Automata-Based Modeling of Interrupts in the Linux PREEMPT RT Kernel

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

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

***Abstract*—This paper presents a methodology to model and check the behavior of a part of the Linux kernel by applying automaton theory and in-kernel tracing from real execution. It is possible to check that the state transitions of the kernel during a real execution match with the allowed ones, according to the formal model. The scope of the paper is limited to the IRQ/NMI subsystem of the Linux kernel.**

***Index Terms*—Real-time, Linux, Modeling, Discrete event system, Automata.**

I. Introduction

Linux, albeit being a general-purpose operating system, has been evolving over time in terms of features and timing behavior, so as to become increasingly suitable for a multitude of other and more challenging scenarios, particularly real-time (RT) systems. Linux has undergone a remarkable and relentless effort by several developers to improve real-time performance, e.g., adding full preemptability, chasing and removing any use of the old global "big kernel lock", up to the PREEMPT_RT [1] rework of its kernel internals. This includes deferring part of interrupt handlers kernel threads, and recasting spinlocks and semaphores as rt-mutexes, where priority inheritance can work to prevent priority inversion scenarios, among others. Finally, the introduction of the SCHED_DEADLINE [2] scheduler based on the Constant-Bandwidth Server [3] and EDF added to the capabilities to support real-time and deadline-constrained workloads. The impact of this evolutionary process is witnessed by a number of successful Linux-based OS products for industrial real-time scenarios, such as Red Hat Enterprise Linux for Real-Time [4] (RHEL-RT), among others.

The *de facto* standard for kernel developers and practitioners to evaluate the real-time performance of the kernel is by using the `cyclictest` tool. This allows for measuring the *kernel latency*, defined as the time elapsed from when a task is supposed to resume execution according to its programmed wake-up time, and the time it is actually dispatched on a CPU [5]. Practitioners' mission is to keep the kernel latency bounded, below a precise threshold, e.g., RHEL-RT is known for a latency below 150μs on Intel x86_64 platforms.

On the other hand, the academic literature on real-time systems relies heavily on well-founded theoretical models of real-time applications and kernel behavior, on top of which mathematical abstractions can be effective in identifying the worst-case scheduling scenarios leading to the highest possible response times. However, such approaches often cannot be directly applicable in the context of an OS/kernel like Linux, due to the mismatch between such theoretical behavioral models and the inherent complexity due to the GPOS nature of Linux. The situation is often overly complicated by the presence of different contexts (user-space, kernel-space, "soft IRQ", "hard IRQ", ...), as well as the possibility to enable or disable at any given time hardware (maskable) interrupts, preemption and migrations.

Theoretical schedulability analysis techniques could be improved to model some of the complexities mentioned above, e.g., by modeling kernel latencies as jitters [6] or as blocking times, or by measuring the aggregated effect of OS overheads, and adding it to the tasks execution times [7]. However, these approaches apply theoretical tools for worst-case analysis, using empirical measurements where the worst-case is rarely observed, so the actual input data needed in the analysis, such as worst-case kernel latency, interrupts' minimum inter-arrival times, etc., remains largely unknown or dangerously under-estimated. While the current engineering approach to evaluate such data is to empirically measure them on a running system for extended time periods (see for example the OSADL Realtime QA Farm[1]), a more theoretically sound approach is needed. A first baby-step into the direction of such a more detailed analysis of the Linux kernel timing behavior is the one to build a precise behavioral model of the complex kernel internals, starting from IRQ/NMI handling and task scheduling. It is also important to have practical and automated means to verify the correctness of the model, as well as the correctness of the kernel own behavior w.r.t. such model, particularly useful to verify the absence of latency regressions after bugfixes or enhancement patches.

a) Paper contributions: This paper represents a first step toward the definition of a formal model of the internal behavior of the Linux kernel, focusing on the subsystem handling Non-Maskable Interrupts (NMIs) and regular hardware interrupts (IRQs). The model is based on automata theory [8], [9], where

978-1-5090-6505-9/17/$31.00

[1]https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html

the actions influencing the interrupt handlers behavior are modeled as events.

