
UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Daniel Bristot de Oliveira

# Automata-based Formal Analysis and Verification of the Real-Time Linux Kernel

Pisa
2020

Daniel Bristot de Oliveira

# Automata-based Formal Analysis and Verification of the Real-Time Linux Kernel

Tese submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do título de Doutor em Engenharia de Automação e Sistemas.
Orientador: Prof. Rômulo Silva de Oliveira, Dr.
Coorientador: Prof. Tommaso Cucinotta, Dr.

Pisa
2020

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Bristot de Oliveira, Daniel
Automata-based Formal Analysis and Verification of the Real-Time Linux Kernel / Daniel Bristot de Oliveira ; orientador, Rômulo Silva de Oliveira, coorientador, Tommaso Cucinotta, 2020.
144 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2020.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. Tempo Real. 3. Kernel do Linux. 4. Metodos Formais. 5. Verificação. I. Silva de Oliveira, Rômulo. II. Cucinotta, Tommaso. III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Engenharia de Automação e Sistemas. IV. Título.

Daniel Bristot de Oliveira

**Automata-based Formal Analysis and Verification of the Real-Time Linux Kernel**

O presente trabalho em nível de doutorado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Marco Di Natale, Dr.
Scuola Superiore Sant'Anna

Prof. Giuseppe Lipari, Dr.
University of Lille

Prof. Rivalino Matias Junior, Dr.
Universidade Federal de Uberlândia

Prof. Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Doutor em Engenharia de Automação e Sistemas.

Documento assinado digitalmente
Werner Kraus Junior
Data: 01/07/2020 17:58:38-0300
CPF: 531.085.239-53

Prof. Werner Kraus Junior, Dr.
Coordenação do Programa de
Pós-Graduação

Documento assinado digitalmente
Romulo Silva de Oliveira
Data: 01/07/2020 16:45:30-0300
CPF: 407.617.200-06

Prof. Rômulo Silva de Oliveira, Dr.
Orientador

Pisa, 2020.

To my adivisors, Tommaso and Rômulo.

## ACKNOWLEDGEMENTS

In the first place, I would like to thank my friends and family for supporting me during the development of this research. In particular, I would like to thank Alessandra de Oliveira for motivating me to pursue a Ph.D. degree since I was a teenager. But also for allowing me to use her computer, which was “my” first computer. I also would like to thank Bianca Cartacci, for the unconditional support in the hardest days of these last years.

I would like to thank Red Hat, Inc. for allowing me to continue my academic research in concurrency with my professional life. I would also like to thank the professionals that motivated me to remain in the academy and supported this research, including Nelson Campaner, Steven Rostedt, Juri Lelli and Arnaldo Carvalho de Mello. In special, I would like to thank Clark Williams, not only for his effort in supporting my research at Red Hat but, mainly, for his adivises and personal support.

I would like to thank my Ph.D. colleagues from UFSC and Sant’Anna, in special Karila Palma Silva, Marco Pagani, Paolo Pazzaglia and Daniel Casini, all the members of the Retis Lab, and the administrave staff of the TeCIP institute at Sant’Anna and the PPGEAS at UFSC for the support in the cotutela agreement, in special for Enio Snoeijer.

Finally, and most importantly, I dedicate this work to my advisors. Thanks, Professor Tommaso Cucinotta and Professor Rômulo Silva de Oliveira for supporting me and guiding me in this journey, not only as an academic but as a human as well.

