A Thread Synchronization Model for the PREEMPT_RT Linux Kernel

Daniel B. de Oliveira[a,b,c], Rômulo S. de Oliveira[b], Tommaso Cucinotta[c]

[a] *RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.*

[b] *Department of Systems Automation, UFSC, Florianópolis, Brazil.*

[c] *RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.*

Abstract

This article proposes an automata-based model for describing and validating sequences of kernel events in Linux PREEMPT_RT and how they influence the timeline of threads' execution, comprising preemption control, interrupt handling and control, scheduling and locking. This article also presents an extension of the Linux tracing framework that enables the tracing of kernel events to verify the consistency of the kernel execution compared to the event sequences that are legal according to the formal model. This enables cross-checking of a kernel behavior against the formalized one, and in case of inconsistency, it pinpoints possible areas of improvement of the kernel, useful for regression testing. Indeed, we describe in details three problems in the kernel revealed by using the proposed technique, along with a short summary on how we reported and proposed fixes to the Linux kernel community. As an example of the usage of the model, the analysis of the events involved in the activation of the highest priority thread is presented, describing the delays occurred in this operation in the same granularity used by kernel developers. This illustrates how it is possible to take advantage of the model for analyzing the preemption model of Linux.

Keywords:
Real-time computing, Operating systems, Linux kernel, Automata, Software verification, Synchronization.

1. 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] and distributed high-frequency trading systems [6, 7]. However, there is still a gap between the restrictions imposed in

the task model used in academic work, and the restrictions imposed by the synchronization needed in the real Linux implementation. For instance, the frequent use of assumptions like *tasks are completely independent*, *the operating system is (fully) preemptive*, and *operations are atomic* [8, 9] is a frequent critique from Linux developers. They argue that such restrictions do not reproduce the reality of real-time applications running on Linux, raising doubts about the results of the development of theoretical schedulers when put in practice [10]. On the other hand, Linux was not designed as a real-time operating system (OS), and so does not use the conventions already established in the real-time academic community. For example, the main evaluation metric used on Linux, the *latency*, is an oversimplification of the main metric utilized in the academy, i.e., the response time of tasks [11].

On Linux, tasks can interfere with each other in a non-negligible way both *explicitly*, due to programmed interactions and synchronizations, and *implicitly*, due to in-kernel operations that may cause synchronizations among tasks that are not even directly related [12]. Those in-kernel operations are necessary because of the non-atomic nature of a sophisticated OS like Linux. For example, the highest priority thread, once activated, will not be atomically loaded in the processor, starting to run instantaneously. Instead, to notify the activation of a thread, the system needs to postpone the execution of the scheduler to avoid inconsistency in the data structures used by the scheduler. Then, interrupts must be disabled, to avoid race conditions with interrupt handlers. Hence, delays in scheduling and interrupt handling are created during activation of a thread [13]. The understanding of the synchronization primitives, and how they affect the timing behavior of a thread, is 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. This is done to ensure that such a general-purpose operating system (GPOS) behaves as efficiently as possible in the average case, while at the same time it is done to ensure that, with proper tuning, the kernel can serve an increasing number of real-time use cases as well, effectively becoming a real-time operating system (RTOS). Efforts along this line are witnessed, for example, by the progressive chasing and elimination over the years of any use of the old global kernel lock, the extensive use of fine-grain locking, the widespread adoption of memory barrier primitives, or even the postponement of most interrupt handling code to kernel threads as done in the PREEMPT_RT kernel. These are all examples of a big commitment into reducing the duration of non-preemptible kernel sections to the bare minimum, while allowing for a greater control over the priority and scheduling among the various in-kernel activities, with proper tuning by system administrators.

As a consequence, Linux runs in a satisfactory way for many real-time applications with precise timing requirements. This is possible thanks to a set of operations that ensure

the deterministic operation of Linux. The challenge is then, to describe such operations, using a level of abstraction that removes the complexity due to the in-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 [14, 15, 16, 17]. 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` [18] or `perf`[1] to trace kernel events like interrupt handling, wakeup of a new thread, context switch, etc., while `cyclictest`[2] 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 [19].

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. This article also presents a major result achieved during the validation of the presented model: three problems found in the kernel code, one regarding an inefficiency in the scheduler, another one in the tracing code itself leading to occasional loss of event data, and erroneous usage of a real-time mutex. These problems were reported to the Linux kernel community, including suggestions of fixes, with some of them already present in the current Linux code. 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 requiring reading the kernel code.

This paper constitutes an extended and consolidated version of preliminary results already published at conferences by the authors [20, 21, 22]. However, in this paper, we propose, for the first time, a complete optimized model encompassing both IRQ, NMI, and threads,

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

[2] The tool is available within the `rt-utils` software available at: `https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git`

which has been tested with a broader number of kernel versions (until the last kernel up to date). We present a deeper comparison with prior works and present details about the needed modifications to the kernel and the `perf` framework. Finally, the description on how to use the proposed technique and the results obtained through its application has been extended, including, besides the two bugs already described in earlier works, a third problem which has already been notified to kernel developers and for which we proposed another solution.

Paper Organization. The paper is organized as follows. Section 2 briefly recalls related work in the area and Section 3 provides background information about the automata theory used throughout the paper. Section 4 provides the details of the modeling strategy and discusses the development of the tracing tool used to validate the model. Section 5 includes examples of applicability of the model: first, an accurate description of the dynamics of the operations involved in the scheduling of the highest priority thread is provided; then, three examples of inconsistency between the model and (unexpected) run-time behavior are presented. Finally, Section 6 draws conclusions, while presenting possible next steps for this line of research.

2. Related Work

This section presents prior literature relevant to the work being presented in this paper, spanning across two main areas: use of automata in real-time and operating systems analysis; and formal methods applied to operating systems kernel, with a special focus on papers involving the Linux kernel.

2.1. Automata-based real-time systems analysis

Automata and discrete-event systems theory has been extensively used to verify timing properties of real-time systems. For example, in [23], a methodology based on timed discrete event systems is presented to ensure that a real-time system with multiple-period tasks is reconfigured dynamically at run-time using a safe execution sequence, under the assumption of single-processor, non-preemptive scheduling. In [24, 25, 26], the Kronos tool is used for checking properties of models based on multi-rate and parametric/symbolic timed automata.

In [27], 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. Authors extend symbolic analysis of timed automata [28], by enriching the model with parametric guards and invariant constraints, which is then checked recurring to symbolic model checking. The approach is also applied to an industrial avionic case-study [29], where verification has been carried out using the UPPAAL model checker [30]. A similar methodology can be found in [31], where parametric timed automata are used to perform sensitivity analysis in a distributed real-time system. It makes use of CAN-based

communications and fixed priority CPU scheduling, and solved with a tool called IMITATOR [32]. Similar in purposes is also the work in [33], where a technique is proposed to compute the maximum allowed imprecision on a real-time system specification, still preserving desired timing properties. Additionally, some authors [34] considered composability of automata-based timing specifications, so that timing properties of a complex real-time system can be verified with reduced complexity.

Similarly to the approach of UPPAAL [30], the TIMES tool has been used [35] with an automata-based formalism to describe a network of distributed real-time components for analyzing their temporal behavior from the viewpoint of schedulability.

The mentioned methodologies focus on modeling the timing behavior of the applications, and their reciprocal interference due to scheduling. Compared to the work being presented here, they neglect the exact sequence of steps executed by an operating system kernel, in order to let, for example, a higher-priority task preempt a lower-priority one. None of these works formalize the details of what exact steps are performed by the kernel and within its scheduler and context-switch code path. However, as it will be clarified later, these details can be fundamental to ensure the build of an accurate formal model of the possible interferences among tasks, as common in the real-time analysis literature.

Further works exist introducing mathematical frameworks for analyzing real-time systems, such as in [36, 37], making use of Hybrid and Timed Input/Output Automata to prove safety, liveness and performance properties of a system. However, these methodologies are purely mathematical and oriented towards manual reasoning and properties verification. A comprehensive literature review of these formalisms is out of scope for this paper.

2.1.1. Automata-based models for Linux

In [38], a model of an RT system involving Linux is presented, with two timing domains: a *real-time* and a *non-real-time* one. These are abstracted as a seven-state and three-state models, respectively. The model, however, is a high-level one and does not consider the internal details of the Linux kernel.

The usage of trace and automata to verify conditions in the kernel is also presented in [39]. The paper presents models for SYN-flood, escaping from a chroot jail, and, more interestingly, locking validation and real-time constraints validation. The models are compared against the kernel execution using the LTTng tracer [15]. The models presented are proof of concepts of these ideas, and are very simple: the largest model, about locking validation, has only five states. The real-time constraints model has only two states. But still, this paper corroborates the idea of the connection between automata and tracing as a translation layer from kernel to formal methods, also for problems in the real-time features of Linux.

An important area that makes use of a formal definition of the system is the State based/Stateful robustness testing [40]. Robust testing is a fault tolerance technique [41] also applied

in the OS context. In [42], a case study of state-based robustness testing including the OS states is presented. The OS under investigation is a real-time version of Linux. The results show that the OS state plays an important role in testing for corner cases not covered by traditional robustness. Another project that uses Linux is SABRINE [43], an approach for state-aware robustness testing of OSs using trace and automata. SABRINE works as follows: In the first step, it traces the interactions between OS components. Then, the software automatically extracts state models from the traces. In this phase, the traces are processed, in such a way to find sequences of functions that are similar, to be grouped, forming a pattern. Later, patterns that are similar are grouped in clusters. Finally, it generates the behavioral model from the clusters. A behavioral model consists of states connected by events, in the format of finite-state automata (FSA).

