Modeling the Behavior of Threads in the PREEMPT_RT Linux Kernel Using Automata

Daniel Bristot de Oliveira
Red Hat, Inc.
Scuola Superiore Sant'Anna
Universidade Federal de Santa Catarina
bristot@redhat.com

Tommaso Cucinotta
Scuola Superiore Sant'Anna
tommaso.cucinotta@santannapisa.it

Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina
romulo.deoliveira@ufsc.br

ABSTRACT

This article proposes an automata-based model for describing and verifying the behavior of thread management code in the Linux PREEMPT_RT kernel, on a single-core system. The automata model defines the events that influence the timing behavior of the execution of threads, and the relations among them. This article also presents the extension of the Linux trace features that enable the trace of such events in a real system. Finally, one example is presented of how the presented model and tracing tool helped catching an inefficiency bug in the scheduler code and ultimately led to improving the kernel.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; *Embedded systems*;

KEYWORDS

Real-time systems, Linux kernel, behavioral modeling, code verification, automata, tracing.

ACM Reference Format:
Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. 2018. Modeling the Behavior of Threads in the PREEMPT_RT Linux Kernel Using Automata. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Real-time Linux has been a research topic for more than a decade now, with many scheduling algorithms being implemented in Linux [3, 7, 22, 25]. Despite the constant interaction of Linux development and the real-time research, some conceptual divergence between these groups still exists.

For instance, the common assumption that tasks are completely independent, that the operating system is (fully) preemptive, and that operations are atomic [6, 7] is a frequent critic from Linux developers. They argue that the results of the development of theoretical schedulers does not reproduce the reality of real-time applications running on Linux [15].

In practice, the interaction among tasks can cause some non-negligible delays, even for functions that are not directly related, caused by in-kernel operations [9]. Those in-kernel operations are necessary because of the non-atomic nature of a sophisticated operating system 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 block the execution of the scheduler. Then, interrupts must be disabled as well, to avoid race conditions with interrupt handlers. Hence, delays in the scheduling and interrupt handler are created during activation of a thread [12].

The understanding of such operations, and how they affect the timing behavior of a task, are fundamental for the development of real-time algorithms for Linux. However, the amount of effort required for a researcher to understand all these constraints is not negligible. Rather, 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, turning effectively into a real-time operating system (RTOS). 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 post-ponement of most interrupt handling code to kernel threads as done in the PREEMPT_RT kernel, 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 the system administrator.

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, in a format that facilitates the understanding of Linux code for real-time researchers without being too far from the way developers observe and improve Linux.

Real-time Linux developers evaluate the system using tracing. They interpret a chain of events to understand and improve Linux' timing behavior. For instance, they use `ftrace` or `perf` to trace kernel events like interrupt handling, wakeup of a new thread, context switch, etc.. while `cyclictest` measures the latency of the system.

Ewili'18, 4 October, 2018, Torino, Italy.

When `cyclitest` hits a large latency, the *trace* is interpreted, to identify the chains of events that bring the system to the *state* that caused the latency[1]. Then the kernel is modified to avoid this state.

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

Following the approach presented for IRQs on PREEMPT_RT Linux [13], this article proposes an automata model for threads in Linux on a single-core system. The automata model defines the events that influence the timing behavior of the execution of threads, and the relation among each event.

Paper Contributions. This article discusses an automaton model for threads in Linux, on a single core, and also the extension of the Linux trace features that enable the trace of such events in a real system. The paper also demonstrates how the model can improve the understanding of Linux properties. Finally, it shows one example of how the presented model and tracing techniques helped to catch a bug leading to some inefficiency in the scheduler code, so that a patch could be developed, which has been submitted and accepted by developers for upstream merging.

Paper Organization. The paper is organized as follows: Section 2 briefly recalls some related work and Section 3 provides a short summary of the automata theory used in this paper; Section 4 provides some details of the used modeling strategy and discusses the development of the proposed model. Section 5 describes part of the proposed model, based on the concepts introduced in the previous sections. Finally, Section 6 presents the next steps of this work, pointing toward a better description of Linux's tasks using well defined real-time theory terms.