In the proposed approach, the actual system behavior at run-time is traced in specific points of the kernel by using the `perf` tool[2], and the obtained trace is automatically matched against the possible run-time behaviors as specified in the proposed formal model through the verification tool presented in Section III. The verification tool traces Linux execution, checking if traces recorded on a running Linux kernel match with the automata-based theoretical model. By confirming that the sequences of events generated by Linux are correctly captured by the automata, it is possible to show that the system is correctly modeled, as well as that its observed evolution is compliant with the formal model.

b) Paper organization: The paper is organized as follows: Section II provides a short summary of the automata theory used in this paper; Section III provides some details of the used modeling strategy and discusses the development and verification of the proposed model, while Section IV describes the proposed model, based on the concepts introduced in the previous sections. Finally, Section V briefly recalls some related work and Section VI presents the conclusion of this work, pointing to the next step toward a better description of Linux's tasks using well defined real-time theory terms.

II. Background

Since the formal model for NMIs and IRQs presented in this paper is described through automata, this section quickly recalls basic concepts of automata and the related theory.

We model the Linux kernel evolution over time as a Discrete Event System (DES). 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 DES is characterized by a number of (internal) states. 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" (or "words" or even "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 spoken by an existing system.

A string composed of no events is called the empty string and it is denoted by ε. The length of a string is the number of events contained in it, counting repeated events. If s is a string, $|s|$ will denote the length of s.

[2]See https://perf.wiki.kernel.org/index.php/Main_Page.

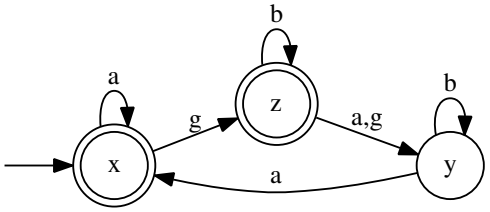


Fig. 1. State transitions diagram (based on Figure 2.1 from [9])

a) Language Definition: A language defined over an event set E is a set of strings formed from events in E.

For example, if the event set is $E = \{a, b, c\}$, then it is possible to define a language $L_1 = \{\varepsilon, a, ab, acc\}$ composed by the four strings/traces ε, a, ab, acc. It is also possible to define the language L_2 composed by all strings with two elements starting from a: $L_2 = \{aa, ab, ac\}$. Of course, it is also possible to define languages with an infinite set of strings/traces, such as, for instance, $L_3 = \{all\ strings\ starting\ with\ a\}$.

A DES can be formally modeled through a language describing all admissible sequences of events that the DES can produce or process, but this kind of modeling is not easy for complex systems, because of the absence of an additional level of structures to describe the system logical behavior. This is where automata come to the rescue. Moreover, the automata formalism is amenable to composition operations, and analysis as well, considering the finite-state case.

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. 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, $f : X \times E \to X$ 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 [10] 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, IRQs and NMIs can be modeled using a set of sub-systems; then, the restrictions imposed to the possible sequences of events (interrupt handlers cannot execute before the corresponding interrupt fires, etc...) are modeled as specifications, allowing the interaction of each event to be precisely described.

III. Model Development

During the model development, described in Figure 2, the informal knowledge about Linux tasks' are modeled using automata theory. The main source of information, in order of importance, are previous papers about the subject [6], hardware vendor documentation [11], kernel documentation and the observation of the system's execution using various tracing tools.

The NMI and IRQ handlers in the Linux kernel have been modeled as two automata, whose alphabets are the two sets of relevant kernel events in Tables I and II.