The possibility of extracting models from the operating system depends on the specification of the operating system components and their interfaces. The object of this paper is not a component of the system, but the set of mechanisms used to synchronize the operations of NMI, IRQs, and threads. The events analyzed in this paper, such as disabling interruptions and preemption, or locks, are present in most of the subsystems. Both works can be seen as complementary. The approach proposed by SABRINE can be used to define the internal models between states of the model proposed by this paper. For example, the model presented in this paper identifies the synchronization events that cause the delay in the activation of the highest priority thread. Many different code paths are taken between the event that blocks the scheduling and the event that re-enables the scheduling. SABRINE's approach can then be used to auto-generate the finite-state automata of such code paths.

The TIMEOUT approach [44] later improved SABRINE by recording the time spent in each state. The FSA is then created using timed automata. The worst case execution time observed during the profiling phase is used as the timing parameter of the Timed-FSA, and so it is also possible to detect timing faults.

2.2. 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 [45], 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*. This allows for an on-demand refinement of parts of the specification by choosing more specific predicates to add to the model while the model checker is running, without any need for revisiting parts of the state space that are not affected by the refinements. Interestingly, authors applied the technique to the verification of safety properties of OS drivers for the Linux and Microsoft Windows NT kernels. The technique required instrumentation of the original drivers, to insert a conditional jump to an error handling piece of code, and a model of the surrounding kernel behavior, in order to allow the model checker to verify whether or not the faulty code could ever be reached.

The static code analyzer SLAM [46] shares major objectives with BLAST, in that it allows for analyzing C programs to detect violation of certain conditions. It has been used also to detect improper usage of the Microsoft Windows XP kernel API by some device drivers.

Witkowski et al. [47] proposed the DDVerify tool, extending the capabilities of BLAST and SLAM, e.g., supporting synchronization constructs, interrupts and deferred tasks.

Chaki et al. [48] proposed MAGIC, a tool for automatic verification of sequential C programs against finite state machine specifications. The tool can analyze a directed 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. The verification is carried out checking the specification against a sequence of increasingly refined abstractions, until either it is verified, or a counter-example is found. This, along with its modular approach, allows the technique to be used with relatively large models avoiding the need for enumerating the state-space of the entire system. Interestingly, MAGIC has been used to verify correctness of a number of functions in the Linux kernel involved in system calls handling mutexes, sockets, and packet sending. The tool has also been extended later to handle concurrent software systems [49], albeit authors focus on verifying correctness and deadlock-freedom in the 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.

There have also been other remarkable works assessing formal correctness of a whole micro-kernel such as seL4 [50], i.e., adherence of the compiled code to its expected behavior, stated in formal mathematical terms. seL4 has also been accompanied by precise WCET analysis [51]. These findings were possible thanks to the simplicity of the seL4 micro-kernel features, e.g., semi-preemptability.

2.2.1. Formal methods in the Linux kernel community

The Linux kernel community is not new to the adoption of formal methods in the kernel development and debugging workflow. Indeed, a remarkable work in this area is the `lockdep` mechanism [52] built into the Linux kernel. Lockdep is capable of identifying errors in using locking primitives that might eventually lead to deadlocks, by observing the order of execution and the calling context of lock calls. 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. The number of different lock states that has to be kept by the kernel is reduced by applying the technique based on locking classes, rather than individual locks.

In [53], a formal memory model is introduced to automate verification of fundamental consistency properties of core kernel synchronization operations across the wide variety of supported architectures and associated memory consistency models. The memory model for

Linux ended being part of the official Linux release, with the addition of the Linux Kernel Memory Consistency Model (LKMM) subsystem, which is an array of tools that formally describe the Linux memory coherency model, and also produce 'litmus tests' in form of kernel code which can be directly executed and tested.

Moreover, the well-known TLA+ formalism [54] has been successfully applied to discover bugs in the Linux kernel. Examples of problems that were discovered or confirmed by using TLA+ goes from the correct handling of the memory management locking in the context switch to the fairness properties of the arm64 ticket spinlock implementation [55]. These recent results raised interest in the potential of the usage of formal methods in the development of Linux.

Finally, among the works that try to conjugate theoretic analytical real-time system models with empirical worst-case estimations based on a Linux OS, we can find [56]. 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 authors call scheduling overheads, then these figures are injected into overhead-aware real-time analysis techniques. The discussion about outliers in [56], along with the explicit admission of the need for removing manually some of them throughout the experimentation, witnesses the need for a more insightful model that provides more accurate information of those overheads. We aim 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. Our automata-based model, that will be detailed in the next sections, sheds some light exactly into this direction.

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 as task scheduling, IRQ and NMI management, and their delicate interplay, as done in this paper.

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

The starting point to describe a DES is the underlying set of events $E = \{e_i\}$, which represents the "alphabet" used to form "words" ("traces") of events that compose the DES language. A trace of a DES run-time behavior can be described as a sequence of the vis-

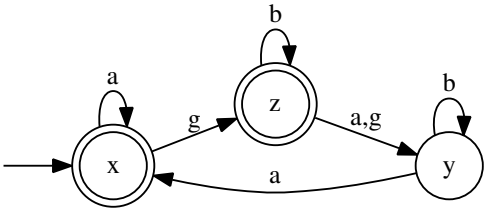

Figure 1: State transitions diagram (based on Fig. 2.1 from [57]).

ited states and the associated events causing state transitions. Hence, a DES evolution is described as a sequence of events $e_1, e_2, e_3, ...e_n$, where $e_i \in E$, with all possible sequences of events defining the language $\mathcal{L}$ that describes entirely 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, though, more flexible modeling formats were developed, being automaton one of these. Automata are characterized by 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 without source 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

$$G = \{X, E, f, \Gamma, x_0, X_m\} \tag{1}$$

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 defining $X = \{x, y, z\}$, $E = \{a, b, g\}$, $f(x, a) = x$, $f(x, g) = z$, $f(y, a) = x$, $f(y, b) = y$, $f(z, b) = z$, $f(z, a) = f(z, g) = y$, $\Gamma(x) = \{a, g\}$, $\Gamma(y) = \{a, b\}$, $\Gamma(z) = \{a, b, g\}$, $x_0 = x$ 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 between automata. An important operation is the parallel composition of two or more automata that can be combined to compose a single, augmented-state, automaton.

3.1. Operations with automata

Parallel composition allows for merging two or more automata models into one single model. The standard way of building a model of the entire system from models of individual system components is by parallel composition [57].

When modeling a system composed of interacting components, it is possible to distinguish two kinds of events, *private* events and *common* events. *Private* events belong to a single component, while *common* events are shared among components.

Given two automata $G_1 = \{X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1}\}$ and $G_2 = \{X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2}\}$, their parallel composition is defined as:

$$G_1 \parallel G_2 := Ac(X_1 \times X_2, E_1 \cup E_2, f_{1\parallel 2}, \Gamma_{1\parallel 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2}) \tag{2}$$

where $Ac()$ is the operation that trims away states that are non *accessible* from the initial state, and:

$$\begin{aligned} f_{1\parallel 2}((x_1, x_2), e) &:= \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2) & \text{if } e \in \Gamma_1(x_1) \backslash E_2 \\ (x_1, f_2(x_2, e) & \text{if } e \in \Gamma_2(x_2) \backslash E_1 \\ \text{undefined} & \text{otherwise} \end{cases} \\ \Gamma_{1\parallel 2}((x_1, x_2)) &:= \Gamma_1(x_1) \cup \Gamma_2(x_2) \end{aligned} \tag{3}$$

In words, in the parallel composition, a private event, that is an event present in only one automaton, can execute whenever possible. A common event, that is, an event in $E_1 \cap E_2$, can only be executed if it is possible in all automata that contain the event, simultaneously.

In the parallel composition product, the initial state is the initial state of all the individual automata. A state is marked if both its sub-states are marked in the composed automata.

3.2. Monolithic vs. modular modeling

In the modeling of complex systems using automata, there are two possible approaches to model a system, the *monolithic* and the *modular* approach [58].

Although this approach is suitable for simple systems, it is not efficient in the modeling of complex systems, as the number of states increases. 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* represents an independent part of the system, modeled using a pairwise disjoint set of events. The *global generator* G is then composed by the parallel composition of all sub-systems' generators. The global generator represents all chains of events that are possible for the composite system.

The *specification* of the synchronization of each sub-system is then done via a set of *specification* automata. Each *specification* synchronizes the actions of two or more *generators*, referring to their events. The parallel composition of the model of each specification composes the global specification S.

The parallel composition of the global generator G and the global specification S creates the model of the system and its synchronizations. There are many benefits in using the modular approach. In the scope of this work, the modular approach enables the analysis of important properties of Linux by observing just a set of operations, and not the global system. This advantage is explored in the analysis conducted in Section 5. The ability to compose the global model is also crucial to the validation of the system as a whole, including the coherence between the automata and the system. The global model enabled the validation of the model with the kernel in constant-time complexity, as described in Section 4.

4. 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 [18], the kernel code analysis, academic documentation about Linux and real-time systems [13] [56], and hardware documentation [59]. 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. The Linux kernel has many different preemption modes, varying from *non-preemptive*, to *fully-preemptive*. This work is based on the *fully-preemptive* mode only, that is the mode utilized by the real-time Linux community. The *fully-preemptive* mode also enables the real-time alternative for locks. For instance, it converts mutexes into real-time mutexes and read/write semaphores into real-time read/write semaphores. Moreover, in the *fully-preemptive* mode, the vast majority of the work done in the hard and soft IRQ context is moved to the thread context. The work left in the hard IRQ context is mostly related to the notification of events from hardware to the threads that will handle the request, or to decisions that cannot be delayed by the scheduler of threads. For example, the timer that

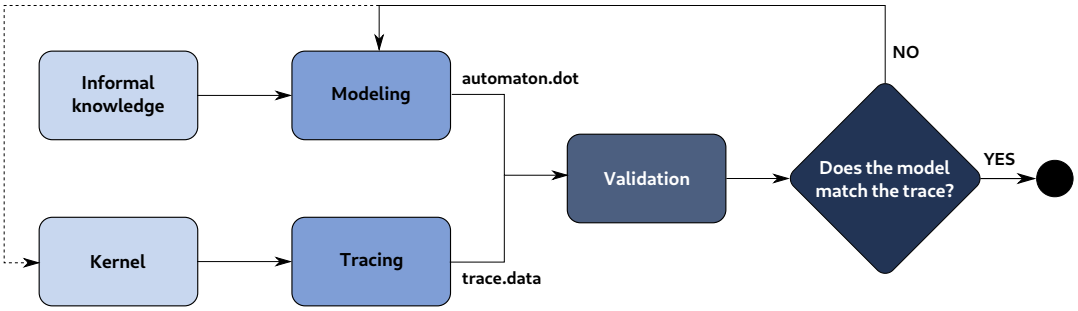

Figure 2: Modeling phases.

notifies the SCHED_DEADLINE about the throttling of a thread must run in this context to avoid being delayed by the task it aims to throttle. 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. However, the kernel was configured to run on a single CPU. The development of the model started with the version *4.14* of the PREEMPT_RT, passing by the version *4.19*. It is currently based on version *5.0*.