2 RELATED WORK

Software verification is an active and bustling area of research, with many techniques involving the use of automata theory [27] or other state-based modeling methodologies, temporal logics and/or techniques similar to process calculus. These are aimed at either ensuring that a given safety/correctness predicate on the system state can never be violated, or, in case a violation is possible, these techniques aim at finding an execution trace/scenario leading to the faulty state, useful to debug the system (or sometimes its abstract model). Classical examples involve modeling and analysis of locking schemes and distributed application protocols, e.g., by using well-known tools such as SPIN [17], TLC+ using TLA+ [20] and/or PlusCal models [21]. These formalisms can also handle verification of timing properties for real-time systems [2]. It is particularly challenging to apply these techniques on code written in general-purpose programming languages, such as C/C++ or Java: either the software is so simple to allow for a complete enumeration of all the possible states, or – the majority of the times – one ends up with the inherently undecidable problem of checking whether or not a predicate can ever be violated. Also, for complex software, the model is usually built as an abstraction of the actual software behavior, introducing a risky semantic gap between the model and the actual software behavior. Such a gap may be reduced by approaches proposing automatic model generation from C code [23], which have the inherent drawback of producing overly big and complex models. However, many techniques have been developed that allow for huge reductions of the search space, allowing these techniques to be usable with a reasonable processing time in various cases of real industrial software. A remarkable example is the use of TLA+ and PlusCal within Amazon Web Services [24], leading to the discovery of various design bugs in DynamoDB, S3, EBS, EC2 and other software components.

In this context, 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 one by Henzinger and others [16], who used control flow automata, 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, implemented within the BLAST tool, 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 [4] 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. More recently, Witkowski et al. [28] proposed the DDVerify tool, extending on the capabilities of BLAST and SLAM, e.g., supporting synchronization constructs, interrupts and deferred tasks.

Another remarkable work is the `lockdep` mechanism [10] built into the Linux kernel, capable of identifying errors in using locking primitives that might eventually lead to deadlocks. The mechanism includes detection of mistaken order of acquisition of multiple (nested) locks throughout multiple kernel code paths, and detection of common mistakes in handling spinlocks across IRQ handler vs process context, e.g., acquiring a spinlock from process context with IRQs enabled as well as from a IRQ handler. Interestingly, the number of different spinlock states that has to be kept by the kernel is reduced by applying the technique based on individual locking classes, rather than individual locks.

There have also been other remarkable works assessing formal correctness of a whole micro-kernel such as seL4 [19], 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 [5]. These findings were possible thanks to the simplicity of the seL4 micro-kernel features, e.g., semi-preemptability.

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 verification of in-kernel code sections responsible for such low-level operations such as task scheduling, IRQ

[1] http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt

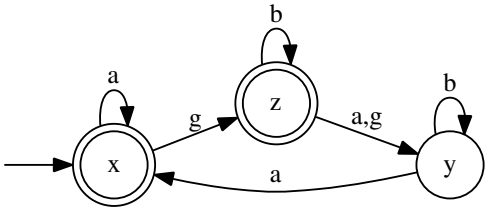

Figure 1: State transitions diagram (based on Figure 2.1 from [8]).

and NMI management, and their delicate interplay. This paper focuses on the automata-based modeling of the task scheduling code, extending our related prior work [13] which focused on the IRQ and NMI management code. These works shed some light exactly into this uncovered spot, in the industrial and research practice literature.

3 BACKGROUND

We model the evolution of threads on Linux over time as a Discrete Event System. A DES can be described in various ways, for example using a *language* (that represents the "legal" 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.

A trace of its run-time behavior can be described as a sequence of the visited states and the associated events causing state transitions. Hence, a DES evolution is described as a sequence of events $e_1, e_2, e_3, ...e_n$.