On a running Linux system, traces composed of these events can be captured using tracepoints[3]. The Linux kernel already provides some useful tracepoints, but others were added as well. Currently, it is not possible to trace hardware events such as the "irq" or "nmi" events in Linux; however, it is possible to assert if the event took place. For example, the interrupt handler is called as a side effect of the occurrence of an interrupt, hence an "irq_entry" event implies that a corresponding "irq" event happened before. The `perf` tool enables and collects the tracepoints from the kernel.

[3]See https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

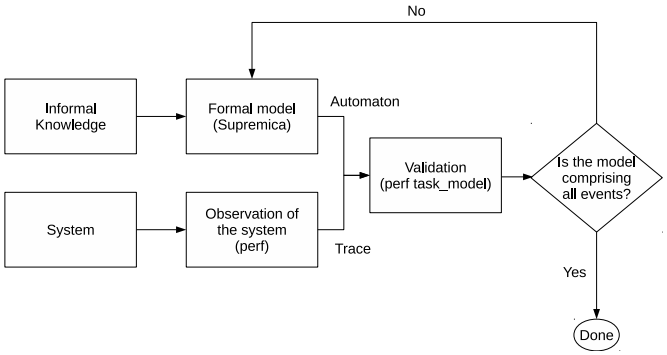


Fig. 2. Modeling Phases.

TABLE I
Alphabet E^{NMI}, used to describe NMI.

Event	Description
nmi	Non-maskable interrupt
nmi_entry	NMI's software handler start
nmi_exit	NMI's software handler return

The automata describing the formal model have been developed using the Supremica IDE [12]. Supremica is an integrated environment for verification, synthesis and simulation of discrete event systems using finite automata. Supremica allows to export the result of the modeling in the DOT format that can be plotted using `graphviz` [13], for example.

To validate the model, the `perf` tool was extended with a new command named `task_model`. This new command automates the tracing of kernel events and the verification of the model against the captured trace. `perf task_model` works in two stages: first, in the `record` mode, the tool

TABLE II
Alphabet E^{IRQ}, used to describe IRQs.

Event	Description
nmi_entry	NMI's software handler start
nmi_exit	NMI's software handler return
irq	Hardware interrupt
irq_entry	IRQ's software handler start
irq_exit	IRQ's software handler return
irq_disable	Disables a single IRQ on all CPUs
irq_enable	Enables a single IRQ on all CPUs
local_irq_enable	Enable local interrupts
local_irq_disable	Disable local interrupts, by software
local_irq_disable_hw	Disable local interrupts, as a consequence of the interrupt
local_irq_disable_hw_n	Disable local interrupts, as a consequence of another interrupt

enables the tracing points associated with the events in E^{NMI} and E^{IRQ} event sets; then, in the verification mode, the tool checks if the captured trace is consistent with the model.

The trace captured in the `record` mode is saved in the default `perf` format. On a 4 CPUs Intel core i7 computer, this generates about $23MB$ of trace per second.

The verification mode receives three inputs: 1) the recorded trace; 2) the NMI model and 3) the IRQ model (both models are exported using the DOT format). With these inputs, an internal representation of the system (composed by m CPUs) is created. Each CPU has one vector of interrupts and one NMI[4]. The NMI and every vector entry are associated to its respective model. At the beginning, all the automata are placed in their initial states; then, the tool starts to parse the trace. Every event is associated to a handler, which is a function that parses it. When invoked, the handler checks if the `event` is accepted in the current state of the model. Each event is tried in the models in which the event is part of the event set. For example, since `nmi_entry` is in both E^{NMI} and E^{IRQ}, all models of the CPU in which the event took place will be checked. On the other hand, `irq_entry` is not present in E^{NMI}, hence the NMI automaton is not tried. If the event is accepted, the current state of the automaton is changed to the new state. If the event is not accepted in any model, an error is generated (printing debug information).