*“Este Cargo foi a Lua e voltou.”*
*(Unknown author.)*

## RESUMO

Sistemas de tempo real são sistemas computacionais em que o comportamento correto não depende apenas do comportamento lógico, mas também do comportamento temporal. Na teoria de sistemas de tempo real, um sistema é uma abstração, modelada usando um conjunto de variáveis que descrevem tão somente o comportamento temporal de seus componentes. O Linux é uma implementação de um sistema operacional (SO), que atualmente suporta algumas das abstrações fundamentais da teoria sistemas de tempo real. Apesar de todas as melhorias da última década, classificar o Linux como um SO de tempo real ainda é uma fonte de atrito entre as comunidades de desenvolvimento do Linux e da teoria de sistemas de tempo real. O principal motivo para este conflito está na análise empírica feitas pelos desenvolvedores do Linux, visto que na teoria, espera-se que as propriedades de um sistema derivem de uma descrição matemática de seu comportamento. Geralmente, um conjunto rigoroso de provas se faz necessário antes de obter qualquer conclusão sobre a previsibilidade do comportamento de tempo de execução de um sistema de tempo real. A diferença entre o Linux de tempo real e a teoria de tempo real nasce na complexidade do kernel do Linux. Isto se dá pelo grande esforço necessário para se entender todas as restrições impostas às tarefas de tempo real no Linux. O desafio é então descrever essas operações, usando um nível de abstração que remova a complexidade inerente ao código do kernel. Esta descrição deve utilizar-se de formato formal que facilite o entendimento da dinâmica do Linux pelos pesquisadores, sem ficar muito longe da maneira como os desenvolvedores observam e melhoram o Linux. Portanto, para melhorar a análise e verificação do Linux de tempo real, esta tese apresenta um modelo formal de sincronização de threads para o kernel PREEMPT_RT do Linux. Esse modelo é construído com base na teoria de autômatos, usando uma abordagem modular que permite a criação de um modelo baseado em um conjunto de subsistemas independentes e nas especificações que definem seu comportamento sincronizado. A tese também apresenta uma metodologia de modelagem, incluindo a estratégia de validação e ferramentas que comparam o modelo com a execução real do sistema. Esse modelo é usado como base para a criação de uma metodologia de verificação em tempo de execução para o kernel do Linux. O método de verificação em tempo de execução usa a geração automática de código do modelo para facilitar o desenvolvimento do sistema de monitoramento. Além disso, este método utiliza-se dos recursos de *tracing* dinâmico do Linux para permitir a verificação *on-the-fly* do sistema, com uma baixa sobrecarga do sistema. Por fim, a modelagem formal do comportamento do kernel é usada como uma etapa intermediária, facilitando o entendimento das regras e propriedades que regem o comportamento de temporização das tarefas do Linux. Por fim, um conjunto de especificações do modelo é utilizado como uma passo lógico na definição de um conjunto de regras e propriedades que definem o comportamento de tempo das tarefas do Linux. Essas propriedades são usadas na definição formal dos componentes e na composição da principal métrica usada pelos desenvolvedores do Linux de tempo real, a latência de escalonamento, no mesmo nível de abstração usado pelos pesquisadores de tempo real. Os valores para essas variáveis são então medidos e analisados por uma ferramenta proposta nesta tese.



**Palavras-chave**: Tempo real. Kernel do Linux. Métodos formais. Automatos. Verificação.

## RESUMO EXPANDIDO

Sistemas de tempo real são sistemas de computacionais que a corretude não depende apenas do comportamento lógico, mas também do comportamento temporal. Em outras palavras, a resposta a uma solicitação está correta apenas se o resultado lógico estiver correto e produzido dentro de um tempo limite ou *deadline*. Caso contrário, o sistema apresentará uma falha (BUTTAZZO, 2011).

Na teoria de escalonamento de tempo real um sistema é uma abstração, modelada usando um conjunto de variáveis que descrevem tão somente o comportamento temporal de seus componentes. Por exemplo, geralmente um sistema é composto por um conjunto de $n$ tarefas $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Cada tarefa é definida com um conjunto de variáveis definidas pelo *modelo de tarefa* do sistema. Por exemplo, no *modelo de tarefa esporádico*, cada tarefa $\tau_i$ é caracterizada por um tempo mínimo entre chegadas $P_i$ e um tempo de execução $C_i$. Essas tarefas são escalonadas em um conjunto de $m$ processadores $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$ e também podem compartilhar $q$ recursos $\sigma = \{\sigma_1, \sigma_2, ..., \sigma_q\}$ que requerem exclusão mútua. Nesse contexto, o principal objetivo do escalonador é atribuir, de alguma forma, o tempo dos processadores de $\rho$ e os recursos de $\sigma$ às tarefas de $\tau$ a fim de concluir todas as ativações de todas as tarefas de $\tau$ cumprindo as restrições de tempo de cada tarefa. A demonstração de que um determinado algoritmo de escalonamento cumpre esse objetivo é feita por métodos analíticos conhecidos como *análise de escalonamento*.