All possible sequences of events define the language that describes the system. Representing a language using an appropriate modeling formalism is then fundamental for the analysis, control and performance evaluation of a DES.

The starting point to describe a DES is the underlying set of events $E = \{e_i\}$ associated with it, that represents the "alphabet" used to form "strings" ("traces") of events that compose the DES language. This framework can be used either to define the language to be performed by a new system, or to formally identify the language understood by an existing system.

A DES can be formally modeled through a language, $\mathcal{L}(G)$, describing all admissible sequences of events that the DES can produce or process. 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 method.

One of the key features of an automaton is its directed graph or state transition diagram representation. For example, consider the event set $E = \{a, b, c\}$ and the state transition diagram in Figure 1, where nodes represent the system states, labeled arcs represent transitions between states, the arrow points to the initial state and the nodes with double circles are marked states. The marked state is the safe state of the the system.

Formally, a deterministic automaton, denoted by G, is a quintuple

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

where X is the set of states, E is the finite set of events, $f : X \times E \rightarrow X$ is the transition function (defining the state transition between states from X due to events from E), x_0 is the initial state and $X_m \subseteq X$ is the set of marked states.

For instance, the automaton G represented on Figure 1 can be described by defining $X = \{x, y, z\}$, $E = \{a, b, g\}$, $f(x, a) = x$, $f(y, a) = x$, $f(z, b) = z$, $f(x, g) = z$, $f(y, b) = y$, $f(z, a) = f(z, g) = y$, item $x_0 = x$ 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 $e \subseteq E$ with $f(x_0, e)$ defined. 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 generated by an automaton $G = \{X, E, f, x_0, X_m\}$ consists of all possible chains of events generated by the state transition diagram starting from the initial state.

One important language generated by automata is the marked language. The marked language is composed of the set of words in $\mathcal{L}(G)$ that lead the state transition diagram to a marked state. 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 secure 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 synchronized to compose a single automaton. In the parallel composition, events not shared between the automata are possible at any state in which it is possible in the local state. Events shared between two automata are possible only when it is possible in every automaton for which the event is part of the set of events. The initial state of the parallel composition is the initial state of all the composed automata. A state is marked if and only if the state is marked in all the automata in the parallel composition.

In general, complex systems can be modeled as composed of many concurrent (and simpler) sub-systems. Automata operations enable the modeling of a complex DES by decomposing it in modules. For example, the approach presented by Ramadge and Wonham [26] allows the modeling of a system composed by many sub-systems. With this approach, the system is modeled as a set of completely independent sub-systems and each sub-system is known as a plant or generator. The composition of all sub-systems generates all possible chains of events, even sequences of events that cannot really be generated by the system in practice. Hence, specifications are defined to remove "impossible sequences" from the language. Specifications are automata using events common in the generators they aim to synchronize.

Using such approach, a thread on Linux can be modeled using a set of sub-systems; then, the restrictions imposed to the possible sequences of events are modeled, allowing the interaction of each event to be precisely described.

4 PROPOSED APPROACH

During the model development, described in Figure 2, the informal knowledge about Linux tasks' is modeled using automata theory. The main source of information, in order of importance, are previous papers about the subject [12], kernel code, documentation

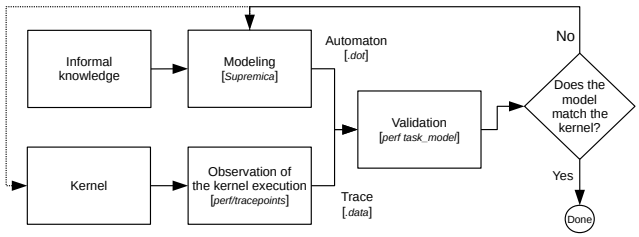

Figure 2: Modeling Phases.

Table 1: Automaton and Kernel events relation and status.

