set of evaluation metrics for Linux.

VI. CONCLUSIONS

The definition of the operations of the Linux kernel that affect the timing behavior of tasks is fundamental for the improvement of the real-time Linux state-of-the-art. By using the modular approach, it was possible to model the essential behavior of Linux utilizing a set of small and easily understood automata. The synchronization of these small automata resulted in an automaton that represents the entire system. The development of the validation method/tooling was simplified because of the shared abstraction of "events".

It is possible to use the model to aid the understanding of complex behavior of Linux, with the benefit of not requiring the knowledge of the entire model. For example, the explanation presented in Section V used only a set of specifications, not all the models. Although the authors expected that the usage of the model could help in the debugging of Linux in future works, the fact that the model produced practical results already during its development was a pleasant surprise.

One main aspect of Linux is the capacity of evolving, creating a new reality of fully distributed systems, for example, with containers and micro-services. Verifying that the changes in the kernel code do not create regressions, breaking the model and the guarantees provided by the PREEMPT_RT, is a major concern of developers. The idea of using the automata model to verify the kernel was presented to the main Linux kernel developers, and there is a consensus that the given approach should be integrated, mainly to improve testing of the logical correctness of the kernel [35], but also for timing regressions, with the creation of new metrics for the PREEMPT_RT kernel [36]. Further improvements in the tooling should be done to arrive in such state. For instance by improving the performance of the tracing by using `eBPF`. The approach has also potential to be used in another areas of the kernel, by the modeling of other components.

The natural continuation of this work is the modeling of the multiprocessor behavior of Linux. Furthermore, a useful follow-up research would be an attempt to merge this kind of model with existing real-time schedulability analysis techniques, in order to verify the usefulness of the more accurate modeling of the OS/kernel relatively complex code, in the case of PREEMPT_RT Linux.

REFERENCES

[1] V. Vardhan, W. Yuan, A. F. H. III, S. V. Adve, R. Kravets, K. Nahrstedt, D. G. Sachs, and D. L. Jones, "GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy," *IJES*, vol. 4, no. 2, pp. 152–169, 2009. [Online]. Available: https://doi.org/10.1504/IJES.2009.027939

[2] C. San Vicente Gutiérrez, L. Usategui San Juan, I. Zamalloa Ugarte, and V. Mayoral Vilches, "Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications," Aug 2018. [Online]. Available: https://arxiv.org/pdf/1808.10821.pdf

[3] A. Dubey, G. Karsai, and S. Abdelwahed, "Compensating for timing jitter in computing systems with general-purpose operating systems," in *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, March 2009, pp. 55–62.

[4] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina, "A real-time service-oriented architecture for industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 3, pp. 267–277, Aug 2009.

[5] J. Condliffe, "U.s. military drones are going to start running on linux," https://gizmodo.com/u-s-military-drones-are-going-to-start-running-on-linu-1572853572, Jul 2014.

[6] J. Corbet, "Linux at NASDAQ OMX," https://lwn.net/Articles/411064/, Oct 2010.

[7] H. Chishiro, "Rt-seed: Real-time middleware for semi-fixed-priority scheduling," in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*.

[8] D. B. de Oliveira and R. S. de Oliveira, "Timing analysis of the PREEMPT_RT Linux kernel," *Softw., Pract. Exper.*, vol. 46, no. 6, pp. 789–819, 2016.

[9] S. Rostedt, "Secrets of the Ftrace function tracer," *Linux Weekly News*, January 2010, available at: http://lwn.net/Articles/370423/ [last accessed 09 May 2017].

[10] X. Wang, Z. Li, and W. M. Wonham, "Dynamic Multiple-Period Reconfiguration of Real-Time Scheduling Based on Timed DES Supervisory Control," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 101–111, Feb 2016.

[11] C. Daws and S. Yovine, "Two examples of verification of multirate timed automata with Kronos," in *Proceedings 16th IEEE Real-Time Systems Symposium*, Dec 1995, pp. 66–75.

[12] A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic Computation of Schedulability Regions Using Parametric Timed Automata," in *2008 Real-Time Systems Symposium*, Nov 2008, pp. 80–89.

[13] K. Lampka, S. Perathoner, and L. Thiele, "Component-based system design: analytic real-time interfaces for state-based component implementations," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 3, pp. 155–170, Jun 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0257-7

[14] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for schedulability analysis and code generation of real-time systems," in *Formal Modeling and Analysis of Timed Systems*, K. G. Larsen and P. Niebert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 60–72.

[15] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, Sept 2004.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 58–70.

[17] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 388–402, June 2004.

[18] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, "Concurrent software verification with states, events, and deadlocks," *Formal Aspects of Computing*, vol. 17, no. 4, pp. 461–483, Dec 2005.

[19] J. Corbet, "The kernel lock validator," https://lwn.net/Articles/185666/, May 2006.

[20] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 405–418.

[21] H. Posadas, E. Villar, D. Ragot, and M. Martinez, "Early modeling of linux-based rtos platforms in a systemc time-approximate co-simulation environment," in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, May 2010, pp. 238–244.

[22] D. B. de Oliveira, R. S. de Oliveira, T. Cucinotta, and L. Abeni, "Automata-based modeling of interrupts in the Linux PREEMPT RT kernel," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2017, pp. 1–8.

[23] B. B. Brandenburg, *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*, 2011. [Online]. Available: https://cs.unc.edu/∼anderson/diss/bbbdiss.pdf

[24] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[25] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control Optim.*, vol. 25, no. 1, pp. 206–230, Jan. 1987.

[26] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol. 3*, September 2016, no. 325384-060US.

[27] J. Corbet, "Statistics from the 4.17 kernel development cycle," May 2018. [Online]. Available: https://lwn.net/Articles/756031/

[28] P. McKenney, "A realtime preemption overview," August 2005. [Online]. Available: https://lwn.net/Articles/146861/

[29] D. B. de Oliveira, "Mind the gap between real-time Linux and real-time theory, Part I," 2018. [Online]. Available: https://wiki.linuxfoundation.org/realtime/events/rt-summit2018/schedule#abstracts

[30] ——, "Mind the gap between real-time Linux and real-time theory, Part II," 2018. [Online]. Available: https://www.linuxplumbersconf.org/event/2/contributions/75/

[31] K. Akesson, M. Fabian, H. Flordal, and R. Malik, "Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems," in *2006 8th International Workshop on Discrete Event Systems*, July 2006, pp. 384–385.

[32] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz – open source graph drawing tools," in *International Symposium on Graph Drawing*. Springer, 2001, pp. 483–484.

[33] D. B. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Modeling the Behavior of Threads in the PREEMPT_RT Linux Kernel Using Automata," in *Proceedings of the Embedded Operating System Workshop (EWiLi)*, Turin, Italy, October 2018.

[34] D. B. de Oliveira, "Companion Page for ISORC 2019 paper," 2019. [Online]. Available: http://bristot.me/isorc-2019/

[35] ——, "How can we catch problems that can break the PREEMPT_RT preemption model?" 2018. [Online]. Available: https://linuxplumbersconf.org/event/2/contributions/190/

[36] ——, "Beyond the latency: New metrics for the real-time kernel," 2018. [Online]. Available: https://linuxplumbersconf.org/event/2/contributions/241/