*Linux como um sistema operacional de tempo real*

O Linux é uma implementação de um sistema operacional (SO) que atualmente suporta algumas das abstrações fundamentais da teoria de escalonamento de tempo real. No Linux, do ponto de vista do escalonador, as tarefas são as *threads*. O objetivo dos escalonadores é atribuir o tempo dos processadores às *threads* do sistema. Com o SCHED_DEADLINE (LELLI et al., 2016), o Linux suporta o *modelo de tarefa esporádico*, permitindo a configuração do *tempo de execução* e do *período*, equivalentes ao $C_i$ e $P_i$, de uma determinada *thread*. Outra tecnologia importante que habilita o Linux para sistemas de tempo real é o PREEMPT_RT. Os desenvolvedores do PREEMPT_RT reformularam extensivamente o kernel do Linux para reduzir as seções de código que poderiam atrasar o escalonamento da *thread* de mais alta prioridade, com o objetivo de transformar o Linux em um sistema operacional preemptivo, aproximando o comportamento Linux ao dos sistemas preemptivos teóricos. Essa característica permite o uso do Linux para uma ampla variedade de aplicações que requerem baixa latência na resposta a uma solicitação. De fato, o Linux de tempo real tem sido usado com sucesso em vários projetos acadêmicos e industriais, desde infra-estruturas distribuídas e orientadas a serviços (VARDHAN et al., 2009), robótica (GUTIÉRREZ et al., 2018), redes de sensores (DUBEY; KARSAI; ABDELWAHED, 2009) e automação industrial (CUCINOTTA et al., 2009), até no controle de drones militares (CONDLIFFE, 2014) e sistemas de bolsas de valores (CORBET, J., 2010).

Apesar de todas as melhorias da última década, classificar o Linux como um sistema operacional de tempo real ainda é uma fonte de conflito entre os desenvolvedores do Linux e a academia. Em resumo, existem dois pontos principais de divergência: a

maneira como o Linux é analisado e as suposições usadas no desenvolvimento de novos algoritmos teóricos.

O uso frequente de suposições como *as tarefas são completamente independentes* e *as operações são atômicas* (BRANDENBURG, B. B.; ANDERSON, J. H., 2007; CALANDRINO et al., 2006) são consideradas uma simplificação excessiva do comportamento de um sistema moderno pelos desenvolvedores do kernel do Linux (GLEIXNER, 2010). No Linux, as tarefas podem interferir entre si de uma maneira não negligenciável. Estas interferências causam atrasos, como por exemplo, o atraso que sofre a tarefa de mais alta prioridade em começar a sua execução. Esse atraso é conhecido como *atraso de escalonamento* ou *latência de escalonamento*. Reduzir o atraso de escalonamento é a meta principal dos desenvolvedores do PREEMPT_RT. Os desenvolvedores do PREEMPT_RT medem este atraso usando o `cyclictest`. O `cyclictest` simula uma tarefa de mais alta prioridade e mede a diferença entre o tempo esperado para o início da sua execução e o tempo efetivo de início da execução, deste modo, medindo o atraso de escalonamento.

Apesar de ser útil, a forma com que o `cyclictest` mede o atraso de escalonamento é considerada uma simplificação excessiva do problema pela academia. Para a academia, as propriedades de um sistema de tempo real devem derivar de uma descrição matemática do seu comportamento. Além do mais, um conjunto rigoroso de provas é necessário para qualquer conclusão sobre a previsibilidade do comportamento de tempo de execução de um sistema.

A diferença entre o Linux de tempo real e a teoria de tempo real nasce na complexidade do kernel do Linux. O esforço necessário para entender todas as restrições impostas às tarefas de tempo real no Linux não é desprezível. Pode levar anos para um iniciante entender os aspectos internos do kernel do Linux. A complexidade é de fato uma barreira, não apenas para pesquisadores, mas também para desenvolvedores. O entendimento das primitivas de sincronização e como elas afetam o comportamento temporal das *threads* é fundamental para a definição do Linux nos termos da teoría de sistemas de tempo real. O desafio então é descrever essas operações, usando um nível de abstração que remova a complexidade do código no kernel, sem perder em detalhes.

*Métodos formais, modelos formais e runtime verification*

Os métodos formais consistem em uma coleção de técnicas matemáticas utilizadas na especificação de um sistema. As especificações de um sistema podem ser usadas para vários propósitos. Por exemplo, para fornecer uma prova rigorosa de que o programa implementado satisfaz algumas propriedades. A vantagem de usar a notação matemática é que ela remove a natureza ambígua da linguagem natural enquanto permite a verificação automática do sistema.

Apesar dos argumentos a favor do uso de métodos formais, a sua aplicação é geralmente restrita a setores específicos. As razões mais comumente citadas para tal são a complexidade da notação matemática usada nas especificações e os requisitos de espaço em memória e tempo de processamento necessários para a verificação de um sistema usando métodos formais.