Kernel event	Automaton event	Status
irq:local_irq_disable	local_irq_disable	new
irq:local_irq_enable	local_irq_enable	new
sched:sched_preempt_disable	preempt_disable	new
sched:sched_preempt_enable	preempt_enable	new
sched:sched_need_resched	sched_need_resched	new
sched:sched_set_state	sched_set_state_runnable	new
sched:sched_set_state	sched_set_state_sleepable	new
sched:sched_entry	schedule_entry	new
sched:sched_exit	schedule_exit	new
sched:sched_switch	sched_switch_in	exist
sched:sched_switch	sched_switch_in_o	exist
sched:sched_switch	sched_switch_out_o	exist
sched:sched_switch	sched_switch_preempt	exist
sched:sched_switch	sched_switch_suspend	exist
sched:sched_waking	sched_waking	exist

and the observation of the system's execution using various tracing tools, and hardware documentation [18]. During the development of the model, the execution of the model and the execution of the kernel are checked continuously one against the other. If a problem is detected, the chain of events is evaluated, resulting in a change in the model. However, as the model becomes more mature, it can be the case that the problem resides in the kernel, as a result of a misusage of the kernel methods. If a kernel problem is detected, a change in the kernel can be proposed.

4.1 Events

The events used in the automata modeling and their relative kernel events are presented in Table 1. The status column shows if trace event of the kernel is new or if it is an existing one. When a kernel event refers to more than one event, the extra fields of the kernel event are used to distinguish between automaton events.

The `perf` tool was extended to collect the trace of the kernel events used for the modeling.

4.2 Modeling

The automata describing the formal model have been developed using the Supremica IDE [1]. Supremica is an integrated environment for verification, synthesis and simulation of discrete event systems using finite automata. Supremica allows exporting the result of the modeling in the DOT format that can be plotted using `graphviz` [14], for example.

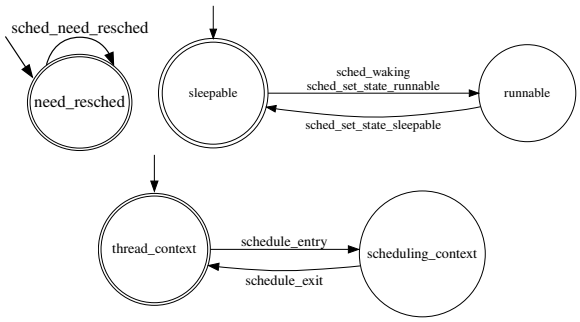

Figure 3: Examples of generators: Need Resched (on top, left) Sleepable and Runnable (on top, right) Scheduling Context (bottom).

An efficient way to model complex systems is using the modular approach. In the modular approach, rather than modeling the system as a single automaton, the system is divided into *generators* and *specifications*. The generators are the system's events modeled as a set of independent sub-systems, where each sub-system has its own independent set of events. Then, the relation of the events of each sub-system is modeled as a set of specifications. Similarly, each specification is modeled independently, using the alphabet of the sub-systems of the generator it aims to control.

Examples of generators are shown in Figure 3. The Need Resched generator contains only one event and one state. The Sleepable and Runnable generators have 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 models the call and return of the main scheduling function of Linux, which is `__scheduler()`.

Figure 4 shows the specifications of events that allows a task to call the scheduler. The explanation of this specification is done in the next section.

The final model is done with the parallel composition of all generators and specifications. It is composed of 15 events, 7 generators, and 10 specifications. Resulting in 149 states and 327 transitions. The model is non-blocking, has no forbidden states and is deterministic.

As expected, the final model has many events, which makes it difficult to use. However, the modular approach allows the use of only parts of the model in the analysis, as demonstrated in the next section.

5 EARLY RESULTS

The modular modeling of the thread creates many possibilities. The first is the ability to analyze the properties of threads in Linux without a deep knowledge of the internals of the kernel.