4.1. Events

The most important part of the modeling is the choice of events used in the model. As a computer program, Linux itself is already a model. However, it is a model with millions of lines of code and complex data structures. The difficulty is then to reduce the model to the set of events that contributes more to the purpose of the model. The level of abstraction used in this paper is the one used by real-time Linux developers while discussing the base of scheduling and synchronization problems, in the terms of real-time systems.

Linux schedules threads on the processors, but threads are not the sole execution context. In addition to the threads, interrupts are considered a distinguished execution context. Interrupts are used by external devices to notify asynchronous events. For example, a network card uses interrupts to inform of the arrival of network packets, which are handled by the driver to deliver the packet contents to an application. Linux recognizes two different kinds of interrupts: IRQs or *Maskable Interrupts* are those for which it is possible to postpone the handling by temporarily disabling them, and NMIs or *Non-Maskable interrupts*, that are those that cannot be temporarily disabled. Likewise, on Linux, the model considers these three execution contexts: Threads, IRQs, and NMI, modeling the context and the synchronization of these classes of tasks. To validate the level of abstraction and events, the model was discussed with the main real-time Linux developers at the Real-time Linux Summit 2018 [60] and The Linux Plumbers Conference 2018 [61, 62].

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 behav-

Table 1: IRQ Related Events.

Kernel event	**Automaton event**	**Description**
hw_local_irq_disable	preemptirq:irq_disable	Begin IRQ handler
hw_local_irq_enable	preemptirq:irq_enable	Return IRQ handler
local_irq_disable	preemptirq:irq_disable	Mask IRQs
local_irq_enable	preemptirq:local_irq_enable	Unmask IRQs
nmi_entry	**irq_vectors:nmi**	Begin NMI handler
nmi_exit	**irq_vectors:nmi**	Return NMI Handler

ior of the system satisfactorily. For example, although the `sched:sched_waking tracepoint` includes the `prio` field containing the priority of the just awakened thread, it 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.

Tables 1, 2 and 3 present 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. `tracepoints` in **bold font** are the ones added to the kernel during the modeling phase.

Linux kernel evolves very fast. For instance, in a very recent release (*4.17*), around 1559000 lines were changed (690000 additions, 869000 deletions) [63]. 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 over time. IRQs and NMI context, and the possibility of masking IRQs are present in Linux since its very early days. The *fully preemptive* mode, and the functions to disable preemption are present since the beginning of the PREEMPT_RT, dating back to year 2005 [64]. Moreover, 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, virtual runtime, etc.). Hence, locking and schedulers might even change, but the events and their effects in the timeline of threads stay invariable.

4.2. Modeling

The automata model was developed using the Supremica IDE [65]. 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

Table 2: Scheduling Related Events.

Kernel event	**Automaton event**	**Description**
preempt_disable	preemptirq:preempt_disable	Disable preemption
preempt_enable	preemptirq:preempt_enable	Enable preemption
preempt_disable_sched	preemptirq:preempt_disable	Disable preemption to call the scheduler
preempt_enable_sched	preemptirq: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
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
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

Table 3: Locking Related Events.

Kernel event	**Automaton event**	**Description**
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
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

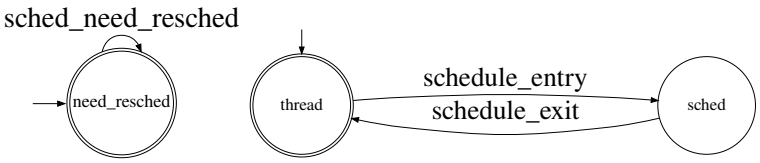

Figure 3: Examples of generators: *G05* Need Resched (left) and *G04* Scheduling Context (right).

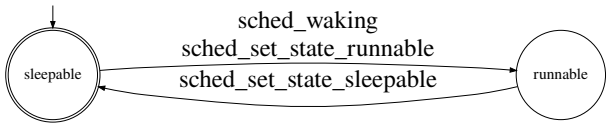

Figure 4: Examples of generators: *G01* Sleepable and Runnable.

that can be plotted using `graphviz` [66], for example.

The model was developed using the modular approach. All generators and specification were developed manually. The generators are the system's events 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 and 4. The *Need Resched* generator (*G05*) has 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 4 shows statistics about the generators and specifications that compose the Model. The complete Model is generated from the parallel composition of all generators and specifications. The parallel composition is done via the `Supremica` tool, automatically. The Model has 34 events, 9017 states, and 20103 transitions. Moreover, the Model has only one marked state, has no forbidden states, and it is deterministic and non-blocking.

The complete Model exposes the complexity of Linux. At 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 events. The complete Model, however, makes the validation of the trace more efficient, as a single automaton is validated. Hence, both the modules and the complete model are useful in the modeling and validation processes.

One frequent question made during the development of this work was: *Is it possible to auto-*

Table 4: 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	5	6
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	8	9	27
S19 Need resched forces scheduling	7	25	53
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	12
S32 Does not suspend in mutex	3	5	11
S33 Does not suspend in rw lock	3	8	16
Model	9017	34	20103

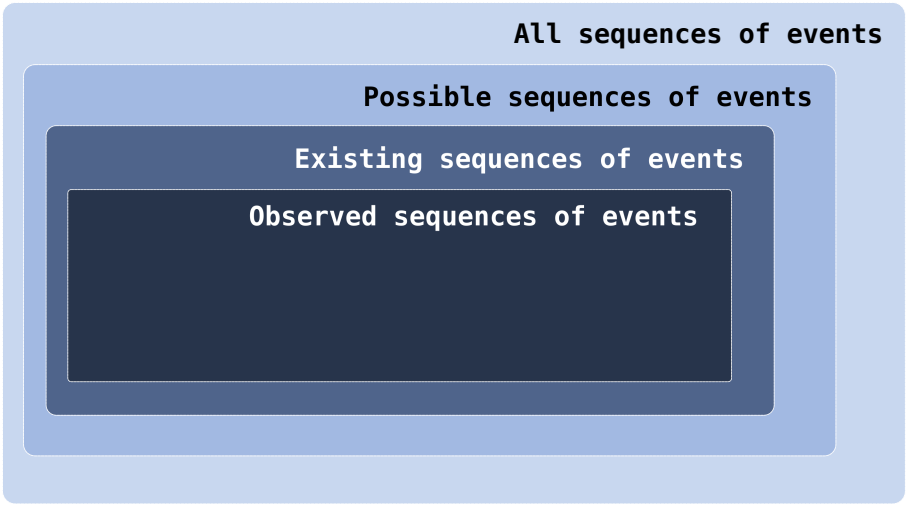

Figure 5: Sets of sequences of event.

matically create a model from the trace output? For instance, SABRINE [43] has done this before, so why not? It is certainly possible to trace the events used in this work and transform them into an automaton or a direct acyclic graph (DAG). However, some difficulties arise from such an approach.

Using Figure 5 as a reference, the approach presented in this paper starts with the outer set. The synchronization of all *generators* creates an automaton with *all* sequences of events. The synchronization of the *generators* and *specifications* reduces the set of events to those events that are possible in the system.

In the reverse order, the trace of kernel events starts from the inner set, observing the sequences of events that happened in a given system. However, there is no guarantee that *all existing* sequences of events will be *observed* in a given system, or in reasonable time. Similarly to what happens with code coverage in software testing, in order to observe a set of interesting event sequences that may possibly trigger critical conditions, the system has to be traced for a sufficiently long time, and it must be stimulated with a sufficiently diverse set of workload conditions. For example, the chances of observing NMIs occurring in all possible states they could happen are meager, given that they do not happen very often. Moreover, there are sequences of events that *do not exist* in the code but are *possible*. For instance, if the system is not idle, and the current thread disables the preemption to call the scheduler, there is no place in the code in which interrupts get intentionally disabled before calling the scheduler. This does not mean it is not a possible sequence, as long as interrupts get enabled before calling the scheduler. The kernel will not break if such a sequence appears in the future.