Um modelo é uma *abstração* (um conjunto de equações matemáticas) de um sistema, enquanto um sistema é *algo real*, por exemplo, um amplificador, um carro, uma fábrica,

um corpo humano, etc. O processo de modelagem de um sistema envolve a definição de um conjunto de variáveis mensuráveis associadas ao sistema em questão. Freqüentemente, o modelo apenas aproxima o comportamento completo do sistema. A *adequação* é um aspecto crucial durante a definição do nível de abstração usado no modelo. A *adequação* de um modelo determina com que eficácia ele representa o comportamento subjacente do sistema (O'REGAN, 2017). Os termos sistema e modelo podem ser usados alternadamente quando um modelo *adequado* é encontrado.

Portanto, representa um desafio para este trabalho definir um nível de abstração *adequado* que, ao mesmo tempo, explique o comportamento em tempo de execução das tarefas de tempo real do Linux, evitando a conhecida limitação dos métodos formais.

Entre as técnicas disponíveis para a aplicação de métodos formais, foi escolhida a técnica de *runtime verification* (RV) devido à natureza do tempo de execução do modelo proposto. *Runtime verification* é um método leve, porém rigoroso, que complementa as técnicas de verificação exaustivas clássicas (como *model checking* e *theorem proving*) com uma abordagem mais prática. Ao preço de uma cobertura de execução limitada, que analisa uma única execução *trace* de um sistema, RV pode fornecer informações precisas sobre o comportamento em tempo de execução do sistema monitorado (FALCONE et al., 2018).

### Objetivo

Com o objetivo de melhorar a análise e verificação do kernel Linux de tempo real, esta tese propõe a criação de um modelo formal para as tarefa do Linux, incluindo as primitivas de sincronização que influenciam seu comportamento temporal. Este modelo deve permitir a verificação formal do comportamento *lógico* do sistema, bem como a análise formal do seu comportamento *temporal*.

### Contribuições

As contribuições desta tese para o estado da arte estão divididas em três etapas, descritas a seguir.

*Um modelo formal para as sincronização das threads do Linux de tempo real*

Esta etapa compreende o desenvolvimento de um modelo formal para as sincronização das *threads* do Linux de tempo real, e inclui:

- a definição de uma metodologia de modelagem usando a teoria dos autômatos e a abordagem modular;
- o modelo de sincronização das *threads* para o kernel PREEMPT_RT Linux;
- uma ferramenta de verificação de tempo de execução *offline* que pode ser usada na validação de modelo e na verificação de tempo de execução do kernel.

*Um método eficiente para a verificação formal do kernel do Linux*

Esta etapa compreende o aperfeiçoamento da técnica de verificação, possibilitando a verificação do comportamento lógico do sistema eficientemente em tempo de execução, e inclui:

- o desenvolvimento de uma abordagem dinâmica de verificação em tempo de execução para o kernel Linux, permitindo o monitoramento do sistema em produção;
- o desenvolvimento de uma ferramenta de geração automática de código do kernel a partir de um modelo de autômato;
- a análise de desempenho do método, demonstrando seu baixo impacto no desempenho do sistema.

*Análise temporal do comportamento do kernel do Linux*

A última etapa utiliza o modelo proposto anteriormente para extrair um conjunto de regras e propriedades que definem o comportamento de temporal do atraso de escalonamento. As principais contribuições desta etapa são:

- a definição de um conjunto de regras e propriedades sobre a dinâmica básica de sincronização do kernel Linux, necessária para a definição formal do atraso de escalonamento;
- a definição formal do conjunto de variáveis e da equação que define o atraso de escalonamento da tarefa de mais alta prioridade;
- uma ferramenta eficiente utilizada na medição e análise do atraso de escalonamento.

**Considerações finais**

O desenvolvimento de um modelo abstrato usando métodos formais foi uma resposta natural para desvendar a complexidade do Linux de uma maneira determinística. A simplicidade do formato dos autômatos e a flexibilidade da abordagem modular foram a combinação perfeita para conectar essas três áreas complexas: o kernel do Linux, a teoria dos sistemas em tempo real e os métodos formais. Com relação ao comportamento *lógico*, o formalismo dos autômatos tornou possível a verificação com baixo *overhead*. Com relação ao comportamento *temporal*, a definição formal do conjunto de variáveis e da equação que define o atraso de escalonamento da tarefa de mais alta prioridade permitiu a análise do Linux usando um método aceito pela comunidade acadêmica, algo que era fonte de atrito por mais de uma década (BRANDENBUG; ANDERSON, 2009). É importante observar que esse problema permaneceu aberto não por causa de uma *rivalidade*, mas por causa da complexidade de traduzir o comportamento do kernel para o formalismo de escalonamento de tempo real.