For instance, the model in Figure 4 presents the conditions for the thread under analysis to call the scheduler. In the initial state, in which the thread is not running, the `schedule_entry` event is recognized. When the `sched_switch_in` switch the state of the thread to `running`, the `schedule_entry` is not allowed. The `running` state,

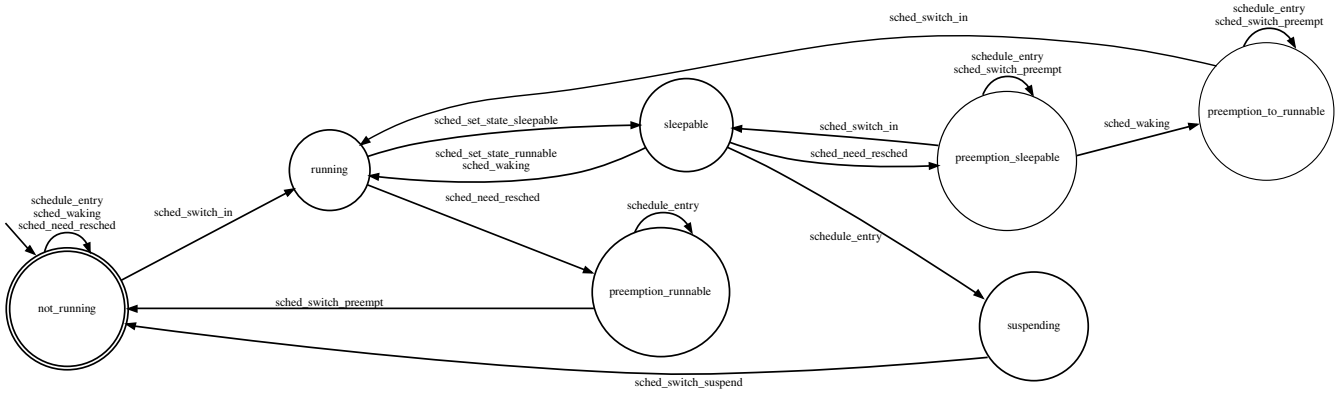

Figure 4: Example of Specifications: Necessary and Sufficient conditions to call the scheduler.

```
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]   784.425941:         sched:sched_entry: at preempt_schedule_common
5:     kworker/0:2   728 [000]   784.425945:        sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:1:724 [120]
6: irq/14-ata_piix    86 [000]   784.426515:        sched:sched_waking: comm=kworker/0:2 pid=728 prio=120 target_cpu=000
7:     kworker/0:1   724 [000]   784.426610:        sched:sched_switch: kworker/0:1:724 [120] t ==> kworker/0:2:728 [120]
8:     kworker/0:2   728 [000]   784.426616:         sched:sched_entry: at schedule
9:     kworker/0:2   728 [000]   784.426619:        sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:2:728 [120]
```

Figure 5: Kernel trace excerpt.

though, recognizes two events, the `sched_set_state_sleepable` and the `sched_need_resched`.

In the case of the occurrence of the event `sched_set_state_sleepable`, the thread changes the state to `sleepable`, where the `schedule_entry` is recognized. Hence, the occurrence of `sched_set_state_sleepable` is *sufficient* to call the scheduler. However, 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.

In the other case of the occurrence of the event `sched_need_resched`, the `schedule_entry` 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 causing a preemption. 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.

Another possibility created by the model is the identification of problems of the kernel. For example, taking the trace of Figure 5, considering the thread `kworker/0:2` in analysis, and the model in Figure 4 in the initial state, the events and state transitions of Table 2 happens.

The thread `kworker/0:2` started to run at Line 1. After in `running` state, it sets its state to `sleepable` in Line 2, followed by the `need_resched` event in Line 3, causing the scheduler to be called in Line 4. Then, the thread switched the context in preemption and left the processor. At Line 6 the thread is awakened, switching

Table 2: Events and state transitions of Figure 5.