Hence, the refinement of the approach presented in this paper has the potential to define all the *possible sequences of events* accurately. While an automatic approach can build an

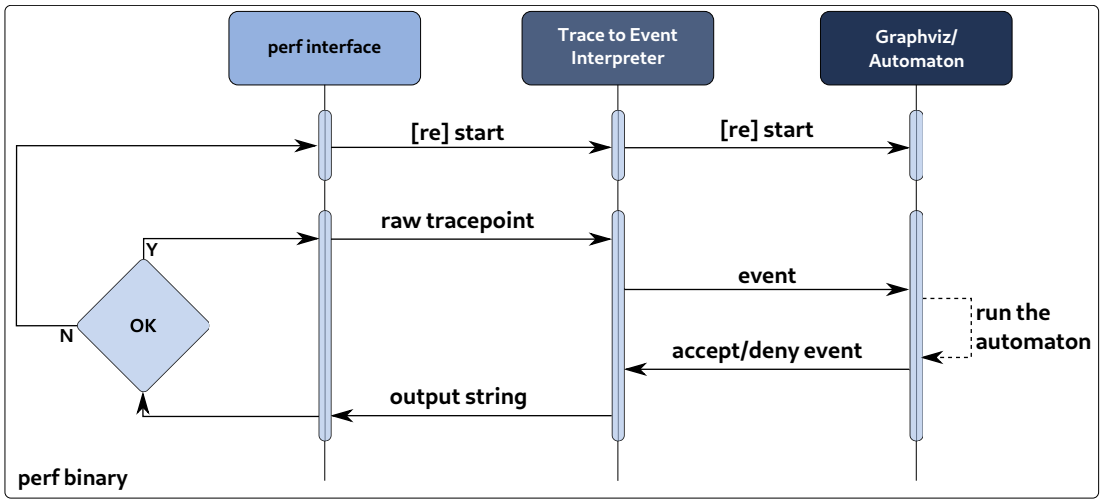

Figure 6: `perf task_model report` dynamic.

automaton with all *observed* sequences of events, the amount of time and resources required to observe all existing sequences of events is undoubtedly challenging.

The major problem, however, comes from the occurrence of an event that is not present in the model. In the modular approach, it is possible to analyze each generator and specification separately, avoiding the analysis in the global model. A hypothetical automatically generated model would require the analysis of the global automaton, which is not reasonable, given the number of states and transitions of the global model. Furthermore, in the likely presence of non-possible sequences in the kernel, the automated mode is prone to include non-possible sequences in the model.

However, these methods are complementary: The modeling approach presented in this paper was validated by observing the kernel execution. By observing the kernel events, the automaton generated by the kernel is compared against the model, as described in the next section.

4.3. 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`, and it was developed as a *built-in command*. A `perf` *built-in command* is a very efficient way to extend `perf` features: They are written in `C` and compiled in the `perf` binary. The `perf thread_model` has two operation modes: the `record` mode and the `report` mode.

The `record` mode configures the tracing session and collects the data. This phase involves both the Linux kernel tracing features and `perf` itself in the user-space. In the kernel side, `tracepoints` are enabled, recording the events in the trace buffer. Once set, `tracepoints` collect data using lock-free primitives that do not generate events considered in the model,

```
struct task_model {
        struct perf_tool        tool;
        ... other definitions ...
};
```

Figure 7: `perf_tool` definition inside the `task_model` structure.

```
struct task_model tmodel = {
        .tool = {
                .lost           = process_lost_event,
                .lost_samples   = process_lost_event,
                .sample         = process_sample_event,
                .ordered_events = true,
        },
        ... other definitions ...
};
```

Figure 8: `task_model` and *perf_tool* initialization.

not influencing in the model validation. In the user-space side, `perf` continues running, collecting the trace data from the kernel space, saving it in a `perf.data` file.

The `record` phase challenge is to deal with the high-frequency of events. A Linux system, even without a workload, generates a considerable amount of events due to housekeeping activities. For example, the periodic scheduler tick, RCU callbacks, network and disk operations, and so on. Moreover, the user-space side of `perf` generates events itself. A typical 30 seconds record of tracing of the system running `cyclictest` as workload generates around 27000000 events, amounting to 2.5 GB of data. To reduce the effect of the tracing session itself, and the loss of tracing data, a 1 GB trace buffer was allocated in the kernel, and the data was collected every 5 seconds.

After recording, the trace analysis is done using the `perf thread_model report` mode. The `report` mode has three basic arguments: 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 6. When starting, `perf` interface opens the trace file, and uses the `Graphviz` library[3] to open and parse the `.dot` file. The connection between the trace file and the automata is done in the `Trace to Event Interpreter` layer.

In the *built-in command* source, the `Trace to Event Interpreter` is implemented as a `perf_tool`. The definition of the tool is shown in Figures 7 and 8. Two callbacks implement the tool, one to handle tracing samples, and another to notify the loss of tracing events. While processing the events, `perf` calls the function `process_sample_event` for each trace entry. Another important point of the tool is that the default handler for the kernel events is substituted by a custom handler, as shown in Figure 9.

The `process_sample_event` waits for the initial condition of the automaton to be reached in the trace. After the initial condition is met, the callback functions start to be called. Figure 10 shows an example of a `tracepoint` callback handler. The `tracepoint` handlers translate the `raw trace` to an *event* string used in the model.

The `process_event` function, in Figure 11, is used to run the automaton. If the automaton

[3]More information is available at: `http://graphviz.org/`.

```
	const struct perf_evsel_str_handler model_tracepoints[] = {

		{ "irq_vectors:nmi_exit",			process_nmi_exit		},
			/* nmi_entry should be the last for NMI */
		{ "irq_vectors:nmi_entry",			process_nmi_entry		},
		{ "irq_vectors:move_cleanup_exit",		process_int_exit		},
		{ "irq_vectors:move_cleanup_entry",		process_int_entry		},
		... lines omitted ...
		{ "preemptirq:preempt_disable",			process_thread_preempt_disable	},
		{ "preemptirq:preempt_enable",			process_thread_preempt_enable	},
		{ "sched:sched_entry",				process_thread_sched_entry	},
			/* sched_exit should be the last for THREAD */
		{ "sched:sched_exit",				process_thread_sched_exit	},
	};
```

Figure 9: `perf thread_model`: Events to callback mapping.

```
static int process_nmi_entry(struct task_model *tmodel,
			     struct perf_evsel *evsel __maybe_unused,
			     struct perf_sample *sample)
{
	const char *event = "nmi_entry";
	struct cpu *c = cpu_of_system(&tmodel->system, sample->cpu);

	c->in_nmi = 1;
	process_event(tmodel, sample, event);

	return 0;
}
```

Figure 10: Handler for the `irq_vectors:nmi_entry tracepoint`.

accepts the event, the regular output is printed. Otherwise, an error message is printed, the tool resets the automaton and discards upcoming events in the trace until the initial condition of the automaton is recognized again. Finally, because of the high-frequency of events, it might be the case that the trace buffer discards some events, causing the loss of synchronization between the trace and the automaton. When an event loss is detected, `perf` is instructed to call the function `process_lost_event` (see Figure 8), notifying the user, and resetting the model. Either way, the trace continues to be parsed and evaluated until the end of the trace file.

The model 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 the events is done in constant 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 12.

When in a given state, if the kernel event is not possible in the automaton, the tool prints an error message. It is then possible to use the `Supremica` simulation mode to identify the state of the automaton, and the raw trace to determine the events generated by the kernel.

```
static int process_event(struct task_model *tmodel, struct perf_sample *sample,
                  const char *event)
{
        int retval;

        event_print_before(tmodel, sample, event);
        retval = automaton_event(tmodel->automaton, event);
        if (!retval) {
                event_not_expected(tmodel->automaton, event, sample);
                reset_model(tmodel);
        }
        event_print_after(tmodel, sample, event);
        return retval;
}
```

Figure 11: *process_event*: Trying to run the automata.

```
 1: Reference model: thread.dot
 2: +---->  +=thread of interest - .=other threads
 3: | +-> T=Thread - I=IRQ - N=NMI
 4: | |
 5: | |     TID |   timestamp   | cpu |           event               | state   | safe?
 6: [...]
 7: . T       8    436.912532    [000]              preempt_enable ->       q0 safe
 8: . T       8    436.912534    [000]            local_irq_disable ->    q8102
 9: . T       8    436.912535    [000]              preempt_disable ->   q19421
10: . T       8    436.912535    [000]                 sched_waking ->      q99
12: . T       8    436.912535    [000]           sched_need_resched ->   q14076
13: . T       8    436.912535    [000]             local_irq_enable ->    q1965
14: . T       8    436.912536    [000]              preempt_enable ->   q12256
15: . T       8    436.912536    [000]        preempt_disable_sched -> q18615,q23376
16: . T       8    436.912536    [000]               schedule_entry -> q16926,q17108,q2649,q7400
17: . T       8    436.912537    [000]            local_irq_disable -> q11700,q14046,q21391,q23792
18: . T       8    436.912537    [000]           sched_switch_out_o -> q10337,q20018,q21933,q7672
19: . T       8    436.912537    [000]              sched_switch_in -> q10268,q20126
20: + T    1840    436.912537    [000]             local_irq_enable ->   q20036
21: + T    1840    436.912538    [000]                schedule_exit ->   q21033
22: + T    1840    436.912538    [000]         preempt_enable_sched ->    q4303
```

Figure 12: Example of the `perf thread_model` output: a thread activation.

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 the `perf` tool. Indeed, during the development of the model, three problems were reported to the Linux community. More details will follow in Section 5.2. 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 reproduce the experiments are available at this paper's *Companion Page* [67].

5. Applications of the Model

The model has manifold applications. This section presents two of them: 1) usage of the model to describe the behavior of the kernel (Section 5.1); and 2) the usage of the model

for runtime verification [68] (Section 5.2), including discussion of the problems that have already been found in the Linux kernel with the proposed methodology and reported to the community.

5.1. Using the model to understand kernel's dynamics

This section analyzes part of the kernel events 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 analysis is done based on the model, not the kernel code.

The generators that act during the activation of a thread are described first, then the specifications that synchronize the generators are presented. Next, specifications and generators are used to explain the possible paths taken during the execution, and how they influence the activation delay.

5.1.1. Generators

The model considers three execution contexts: 1) NMI; 2) IRQs and 3) Threads, referred to as *tasks* in what follows. The generator *G09* in Figure 13 shows 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 [20]. In order 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 unmasking 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()` and similar functions. The generator *G07* models the masking of interrupts by a thread. The generator *G08* models the masking of interrupts by the hardware to handle a hardware interrupt. These generators are presented in Figure 14.

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 4).

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

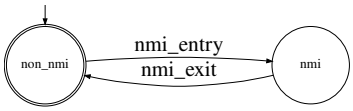

Figure 13: *G09* NMI generator.