When the tool found that the formal model was not compatible with some trace captured with perf, the model was changed to comprise the observed behavior of the system, and validated again. This iterative refinement of the model was repeated until the automata matched all the traces captured with perf. It was needed around 100 runs to develop the model. In order to create a load in the system, the `rt-tests` tools where used. The rt-tests is the main tool set to evaluate real-time Linux, including `cyclictest`.

IV. A Model for NMIs and IRQs

Intel CPUs provide two mechanisms to notify occurrence of asynchronous or synchronous events (generated by external devices or by the CPU). These mechanisms are Interrupts and Exceptions. Interrupts or exceptions generally force a change in the execution path of the current task, activating the execution of their respective handlers (in kernel space).

Interrupts are used by external devices to notify asynchronous events (handled by the OS kernel using a special routine). For example, a network card uses interrupts to notify the arrival of network packets, which are handled by the driver to deliver the packet contents to an application. Exceptions are generated by the CPU when some predefined exceptional conditions occur (errors like a division by 0, but also page faults, or the execution of CPU instructions that a debugger is waiting for, etc.). While exceptions are synchronous (always generated in response to events happening in the current program), interrupts can be raised due to requests not related to the current program. Therefore, exceptions can be considered as part of the current task, while interrupts are asynchronous and can not be considered part of it (so, they are considered to be executing in their own context - or being a different class of tasks). As this work aims at modeling the system as a set of tasks, hereafter we consider only the interrupts behavior.

[4]Due to its particular behavior, the NMI is represented outside of the interrupt vector.

A. Interrupts

The automata presented in this paper model both the hardware NMIs / IRQs and their handlers in the Linux kernel. Hardware NMIs and IRQs are delivered to the CPU when the values of two dedicated logical lines (NMI and INTR) are set to 1, and are associated with the *nmi* and *irq* events.

The left part of Figure 3 presents the hardware part of a NMI. The hardware activates the NMI, represented by the *nmi* event. A similar behavior is presented by an IRQ, as modeled with the automaton presented in the right part of the figure.

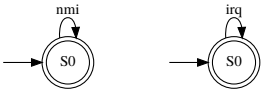


Fig. 3. NMI (on the left) and IRQ (on the right) states.

Note that each automaton constrains the sequence of possible events: node names are automatically picked by Supremica, so nodes with the same name in different diagrams are not related, whilst events with the same name are the same event.

B. Interrupt Handlers

On occurrence of an IRQ, the processor saves the context of the current task and starts the execution of the IRQ handler, which is a software routine. The control returns to the current task after the handling of the IRQ.

Figure 4 presents the handlers of an NMI (on the left) and of an IRQ (on the right). The *nmi_entry* event notifies the start of the NMI handler, that finishes execution with the *nmi_exit* event, while the *irq_entry* event notifies the start of the IRQ handler, that finishes its execution with the *irq_exit* event.

C. Hardware and Software Interrupts Modeling

The hardware and software models presented in Figure 3 and 4 must be synchronized. The synchronization of the automaton for hardware NMIs and the automaton for their handler (Figure 4) results in the automaton of Figure 5, that enables all the possible combinations of events. However, there are sequences of events which are not possible: for example, it is not possible to have an *nmi_entry* event before the corresponding *nmi* event.

In order for this automaton to be correct, a control specification must be added to it, as shown in Figure 6: according to this specification, the software handler events are blocked until the occurrence of the *nmi* event. Moreover, the *nmi* is blocked until the occurrence of the event *nmi_exit* (the NMI is disabled until the end of its handler).

The product of the two generators and the specification generates the full behavior of an NMI (both hardware and

Fig. 4. NMI (on the left) and IRQ (on the right) handler.

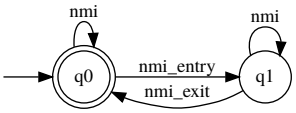


Fig. 5. NMI model describing both hardware and handler.

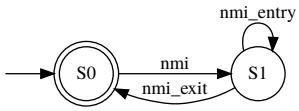


Fig. 6. The software handler is activated by the hardware request.