## ABSTRACT

Real-time systems are computing systems where the correct behavior does not depend only on the functional behavior, but also on the timing behavior. In the real-time scheduling theory, a system is an abstraction, modeled using a set of variables that describe the sole timing behavior of its components. Linux is an implementation of an operating system (OS), that nowadays supports some of the fundamental abstractions from the real-time scheduling theory. Despite all improvements of the last decade, classifying Linux as a real-time operating system (RTOS) is still a source of conflict between real-time Linux and scheduling communities. The central reasons for this conflict lie in the empirical analysis of the timing properties of Linux made by practitioners, as it is expected that the properties of a real-time system derive from a mathematical description of the behavior of the system. Generally, a rigorous set of proofs is required for any conclusion about the predictability of the runtime behavior of a real-time system. The gap between the real-time Linux and real-time theory roots in the Linux kernel complexity. The amount of effort required to understand all the constraints imposed on real-time tasks on Linux is not negligible. 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 formal format that facilitates the understanding of Linux dynamics for real-time researchers, without being too far from the way developers observe and improve Linux. Hence, to improve the real-time Linux runtime analysis and verification, this thesis presents a formal thread synchronization model for the PREEMPT_RT Linux kernel. This model is built upon the automata formalism, using a modular approach that enables the creation of a model based on a set of independent sub-systems and the specifications that define their synchronized behavior. The thesis also presents a viable modeling methodology, including the validation strategy and tooling that compares the model against the real execution of the system. This model is then used as the base for the creation of a runtime verification of the method for the Linux kernel. The runtime verification method uses automatic code generation from the model to facilitate the development of the monitoring system. Moreover, it uses the dynamic tracing features of Linux to enable *on-the-fly* verification of the system, at a low overhead. Finally, the formal modeling of the kernel behavior is used as an intermediary step, facilitating the understanding of the rules and properties that rule the timing behavior of Linux tasks. These properties are then used in the formal definition of the components and composition of the main metric used by the real-time Linux developers, the scheduling latency, in the same level of abstraction used by real-time researchers. The values for these variables are then measured and analyzed by a tool proposed in this thesis.



**Keywords**: Real-time Systems. Linux kernel. Formal methods. Automata. Runtime Verification.

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| API | application programming interface |
| APIC | advanced programmable interrupt controller |
| CCS | calculus communicating systems |
| CFS | completely fair scheduler |
| CI | continuous integration |
| CPU | central processing unit |
| CSP | communicating sequential processes |
| DES | discrete event system |
| EDF | earliest deadline first |
| GPOS | general-purpose operating system |
| INTs | interrupts |
| IRQ | maskable interrupt |
| IRQs | maskable interrupts |
| LDV | Linux driver verification |
| LKMM | Linux kernel memory consistency model |
| NMI | nonmaskable interrupt |
| OS | operating system |
| OSes | operating systems |
| PID | process identification number |
| RT | real-time |
| RTA | response-time analysis |
| RTOS | real-time operating system |
| RV | runtime verification |
| SAT | boolean satisfiability |
| SDV | static driver verifier |
| SLIC | specification language for interface checking |
| SMP | symmetric multiprocessing |
| WCET | worst-case execution time |

**CONTENTS**

# 1 INTRODUCTION

Real-time systems are computing systems where the correct behavior does not depend only on the functional behavior, but also on the timing behavior. In other words, the response to a request is only correct if the logical result is correct and produced within a *deadline*. Otherwise, the system will be showing a defect (BUTTAZZO, 2011).

In the real-time scheduling theory, a system is an abstraction, modeled using a set of variables that describe the sole timing behavior of its components . For example, usually a system is composed by a set of $n$ tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task is defined as a set of variables defined by the *task model* of the system. For instance, in *sporadic task model*, each task $\tau_i$ is characterized by a minimum time between arrivals $P_i$ and an execution time $C_i$. These tasks are scheduled on a set of $m$ processors $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$ and they may share $q$ resources $\sigma = \{\sigma_1, \sigma_2, ..., \sigma_q\}$ which require mutual exclusion. In this context, the main goal of the scheduling is to somehow assign the time of processors from $\rho$ and resources from $\sigma$ to tasks from $\tau$ in order to finish all jobs of all tasks from $\tau$ while meeting the timing constraints of each task. The demonstration that a given scheduling algorithm can achieve such a goal is done through analytical methods known as *schedulability analysis*.