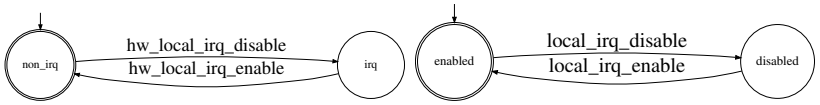

Figure 14: *G08* IRQ Handling (left); *G07* IRQ Masking (right) generators.

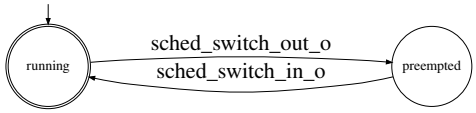

Figure 15: *G03* Context switch other thread generator.

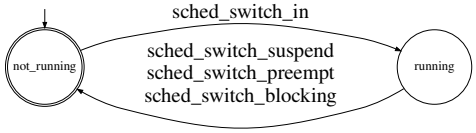

Figure 16: *G02* Context switch generator.

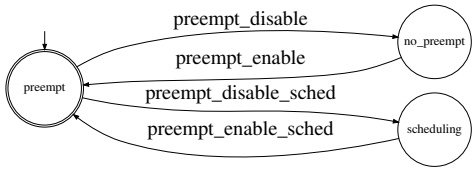

Figure 17: *G06* Preempt disable.

the idle state runs as if it was a thread, the lowest priority thread. It is named `swapper` and has the *pid* 0. 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 15.

The context switch generator for the thread under analysis is slightly different. In the initial state, the thread is not running. After it starts 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 16.

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

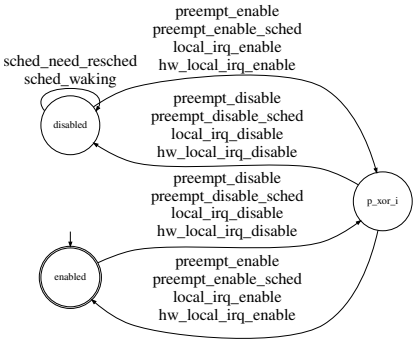

Figure 18: *S02* Wakeup and Need resched takes place with IRQs and preemption disabled.

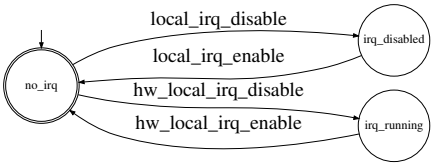

Figure 19: *S17* IRQ disabled.

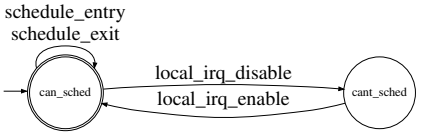

Figure 20: *S05* Scheduler called with interrupts enabled.

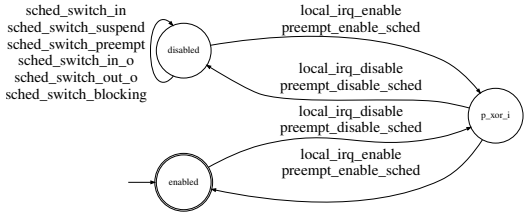

Figure 21: *S07* Switch with interrupts and preempt disabled.

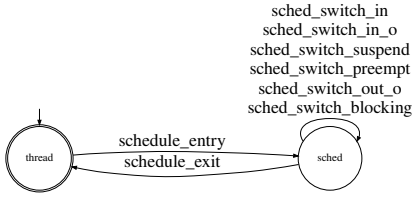

Figure 22: *S08* Switch while scheduling.

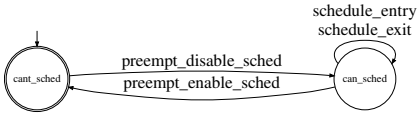

Figure 23: *S03* Scheduler called with preemption disabled.

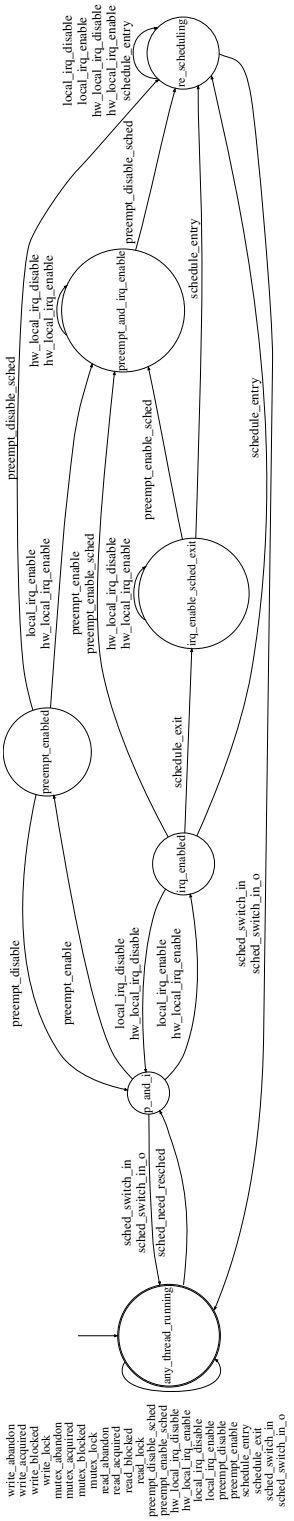

Figure 24: *S19* Need resched forces scheduling.

```
1: . T       8    271158.409224    [000]              preempt_enable ->       q0 safe
2: . T       8    271158.409224    [000]       preempt_disable_sched ->    q7169
3: . I       8    271158.410164    [000]        hw_local_irq_disable ->   q23201
4: . I       8    271158.410166    [000]          sched_need_resched ->    q2204
5: . I       8    271158.410167    [000]         hw_local_irq_enable ->      q68
6: . T       8    271158.410168    [000]              schedule_entry ->   q12032
7: . T       8    271158.410169    [000]           local_irq_disable ->   q15580
8: . T       8    271158.410169    [000]          sched_switch_out_o ->   q19424
9: . T       8    271158.410169    [000]           sched_switch_in_o ->   q16186
```

Figure 25: Need resched when already on the way to schedule.

events are generated, the second case generates the events `preempt_disable_sched` and `preempt_enable_sched`. These two possibilities are modeled in the generator *G06*, as shown in Figure 17.

5.1.2. Specification

In Figure 18, the specification *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. The specification works as follows: in the initial state (*enabled*) preemption and interrupts are enabled. In this state `sched_waking` and `sched_need_resched` are not possible. After disabling either preemption or interrupts, the automaton moves to the state *p_xor_i*. In this state, the events are still not possible. Then, after disabling both, 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 19. 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 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 IRQs 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 21 and 22, respectively. The scheduler execution has two main specifications as well: the specification *S03*, in Figure 23, 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 20.

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 24, 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 the context switch in of 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 to the initial state can happen through a sole state, the *re_scheduling*. From the state *p_and_i* until *re_scheduling*, 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*). For instance through the chain of events set_need_resched, hw_local_irq_enable, schedule_entry bring the specification *S17* to the re_scheduling state, forcing the scheduler, as exemplified in the lines 4, 5 and 6 in Figure 25. 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.

```
 1:   ktimersoftd/0     8 [000]   784.425631:           sched:sched_switch: ktimersoftd/0:8 [120] R ==> kworker/0:2:728 [120]
 2:     kworker/0:2   728 [000]   784.425926:        sched:sched_set_state: sleepable
 3:     kworker/0:2   728 [000]   784.425936:       sched:set_need_resched: comm=kworker/0:2 pid=728
 4:     kworker/0:2   728 [000]   784.425939:  sched:sched_preempt_disable: at ___preempt_schedule <- ___preempt_schedule
 5:     kworker/0:2   728 [000]   784.425941:            sched:sched_entry: at preempt_schedule_common
 6:     kworker/0:2   728 [000]   784.425945:           sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:1:724 [120]
 7: irq/14-ata_piix    86 [000]   784.426515:           sched:sched_waking: comm=kworker/0:2 pid=728 prio=120 target_cpu=000
 8:     kworker/0:1   724 [000]   784.426610:           sched:sched_switch: kworker/0:1:724 [120] t ==> kworker/0:2:728 [120]
 9:     kworker/0:2   728 [000]   784.426615:  sched:sched_preempt_disable: at schedule <- schedule
10:     kworker/0:2   728 [000]   784.426616:            sched:sched_entry: at schedule
11:     kworker/0:2   728 [000]   784.426619:           sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:2:728 [120]
```

Figure 26: Kernel trace excerpt.

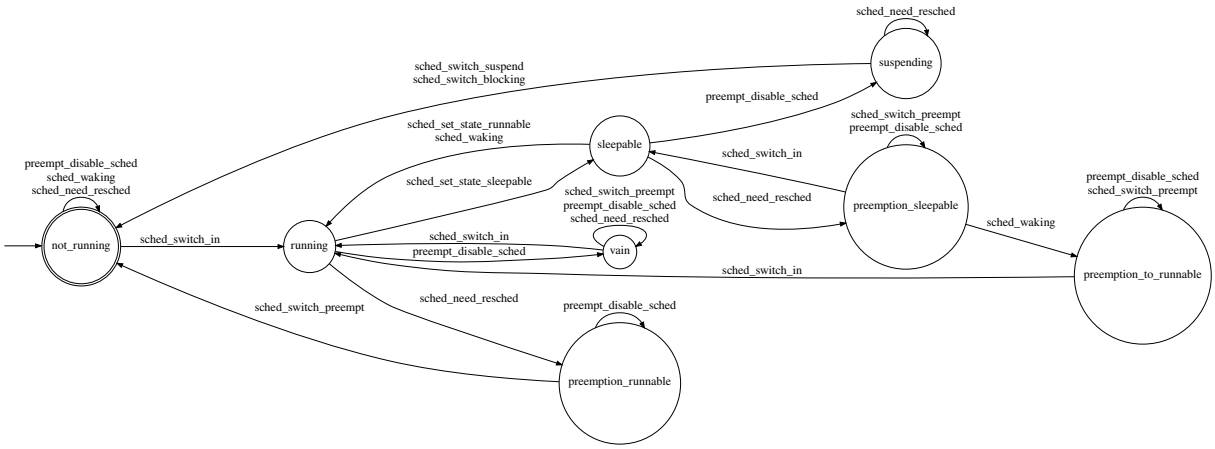

Figure 27: *S18* Scheduler call sufficient and necessary conditions.