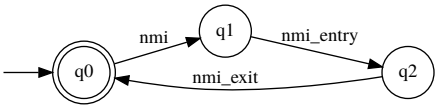


Fig. 7. Final NMI model (hardware and handler) with all the restrictions.

software handler), presented in Figure 7. As expected, this behavior is rather simple: the hardware event takes place, causing the handler to start, followed by the end of the handler.

IRQs differ from NMIs because they can be temporally disabled, delaying the delivery of new IRQ requests and the activation of their handlers. For example, in the Intel CPUs, IRQs can be disabled by modifying a flag in the EFLAGS register (a 32-bits register containing a set of status, control, and system flags, similar to the PSW present in other CPUs). The IF field of this register can be used to block the handling of IRQs in a CPU: IF is set, IRQs are delivered, while when IF is 0, IRQs are delayed until the IF flag is set again. The value of this flag can be changed by using some Assembly instructions, and it is automatically reset to 0 when an IRQ fires. IRQs can be explicitly re-enabled before the end of the handler.

Although interrupt handlers sometimes re-enable IRQs in the vanilla version of the Linux kernel, that is not the case when using PREEMPT RT (where IRQs are not re-enabled before handlers' completion).

Since an IRQ can be disabled by itself, by another interrupt request, or by a thread executing a special CPU instruction, three different events have used to model IRQ disabling. The *local_irq_disable_hw* event is generated when the IRQ itself causes the IRQs to be disabled, the *local_irq_disable_hw_n* event is generated when IRQs are disabled because of a different interrupt, and the *local_irq_disable* event is generated when IRQs are disabled by the execution of an Assembly instruction. A single event represents IRQs being enabled, the *local_irq_enable* event. Figure 8 describes this behavior.

In addition to being disabled in a CPU, it is possible for a single IRQ be disabled on all CPUs, using the *disable_irq()* function of the Linux kernel (the IRQ can be later enabled again by using the *enable_irq()* function). The model for these functions is presented in Figure 9.

All the automata described above need to be complemented by some specifications to correctly model only the possible sequences of interrupt generation, handler execution, IRQ disabling/enabling, etc. For example, local IRQs are disabled during the execution of the handler, and the disabling happens after the hardware generates the *irq* event (IRQs are later re-enabled when the handler finishes its execution). This specification is modeled in Figure 11.(a).

Figure 10 restricts the software handler to execute after IRQs have been disabled in hardware. This is done by not enabling the *irq_entry* and *irq_exit* before the occurrence of the event disabling local IRQs as a side effect of its *irq* event.

Once IRQs have been disabled by the hardware, they stay disabled until the end of the handler (*irq_exit* event). This is an important property because it means that once the handler is started IRQs will not suffer interference from other IRQs, resembling non-preemptive tasks. This specification is modeled in Figure 11.(b).

Once an IRQ starts being handled, at least from the operating system point of view, it will not fire again. So, the *irq* event remains blocked until the return of the handler, as modeled by the specification in Figure 11.(c).

The *disable_irq* event inhibits the hardware starting handling an IRQ (and also has an effect similar to masking local IRQs). This state will be held until IRQs are enabled again (*enable_irq* event). The automaton of Figure 11.(h) models the restrictions imposed by the combination of methods which may mask an IRQ. After being fired, an IRQ will have its handler delayed if, for some reason, it is masked, and this can happen for 3 reasons: the single IRQ is disabled, local IRQs are disabled, or a different IRQ disabled local IRQs.

In the initial state, neither local nor the specific IRQ is masked. The methods to mask IRQs can be nested, and this added complexity to the specification, because it needs to keep track of which methods are masking the IRQ. If an IRQ fires while masked, its handling will be delayed until being unmasked by all methods. Once unmasked, the methods able to mask an IRQ will be blocked and so the interrupt can be handled, unless a higher priority interrupt takes place.