Line	Event	New state
1	`sched_switch_in`	`running`
2	`sched_set_state_sleepable`	`sleepable`
3	`sched_need_resched`	`preemption_sleepable`
4	`sched_entry`	`preemption_sleepable`
5	`sched_switch_preempt`	`preemption_sleepable`
6	`sched_waking`	`preemption_to_runnable`
7	`sched_switch_in`	`running`
8	`sched_entry`	not recognized!

the state to `preemption_to_runnable`, and then at Line 7 the `context_switch_in` takes place and the thread starts to run. However, right after returning from the scheduler function, the thread calls the scheduler again at Line 8, and this was not expected.

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 suggestions of the fix was to not call the

scheduler during the path to call the scheduler. With the suggested fix, the language generated by the system was recognized by the automata. The suggestion has been recently submitted to the real-time Linux kernel development list, and it was accepted for mainline integration [11].

6 CONCLUSIONS

The understanding of in-kernel operations, and how they affect the timing behavior of a task, are fundamental for the development of real-time algorithms for Linux. However, due to the complexity of Linux' code base, this understanding may need an prohibitive amount of time, justifying the creation of a simplified model, that abstracts the complexity of the system, while describing the essential operations that influence the timing behavior of threads on Linux. The timing behavior of Linux is evaluated using the trace of events by practitioners, which are common notions for the automata theory, used in the modeling of DES. The usage of the automata theory adequately enabled the modeling of Linux, allowing the statement to of Linux' properties, as used in real-time papers, and the detection of non-optimal behaviors on Linux, as in practice. This is not a finished work, however. The model needs to be extended to comprise locking and interrupts, in the first moment, and also to include multi-core operations, like the migration of threads. Moreover, the model can also be used in the development of a validator of the timing behavior of tasks on Linux, like `lockdep` does for locks.

REFERENCES

[1] Å kesson, K., Fabian, M., Flordal, H., and Malik, R. Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. In *Discrete Event Systems, 2006 8th International Workshop on* (2006), pp. 384–385.

[2] Abadi, M., and Lamport, L. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994), 1543–1571.

[3] Abeni, L., Goel, A., Krasic, C., Snow, J., and Walpole, J. A measurement-based analysis of the real-time performance of linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium* (2002), pp. 133–142.

[4] Ball, T., and Rajamani, S. K. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 1–3.

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

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

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

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

[9] Cerqueira, F., and Brandenburg, B. 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.

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

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

[12] de Oliveira, D. B., and de Oliveira, R. S. Timing analysis of the PREEMPT RT linux kernel. *Softw., Pract. Exper. 46*, 6 (2016), 789–819.

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

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

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

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

[17] Holzmann, G. J. The model checker spin. *IEEE Trans. Softw. Eng. 23*, 5 (May 1997), 279–295.

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

[19] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 207–220.

[20] Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.

[21] Lamport, L. *The PlusCal Algorithm Language.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 36–60.

[22] Lelli, J., Scordino, C., Abeni, L., and Faggioli, D. Deadline scheduling in the linux kernel. *Software: Practice and Experience 46*, 6 (2016), 821–839.

[23] Methni, A., Lemerre, M., Ben Hedia, B., Haddad, S., and Barkaoui, K. *Specifying and Verifying Concurrent C Programs with TLA+.* Springer International Publishing, Cham, 2015, pp. 206–222.

[24] Newcombe, C. *Why Amazon Chose TLA +.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 25–39.

[25] Palopoli, L., Cucinotta, T., Marzario, L., and Lipari, G. Aquosa – adaptive quality of service architecture. *Softw. Pract. Exper. 39*, 1 (Jan. 2009), 1–31.

[26] Ramadge, P. J., and Wonham, W. M. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim. 25*, 1 (Jan. 1987), 206–230.

[27] Vardi, M. Y., and Wolper, P. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. on Logic in Computer Science* (1986), pp. 322–331.

[28] Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. Model checking concurrent linux device drivers. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 501–504.