5.2. Kernel problems outlined by the automaton

This section presents three problems found in the kernel while validating the model. The first is an optimization case, the second is a problem in the tracing, and the third is regarding an invalid usage of real-time mutex in an interrupt handler.

5.2.1. Scheduling in vain

In Linux, the main scheduler function (`__schedule()`) is always called with preemption disabled, as shown in Figure 23. In the model, it can be seen as the event that precedes a scheduler call. The specification in Figure 27 presents the conditions for the thread under analysis to disable preemption to call the scheduler. In the initial state, in which the thread is not running, the `preempt_disable_sched` event is recognized, because other threads can indeed schedule. The `sched_switch_in` switches the state of the thread to `running`. The `running` state recognizes three events, the `sched_set_state_sleepable`, the `sched_need_resched`, and the `preempt_disable_sched`. In the case of the occurrence of the event `sched_set_state_sleepable`, the thread changes the state to `sleepable`, where the `preempt_disable_sched` is recognized as well. In these states, the *sufficient* conditions to call the scheduler exist. However, in the `sleepable` state, the thread can return to the previous state with the occurrence of the event `sched_set_state_runnable`, and so the scheduler will not *necessarily* be called.

Table 5: Events and state transitions of Figure 26.

Line	Event	New state
1	sched_switch_in	running
2	sched_set_state_sleepable	sleepable
3	sched_need_resched	preemption_sleepable
4	preempt_disable_sched	preemption_sleepable
6	sched_switch_preempt	preemption_sleepable
7	sched_waking	preemption_to_runnable
8	sched_switch_in	running
9	preempt_disable_sched	vain

In the sleepable state, in the case of the occurrence of the event sched_need_resched, the preempt_disable_sched will become possible, moving the thread to the state preemption_runnable. In this state, though, it is not possible to return to the running state without a sched_switch_in event, meaning that a preemption will occur. As the preemption only occurs in the scheduling context, the sched_need_resched event is both a *necessary* and a *sufficient* condition to call the scheduler.

In the running state, it is already possible to call the scheduler, bringing to a state named vain, which is a special case. Taking the trace of Figure 26, considering the thread kworker/0:2 in analysis, and the model in Figure 27 in the initial state, the events and state transitions of Table 5 take place.

The thread kworker/0:2 started to run at Line 1. From the running state, it sets its state to sleepable in Line 2, followed by the need_resched event in Line 3, causing the preemption to be disabled in Line 4, to call the scheduler in Line 5. Then, the thread switched the context in preemption and left the processor. At Line 7 the thread is awakened, switching the state to preemption_to_runnable. At Line 8 the context_switch_in takes place, and the thread starts to run. However, right after returning from the scheduler function, the thread disables the preemption to call the scheduler again at Line 9 and 10, calling the scheduler in vain state. In fact, as shown in Figure 26, the call to the scheduler was in *vain*, at Line 11, as no real context switch takes place.

In a deeper analysis, before calling __schedule() to cause a context switch, the schedule() function runs sched_submit_work() to dispatch deferred work that was postponed to the point that the thread is leaving the processor voluntarily, as an optimization. The optimization, however, caused a preemption, that caused the scheduler to be called in the path to call the scheduler. Hence, calling the scheduler twice. Calling the scheduler twice does not cause a logical problem. But it causes the strange effect of calling the scheduler in vain, doubling the scheduler overhead.

This behavior was reported to the Linux community, along with a suggestion of fix. The suggestion was submitted to the real-time Linux kernel development list, and it was accepted for mainline integration [69].

```
. T     419    361931.701759    [000]               preempt_enable ->       q0 safe
. T     419    361931.701761    [000]              preempt_disable ->   q17630
. T     419    361931.701761    [000]               preempt_enable ->       q0 safe
. T     419    361931.701762    [000]                 sched_waking
        361931.701762 event sched_waking is not expected in state q0
```

Figure 28: Missing kernel events: The output of `perf thread_model`.

```
xfsaild/dm-0 419 [000] 361931.701761: sched:sched_preempt_disable: at cfq_remove_request <- cfq_remove_request
xfsaild/dm-0 419 [000] 361931.701761:  sched:sched_preempt_enable: at cfq_remove_request <- cfq_remove_request
xfsaild/dm-0 419 [000] 361931.701762:          sched:sched_waking: comm=irq/30-megasas pid=311 prio=49 target_cpu=000
```

Figure 29: Missing kernel events: The output of kernel `tracepoints`.

5.2.2. Tracing dropping events

During the validation phase, sometimes, the output of the `perf thread_model` pointed to an error in the conditions in which either the `sched_waking` or `sched_need_resched` events happen, like in Figure 28.

Both mentioned events require the preemption and IRQs to be disabled, as modeled in Figure 18, which raised the attention for a possible problem in the kernel. While analyzing the problem, it was noticed that the thing in common with all the occurrences of these errors was that they took place in the *wakeup* of threads that are generally awakened by interrupts. For instance, the trace in Figure 29 shows the raw trace from kernel for the case evaluated in Figure 28, in which the thread that handles the IRQ of an HDD controller was being awakened.

By checking the kernel code, it is possible to see that the wakeup of a thread and the setting of *need resched* flag always occur with the `rq_lock` taken, and this ensures that both IRQs and preemption are disabled. Also, checking with the `ftrace` function tracer, it was possible to observe that interrupts and preemption were always disabled on the occurrence of the `sched_waking` or `sched_need_resched` events by checking the flags of the events.

A bug report was sent to the Linux kernel developers [70]. The problems turned out to be in the tracing recursion control.

Despite being lock-free and lightweight, tracing operations are not atomic, requiring the execution of functions to register the trace into the trace-buffer, as in the pseudo-code in Figure 30.

Many kernel functions are set as non-traceable, avoiding this problem. However, setting functions as non-traceable might not always be desirable, as some of these functions may be of interest for the developer in other call sites. To overcome this problem, the trace subsystem uses a *context-aware recursive lock*. When the trace function is called, it will try to take the lock. Considering the execution of a thread, if the lock was not taken, the trace function will proceed normally. If the lock was already taken, the trace function returns

```
a_kernel_function() {
    trace_function() {
        func_used_by_trace() {
            trace_function() {

                /* Trace Recursion */
```

Figure 30: Pseudo-code of tracing recurrence.

```
0)   ==========> |
0)               |  do_IRQ() { /* First C function */
0)               |    irq_enter() {
0)               |       /* set the IRQ context. */
0)   1.081 us    |    }
0)               |    handle_irq() {
0)               |      /* IRQ handling code */
0) + 10.290 us   |    }
0)               |    irq_exit() {
0)               |       /* unset the IRQ context. */
0)   6.657 us    |    }
0) + 18.995 us   |  }
0)   <========== |
```

Figure 31: Trace excerpt with comments of where the IRQ context is identified in the trace.

without tracing, avoiding the recursion problem.

However, the recursion is allowed for the case of a task in another context. For example, if a thread owns the lock when an IRQ takes place, it is desired that the IRQ can take the recursive lock to trace its execution. Likewise for NMIs. Hence, the recursive lock avoids recursion of the trace in the same task context, but not on a different task context.

The *context-aware recursive lock* works correctly. The problem is that the variable with information about the task context is set after the execution of the first functions of the IRQ and NMI handlers, as in Figure 31. Hence, if an interrupt takes place during the recording of a trace entry, the function `do_IRQ()` will be detected as a recursion in the trace, and will not be registered, likewise, the `tracepoints` that take place before the operation that sets the current context to the IRQ context.

The solution for this bug requires modification in the detection of the current context by the tracing sub-system. A proof-of-concept patch fixing this problem was proposed by the authors to the Linux kernel developers [71]. It involves detecting the current task context before executing any `C`.

5.2.3. Using a real-time mutex in an interrupt handler

While validating the model against the *4.19-rt* kernel version, the unexpected event in Figure 32 took place. In words, a `mutex_lock` operation was tried with interrupts disabled, to handle an IRQ.

This operation is not expected, due to the specifications *S12* and *S22*, as in Figures 33 and 34. The raw trace showed that a real-time mutex was being taken in the timer interrupt, as shown in Figure 35. The interrupt in case was the timer interrupt, while running the watchdog timer. Figure 36 shows the stack of functions, from the interrupt to the mutex.

This BUG was the first regression found with the model. The model was first built and

```
+ T   32019    2564.541340    [000]             preempt_disable ->    q8250
+ T   32019    2564.541342    [000]           local_irq_enable ->   q13544
+ I   32019    2564.541344    [000]       hw_local_irq_disable ->   q18001
+ I   32019    2564.541345    [000]                  mutex_lock
       2564.541345 event mutex_lock is not expected in state q18001
===== reseting model =====
```

Figure 32: `mutex_lock` not permitted with interrupts disabled.

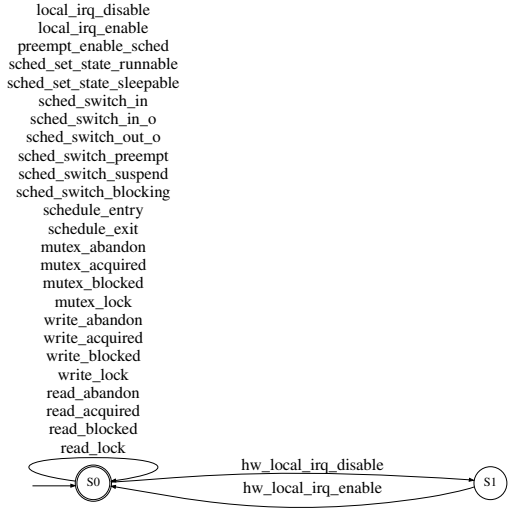

Figure 33: *S12* Events blocked in the IRQ context.