The previous specification models the interactions between IRQs. However, NMIs can also be interfered by the occurrence of an NMI. From IRQ standing point, the important events are the entry and the exit of the NMI handler, as previously seen in the left part of Figure 4. The specification of the interactions between IRQs and NMIs is shown in Figure 11.(e): the occurrence of an NMI blocks all the other events, until the return of the NMI. In order to reduce the number of states, the next specification (presented in Figure 11.(f)) blocks the *disable_irq* and *enable_irq* events in contexts in which it is known that they do not take place. Finally, Figure 11.(g) shows the composition of all the automata describing NMIs, and

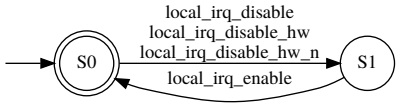


Fig. 8. Disabling all local IRQs.

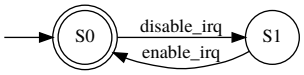


Fig. 9. Disabling a single IRQ.

represents our final model for IRQs.

D. Remarks

As it is possible to observe in Figure 11, the complete model of a single IRQ can lead to an impressive number of states. It would be rather complex to formally describe Linux's IRQ behavior without the usage of a modular approach. The modular approach can also help in a future translation from the automaton specification into a set of rules that describes Linux's behavior in the common vocabulary used in real-time theory. For example, the model in Figure 10 states that IRQ handlers run with local interrupts disabled, and the model in Figure 11.(b) states that interrupts are not enabled again before returning from the IRQ handler. Hence, it is possible to demonstrate that all others IRQs of a processor will be delayed by the handling of another IRQ. In other words, it states that one interrupt cannot preempt another lower priority interrupt. However, it is not possible to state that interrupts are non-preemptive. In the presence of an NMI, all the possible IRQs events are blocked until the return of the NMI handler. As nothing can block the start of the NMI handler, it is possible to state that interrupts are preempted by NMIs, during all NMI's execution. The joint of these two behaviors clarifies that IRQs are not preemptive tasks among its class of tasks, but they are preemptive in the presence of NMI. Finally, the events can be translated into a set of rules describing Linux's tasks behavior in such way to be used in the reference for the development of real-time schedulers for Linux.

V. Related Work

Software verification is an active and bustling area of research, with many techniques involving the use of automata [14] 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 [15], TLC+ using TLA+ [16] and/or PlusCal models [17]. These formalisms can also handle verification of timing properties for real-time systems [18]. It

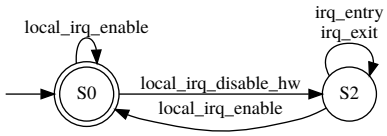


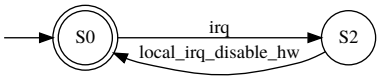
Fig. 10. IRQ handler starts after IRQs being disabled by the hardware.

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 [19], 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 [20], leading to the discovery of various design bugs in DynamoDB, S3, EBS, EC2 and other software components.

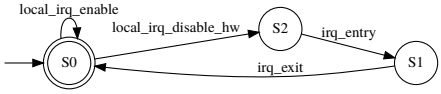
In this context, an area that is particularly challenging is the one of verification of an operating system kernel and its various components. Some works that addressed this problem include the one by Henzinger and others [21], who used control flow automata, combining existing techniques for state-space 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 [22] 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. [23] proposed the DDVerify tool, extending on the capabilities of BLAST and SLAM, e.g., supporting synchronization constructs, interrupts and deferred tasks.