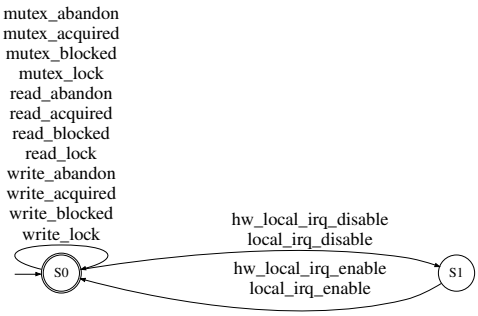

Figure 34: *S22* Lock while interruptible.

verified against the *4.14-rt* kernel [22], and this problem was not present. A change in the `watchdog` behavior added this problem: previously, the watchdog used to run as a dedicated per-cpu thread, awakened by the timer interrupt. This thread used to run with the highest `FIFO` priority. With the addition of the `SCHED_DEADLINE`, the `watchdog` thread started to be postponed by the threads running in the `SCHED_DEADLINE`. To overcome this limitation, the watchdog was moved to the `stop_machine` context, which runs with a priority higher than the `SCHED_DEADLINE`. The problem, though, is that the queue of work in the `stop_machine` uses mutexes. The patch that caused the problem was included in the kernel version *4.19*. The bug was reported to the kernel developers [72].

```
1: bash 32019 [000]  2564.541340:    preemptirq:preempt_disable: caller=__up_write+0x36 parent=__up_write+0x36
2: bash 32019 [000]  2564.541342:        preemptirq:irq_enable: caller=__up_write_unlock+0x75 parent=(nil)F
3: bash 32019 [000]  2564.541344:       preemptirq:irq_disable: caller=trace_hardirqs_off_thunk+0x1a parent=interrupt_entry+0xda
4: bash 32019 [000]  2564.541344: irq_vectors:local_timer_entry: vector=236
5: bash 32019 [000]  2564.541345:         lock:rt_mutex_request: pendingb_lock+0x0 queue_work_on+0x41
```

Figure 35: Trace of `mutex_lock` taken in the timer interrupt handler.

```
smp_apic_timer_interrupt(){
    hrtimer_interrupt() {
        __hrtimer_run_queues() {
            watchdog_timer_fn() {
                stop_one_cpu_nowait() {
                    #ifdef !CONFIG_SMP
                        schedule_work() {
                            queue_work() {
                                queue_work_on() {
                                    local_lock_irqsave() {
                                        __local_lock_irqsave() {
                                            __local_lock_irq() {
                                                spin_lock_irqsave() {
                                                    rt_spin_lock()    {
                                                        mutex_lock()    {
```

Figure 36: Function stack, from the timer IRQ to the `mutex_lock`, used in the report for the Linux kernel developers.

6. Conclusions

Linux is a complex operating system, where common assumptions like that the scheduling operation is atomic do not hold. The need for synchronization between the various task contexts, like threads, IRQs and NMIs; the scheduling operation that cannot re-entry, the lock nesting needed in the lock implementation, add a level of complexity that cannot be avoided for the correct development of theoretical work that aims Linux. 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. For example, the explanation presented in Section 5 used only a set of specifications and not all of the models. 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". The problems found later in the kernel, mainly in the trace, endorse the manual modeling: an automatically generated model from traces, albeit interesting, would potentially include errors induced by possible problems in the kernel.

As regarding future work on this line of research, the natural continuation of the presented work is the modeling of the multiprocessor behavior of Linux, including busy-wait locks and migration restrictions to the model.

Although the authors expected that the usage of the presented model could help in the debugging of Linux in the future, the fact that the model produced practical results during the development was a pleasant surprise. The usage of the automata for the verification of Linux is indeed a point that deserves further research and development, mainly focusing on a more efficient runtime verification method, that could be done entirely in the kernel, like `lockdep` does.

Another potential usage of the model is to define other metrics for the evaluation of the PREEMPT_RT. The model sheds light in the internal operations that cause "*latencies*" in the PREEMPT_RT, and the understanding of these operations can be used to turn the *black box* test into a more precise set of metrics, closing the gap between real-time Linux and real-time theory. Moreover, the model can be used to justify adaptations in well known real-time schedulers, to add the states of Linux to their analytical models.

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 approach should be integrated in the kernel code, mainly to improve testing of the logical correctness of the kernel [62], but also for timing regressions, with the creation of new metrics for the PREEMPT_RT kernel [73]. Further improvements in the tooling should be done to arrive at such goal, for instance by improving the performance of the tracing by using `eBPF`. The approach has also potential to be used in other areas of the kernel, by the modeling of other components.

References

[1] V. Vardhan, W. Yuan, A. F. H. III, S. V. Adve, R. Kravets, K. Nahrstedt, D. G. Sachs, D. L. Jones, GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy, IJES 4 (2) (2009) 152–169 (2009). doi:10.1504/IJES.2009.027939.
URL `https://doi.org/10.1504/IJES.2009.027939`

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

[3] A. Dubey, G. Karsai, 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, 2009, pp. 55–62 (March 2009). doi:10.1109/ISORC.2009.28.

[4] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, F. Rusina, A real-time service-oriented architecture for industrial automation, IEEE Transactions on Industrial Informatics 5 (3) (2009) 267–277 (Aug 2009). doi:10.1109/TII.2009.2027013.

[5] J. Condliffe. U.S. Military Drones Are Going to Start Running on Linux [online] (Jul. 2014).
URL: `https://gizmodo.com/u-s-military-drones-are-going-to-start-running-on-linu-1572853572`.

[6] J. Corbet. Linux at NASDAQ OMX [online] (Oct. 2010).
URL: `https://lwn.net/Articles/411064/`.

[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] B. B. Brandenburg, J. H. Anderson, Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors, in: 19th Euromicro Conference on Real-Time Systems (ECRTS'07), 2007, pp. 61–70 (July 2007). doi:10.1109/ECRTS.2007.17.

[9] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, J. H. Anderson, Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers, in: Proceedings of the 27th IEEE International Real-Time Systems Symposium, RTSS '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 111–126 (2006). doi:10.1109/RTSS.2006.27.

[10] T. Gleixner, Realtime Linux: academia v. reality, Linux Weekly News (July 2010).
URL https://lwn.net/Articles/397422/

[11] B. Brandenbug, J. Anderson, Joint Opportunities for Real-Time Linux and Real-Time System Research, in: Proceedings of the 11th Real-Time Linux Workshop (RTLWS 2009), 2009, pp. 19–30 (Sept 2009).

[12] F. Cerqueira, B. Brandenburg, A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS-RT, in: Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications, 2013, pp. 19–29 (2013).

[13] D. B. de Oliveira, R. S. de Oliveira, Timing analysis of the PREEMPT_RT Linux kernel, Softw., Pract. Exper. 46 (6) (2016) 789–819 (2016). doi:10.1002/spe.2333.

[14] S. Rostedt. Using kernelshark to analyze the real-time scheduler [online] (February 2011).
URL: https://lwn.net/Articles/425583/.

[15] A. Spear, M. Levy, M. Desnoyers, Using tracing to solve the multicore system debug problem, Computer 45 (12) (2012) 60–64 (Dec 2012). doi:10.1109/MC.2012.191.

[16] D. Toupin, Using tracing to diagnose or monitor systems, IEEE Software 28 (1) (2011) 87–91 (Jan 2011). doi:10.1109/MS.2011.20.

[17] B. B. Brandenburg, J. H. Anderson, Feather-trace: A light-weight event tracing toolkit, in: Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications (OSPERT'07), 2007, pp. 61–70 (2007).

[18] S. Rostedt, Secrets of the Ftrace function tracer, Linux Weekly NewsAvailable at: http://lwn.net/Articles/370423/ [last accessed 09 May 2017] (January 2010).

[19] D. B. de Oliveira, D. Casini, R. S. de Oliveira, T. Cucinotta, A. Biondi, G. Buttazzo, Nested Locks in the Lock Implementation: The Real-Time Read-Write Semaphores on Linux, in: Proc. of the 9th International Real-Time Scheduling Open Problems Seminar (RTSOPS 2018), in conjunction with the 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), Barcelona, Spain, 2018 (July 2018).

[20] D. B. de Oliveira, R. S. de Oliveira, T. Cucinotta, 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), 2017, pp. 1–8 (Sept 2017). doi:10.1109/ETFA.2017.8247611.

[21] D. B. de Oliveira, T. Cucinotta, 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, 2018 (October 2018).

[22] D. B. de Oliveira, T. Cucinotta, R. S. de Oliveira, Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel, in: Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), Valencia, Spain, 2019 (May 2019).

[23] X. Wang, Z. Li, W. M. Wonham, Dynamic Multiple-Period Reconfiguration of Real-Time Scheduling Based on Timed DES Supervisory Control, IEEE Transactions on Industrial Informatics 12 (1) (2016) 101–111 (Feb 2016). doi:10.1109/TII.2015.2500161.

[24] S. Yovine, Kronos: A verification tool for real-time systems, International Journal on Software Tools for Technology Transfer 1 (1-2) (1997) 123–133 (1997).

[25] A. Bouajjani, S. Tripakis, S. Yovine, On-the-fly symbolic model checking for real-time systems, in: Proceedings Real-Time Systems Symposium, 1997, pp. 25–34 (Dec 1997). doi:10.1109/REAL.1997.641266.

[26] C. Daws, S. Yovine, Two examples of verification of multirate timed automata with Kronos, in: Proceedings 16th IEEE Real-Time Systems Symposium, 1995, pp. 66–75 (Dec 1995). doi:10.1109/REAL.1995.495197.

[27] A. Cimatti, L. Palopoli, Y. Ramadian, Symbolic Computation of Schedulability Regions Using Parametric Timed Automata, in: 2008 Real-Time Systems Symposium, 2008, pp. 80–89 (Nov 2008). doi:10.1109/RTSS.2008.36.

[28] E. Fersman, P. Pettersson, W. Yi, Timed automata with asynchronous processes: Schedulability and decidability, in: J.-P. Katoen, P. Stevens (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 67–82 (2002).

[29] T. T. H. Le, L. Palopoli, R. Passerone, Y. Ramadian, Timed-automata based schedulability analysis for distributed firm real-time systems: a case study, International

Journal on Software Tools for Technology Transfer 15 (3) (2013) 211–228 (Jun 2013). doi:10.1007/s10009-012-0245-y.
URL https://doi.org/10.1007/s10009-012-0245-y

[30] P. Li, B. Ravindran, S. Suhaib, S. Feizabadi, A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems, IEEE Transactions on Software Engineering 30 (9) (2004) 613–629 (Sept 2004). doi:10.1109/TSE.2004.45.

[31] Y. Sun, R. Soulat, G. Lipari, É. André, L. Fribourg, Parametric schedulability analysis of fixed priority real-time distributed systems, in: C. Artho, P. C. Ölveczky (Eds.), Formal Techniques for Safety-Critical Systems, Springer International Publishing, Cham, 2014, pp. 212–228 (2014).

[32] É. André, L. Fribourg, U. Kühne, R. Soulat, IMITATOR 2.5: A tool for analyzing robustness in scheduling problems, in: D. Giannakopoulou, D. Méry (Eds.), Proceedings of the 18th International Symposium on Formal Methods (FM'12), Vol. 7436 of Lecture Notes in Computer Science, Springer, Paris, France, 2012, pp. 33–36 (Aug. 2012).
URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/AFKS-fm12.pdf

[33] K. G. Larsen, A. Legay, L.-M. Traonouez, A. Wasowski, Robust synthesis for real-time systems, Theoretical Computer Science 515 (2014) 96 – 122 (2014). doi:https://doi.org/10.1016/j.tcs.2013.08.015.
URL http://www.sciencedirect.com/science/article/pii/S0304397513006397

[34] K. Lampka, S. Perathoner, L. Thiele, Component-based system design: analytic real-time interfaces for state-based component implementations, International Journal on Software Tools for Technology Transfer 15 (3) (2013) 155–170 (Jun 2013). doi:10.1007/s10009-012-0257-7.
URL https://doi.org/10.1007/s10009-012-0257-7

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

[36] D. K. Kaynar, N. Lynch, R. Segala, F. Vaandrager, Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems, in: RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003, 2003, pp. 166–177 (Dec 2003). doi:10.1109/REAL.2003.1253264.

[37] N. Lynch, R. Segala, F. Vaandrager, Hybrid I/O automata, Information and Computation 185 (1) (2003) 105 – 157 (2003). doi:https://doi.org/10.1016/S0890-5401(03)00067-1.
URL http://www.sciencedirect.com/science/article/pii/S0890540103000671

[38] H. Posadas, E. Villar, D. Ragot, 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, 2010, pp. 238–244 (May 2010). doi:10.1109/ISORC.2010.18.

[39] G. Matni, M. Dagenais, Automata-based approach for kernel trace analysis, in: 2009 Canadian Conference on Electrical and Computer Engineering, 2009, pp. 970–973 (May 2009). doi:10.1109/CCECE.2009.5090273.

[40] B. Lei, Z. Liu, C. Morisset, X. Li, State based robustness testing for components, Electron. Notes Theor. Comput. Sci. 260 (2010) 173–188 (Jan. 2010). doi:10.1016/j.entcs.2009.12.037.
URL http://dx.doi.org/10.1016/j.entcs.2009.12.037

[41] L. L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, Inc., Norwood, MA, USA, 2001 (2001).

[42] D. Cotroneo, D. Di Leo, R. Natella, R. Pietrantuono, A case study on state-based robustness testing of an operating system for the avionic domain, in: F. Flammini, S. Bologna, V. Vittorini (Eds.), Computer Safety, Reliability, and Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 213–227 (2011).

[43] D. Cotroneo, D. D. Leo, F. Fucci, R. Natella, Sabrine: State-based robustness testing of operating systems, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 125–135 (2013). doi:10.1109/ASE.2013.6693073.
URL https://doi.org/10.1109/ASE.2013.6693073

[44] R. Shahpasand, Y. Sedaghat, S. Paydar, Improving the stateful robustness testing of embedded real-time operating systems, in: 2016 6th International Conference on Computer and Knowledge Engineering (ICCKE), 2016, pp. 159–164 (Oct 2016). doi:10.1109/ICCKE.2016.7802133.

[45] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, ACM, New York, NY, USA, 2002, pp. 58–70 (2002). doi:10.1145/503272.503279.

[46] T. Ball, S. K. Rajamani, The SLAM Project: Debugging System Software via Static Analysis, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, ACM, New York, NY, USA, 2002, pp. 1–3 (2002). doi:10.1145/503272.503274.

[47] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher, Model Checking Concurrent Linux Device Drivers, in: Proceedings of the Twenty-second IEEE/ACM International

Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 501–504 (2007). doi:10.1145/1321631.1321719.

[48] S. Chaki, E. M. Clarke, A. Groce, S. Jha, H. Veith, Modular verification of software components in C, IEEE Transactions on Software Engineering 30 (6) (2004) 388–402 (June 2004). doi:10.1109/TSE.2004.22.

[49] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, N. Sinha, Concurrent software verification with states, events, and deadlocks, Formal Aspects of Computing 17 (4) (2005) 461–483 (Dec 2005). doi:10.1007/s00165-005-0071-z.
URL https://doi.org/10.1007/s00165-005-0071-z

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: Formal Verification of an OS Kernel, in: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 207–220 (2009). doi:10.1145/1629575.1629596.

[51] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, G. Heiser, Timing analysis of a protected operating system kernel, in: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS11), Vienna, Austria, 2011, pp. 339–348 (November 2011).

[52] J. Corbet. The kernel lock validator [online] (May 2006).
URL: https://lwn.net/Articles/185666/.

[53] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, 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, ASPLOS '18, ACM, New York, NY, USA, 2018, pp. 405–418 (2018). doi:10.1145/3173162.3177156.
URL http://doi.acm.org/10.1145/3173162.3177156

[54] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. 16 (3) (1994) 872–923 (May 1994). doi:10.1145/177492.177726.

[55] C. Marinas. Formal methods for kernel hackers [online] (2018).
URL: https://linuxplumbersconf.org/event/2/contributions/60/attachments/18/42/FormalMethodsPlumbers2018.pdf.

[56] B. B. Brandenburg, Scheduling and Locking in Multiprocessor Real-Time Operating Systems, Ph.D. thesis, University of North Carolina at Chapel Hill (2011).
URL https://cs.unc.edu/~anderson/diss/bbbdiss.pdf

[57] C. G. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, 2nd Edition, Springer Publishing Company, Incorporated, 2010 (2010).

[58] P. J. Ramadge, W. M. Wonham, Supervisory control of a class of discrete event processes, SIAM J. Control Optim. 25 (1) (1987) 206–230 (Jan. 1987). doi:10.1137/0325013.

[59] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, 3rd Edition, Intel Corporation, 2016 (2016).

[60] D. B. de Oliveira. Mind the gap between real-time Linux and real-time theory, Part I [online] (2018).
URL: https://wiki.linuxfoundation.org/realtime/events/rt-summit2018/schedule#abstracts.

[61] D. B. de Oliveira. Mind the gap between real-time Linux and real-time theory, Part II [online] (2018).
URL: https://www.linuxplumbersconf.org/event/2/contributions/75/.

[62] D. B. de Oliveira. How can we catch problems that can break the preempt_rt preemption model? [online] (2018).
URL: https://linuxplumbersconf.org/event/2/contributions/190/.

[63] J. Corbet. Statistics from the 4.17 kernel development cycle [online] (May 2018).
URL: https://lwn.net/Articles/756031/.

[64] P. McKenney. A realtime preemption overview [online] (August 2005).
URL: https://lwn.net/Articles/146861/.

[65] K. Akesson, M. Fabian, H. Flordal, R. Malik, Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems, in: 2006 8th International Workshop on Discrete Event Systems, 2006, pp. 384–385 (July 2006). doi:10.1109/WODES.2006.382401.

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

[67] D. B. de Oliveira. Linux Task Model [online] (2019).
URL: http://bristot.me/linux_task_model/.

[68] E. Bartocci, Y. Falcone (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics, Vol. 10457 of Lecture Notes in Computer Science, Springer, 2018 (2018). doi:10.1007/978-3-319-75632-5.
URL https://doi.org/10.1007/978-3-319-75632-5

[69] D. B. de Oliveira. __schedule() being called twice, the second in vain [online] (July 2018).
URL: http://bristot.me/__schedule-being-called-twice-the-second-in-vain/.

[70] D. B. de Oliveira. BUG: ftrace/perf dropping events at the begin of interrupt handlers [online] (2018).
URL: https://www.spinics.net/lists/linux-rt-users/msg19781.html.

[71] D. B. de Oliveira. Early context tracking patch set: fixing perf and ftrace losing events [online] (2019).
URL: http://bristot.me/early-context-tracking-patch-set-fixing-perf-ftrace-losing-events/.

[72] D. B. de Oliveira. BUG-RT: scheduling while in atomic in the watchdog's hrtimer [online] (2019).
URL: https://www.spinics.net/lists/linux-rt-users/msg20376.html.

[73] D. B. de Oliveira. Beyond the latency: New metrics for the real-time kernel [online] (2018).
URL: https://linuxplumbersconf.org/event/2/contributions/241/.