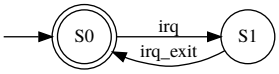
Another remarkable work is the `lockdep` mechanism [24]



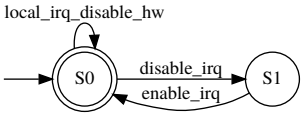
(a): an IRQ masks local interrupts before starting



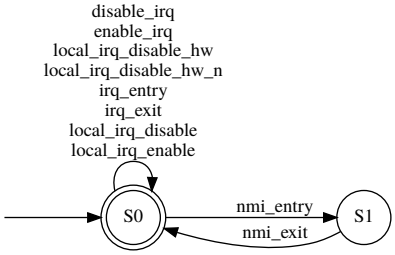
(b): IRQs remain masked when running a interrupt handler



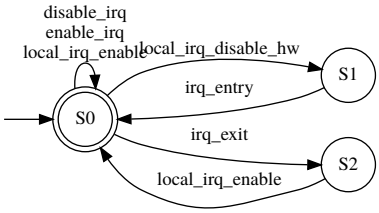
(c): NMIs are not preemptible



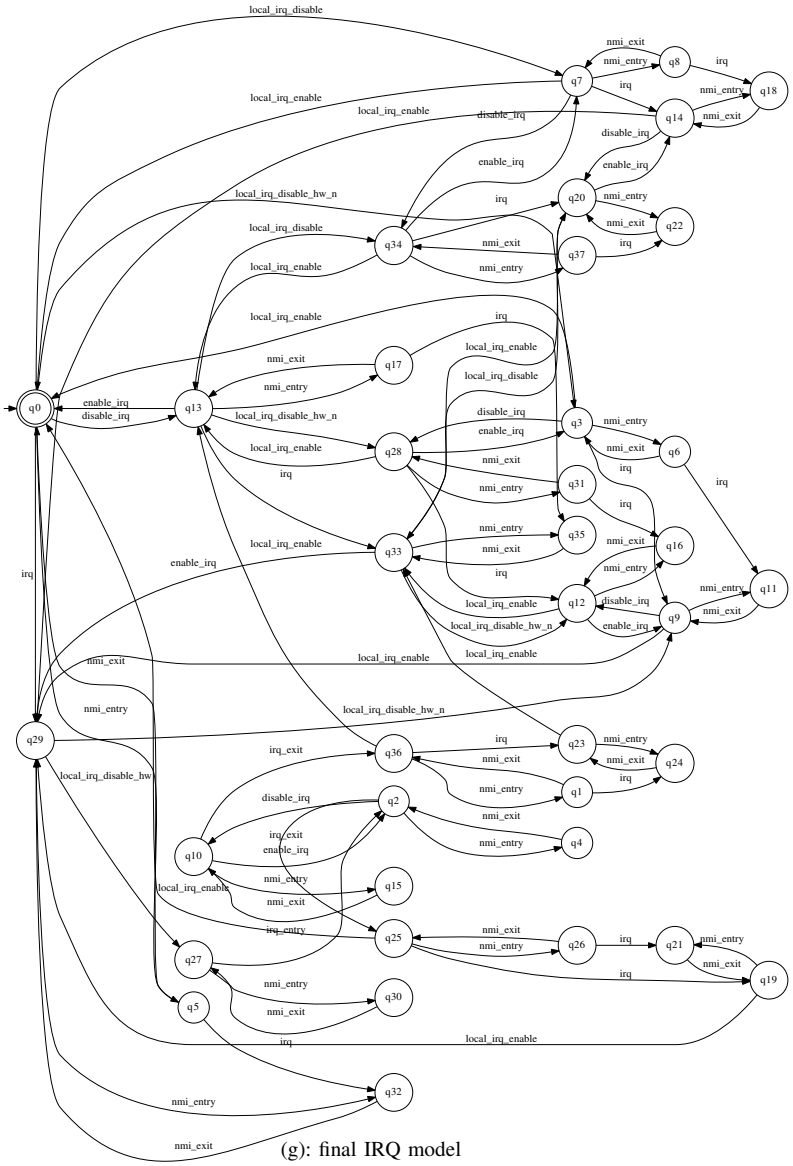
(d): a single IRQ disabled



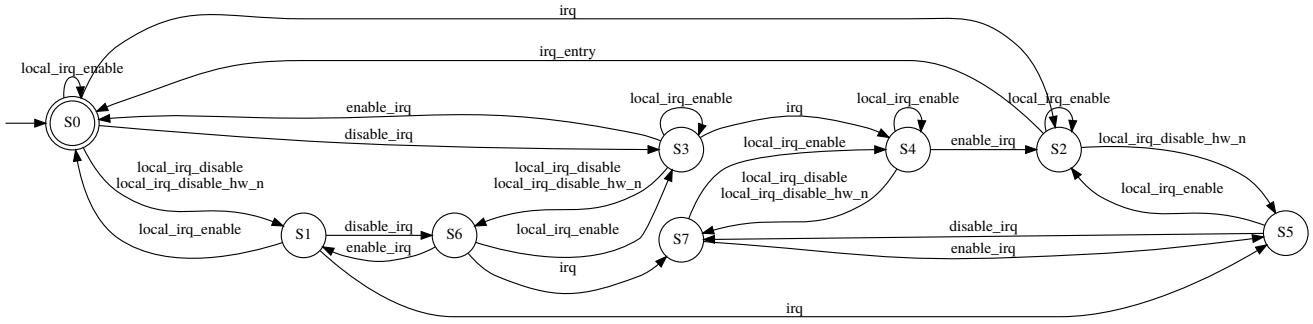
(e): interactions between NMIs and IRQs



(f): *disable_irq* and *enable_irq* cannot happen in some contexts



(g): final IRQ model



(h): specification of IRQ masking

Fig. 11. Elements of the automata-based model.

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 [25], 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 [26]. These findings were possible thanks to the simplicity of the seL4 micro-kernel features, e.g., semi-preemptability.

VI. Conclusions and Future Work

This paper describes our starting point for building an accurate automata-based behavioral model of IRQs, NMI and threads on Linux, for the purpose of enabling a more accurate analysis of the performance of the kernel and hosted real-time workloads than what traditionally done by practitioners. These use to rely exclusively on empirical data, which rarely represents worst-case conditions, whose timing is of great interest in using Linux in real-time use-cases.

The next step in this direction is the inclusion of events and states due to synchronization mechanisms used in IRQ and NMI handlers, still neglected in this preliminary work.

The ultimate goal of this research is allowing system engineers to build comprehensive timing models of embedded real-time applications running on Linux, capable of accounting for the interference across all Linux tasks types [6], from user-space threads up to IRQ and NMI handlers. Therefore, threads are planned to be modeled using the same approach, which is possible thanks to the composability of automata-based models, as described above. However, this will need a deeper investigation of kernel abstractions, to analyze the whole set of additional thread-specific events and transitions, and their interactions with the already identified ones.

Finally, a rigorous description of the Linux timing behavior using automata will hopefully help in an easier integration of theoretical results from real-time researchers with the needs of real-time Linux developers [27], [28].

References

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

[2] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[3] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[4] Red Hat, Inc., "Red Hat Enterprise Linux for Real Time," Available at: https://www.redhat.com/it/resources/red-hat-enterprise-linux-real-time [last accessed 28 March 2017].

[5] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*. San Jose, California: IEEE, September 2002.

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

[7] B. B. Brandenburg and J. H. Anderson, *A Comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 105–124. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92221-6_9

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computations*, 3rd ed. Prentice Hall, 2006.

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

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

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

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

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

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

[15] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[16] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994.

[17] ——, *The PlusCal Algorithm Language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 36–60.

[18] M. Abadi and L. Lamport, "An old-fashioned recipe for real time," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1543–1571, Sep. 1994.

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

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

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

[22] T. Ball and 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*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 1–3.

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

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

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

[26] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "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.

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

[28] T. Gleixner, "Realtime Linux: academia v. reality," *Linux Weekly News*, July 2010, available at https://lwn.net/Articles/397422/.