Linux is an implementation of an OS that nowadays supports some of the fundamental abstractions from the real-time scheduling theory. In Linux, from the scheduling point of view, the most basic unit is the thread. It is then the goal of the schedulers to assign the time from the processors to the threads of the system. With SCHED_DEADLINE (LELLI et al., 2016), Linux supports the classical *sporadic task model*, allowing the configuration of the *runtime* and *period*, equivalent to the $C_i$ and $P_i$, of a given thread. Another key technology enabling Linux for real-time systems is the PREEMPT_RT. The PREEMPT_RT developers have extensively reworked the Linux kernel to reduce the code sections that could delay the scheduling of the highest-priority thread, aiming to transform Linux into a *preemptive* operating system, trying to approximate Linux to the theoretical *preemptive* system. This characteristic allows the usage of Linux for a wide range of workloads that require low latency in the response for a request. Indeed, real-time Linux has been successfully used throughout a number of academic and industrial projects as a fundamental building block of real-time systems, from distributed and service-oriented infrastructures for multimedia (VARDHAN et al., 2009), robotics (GUTIÉRREZ et al., 2018), sensor networks (DUBEY; KARSAI; ABDELWAHED, 2009) and factory automation (CUCINOTTA et al., 2009), to the control of military drones (CONDLIFFE, 2014) and distributed high-frequency trading systems (CORBET, J., 2010).

## 1.1 LINUX AS A REAL-TIME OPERATING SYSTEM

Despite all improvements of the last decade, classifying Linux as a RTOS is still a source of conflict between real-time Linux and scheduling communities. In summary, there are two central sources of divergence: the way that Linux is analyzed, and the assumptions used in the development of new theoretical real-time algorithms.

From the real-time Linux community, the frequent use of assumptions like *tasks are completely independent*, and *operations are atomic* (BRANDENBURG, B. B.; ANDERSON, J. H., 2007; CALANDRINO et al., 2006) are considered an oversimplification of Linux behavior by kernel developers (GLEIXNER, 2010).

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. Those in-kernel operations are necessary because of the non-atomic nature of a sophisticated OS like Linux. For example, the arrival of the highest priority thread will not atomically load its context 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 that define the thread and are used by the scheduling algorithms. Moreover, interrupts must be disabled to avoid race conditions with interrupt handlers. Hence, delays in scheduling and interrupt handling are created during the activation of a thread (OLIVEIRA; OLIVEIRA, 2016). Only after the end of the activation process of a thread is that the system will be able to call the scheduling functions, that will eventually allow the thread to start running. This delay is known as the *scheduling latency*, and it drives the development of the PREEMPT_RT. The PREEMPT_RT developers measure it by using the `cyclictest` tool.

The `cyclictest` tool works by creating a thread that periodically sets an external timer in the future, and suspends its execution waiting for this timer occurrence. When the timer awakens the thread, the thread computes the time difference between the expected activation time and the actual time. By creating one thread per CPU, configured as the highest priority, `cyclictest` is used in practice to measure the *scheduling latency* of each CPU of the system. Maximum observed latency values generally range from a few microseconds on single-CPU systems to 250 us on non-uniform memory access systems, which are acceptable values for a vast range of applications with sub-millisecond timing precision requirements.

Despite useful, the *scheduling latency*, as measured by `cyclictest`, is considered an oversimplification of the problem by the academy. An interesting discussion about this metric, from the real-time scheduling theory standing point, is presented in (BRANDENBUG; ANDERSON, 2009). In that work, the authors argue that the dual notion of correctness of real-time systems, the *logical* and *timing* correctness, is for-

mally addressed in the academy. The properties of a real-time system must derive from a mathematical description of the behavior of the system, and a rigorous set of proofs are required for any conclusion about the predictability of the runtime behavior of a system. It is clear that `cyclictest` results do not provide such a level of certainty, justifying the argument that there are still further steps to be taken toward a better acceptance of Linux as a RTOS.

The gap between the real-time Linux and real-time theory roots in the Linux kernel complexity. The amount of effort required to understand all the constraints imposed on real-time tasks on Linux is not negligible. It might take years for a newcomer to understand the internals of the Linux kernel. The complexity is indeed a barrier, not only for researchers but for developers as well. The understanding of the synchronization primitives and how they affect the timing behavior of a thread is fundamental for the definition of Linux in terms of real-time scheduling.

The challenge is then to describe such operations, using a level of abstraction that removes the complexity of 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 usage of mathematical notation can remove the ambiguous nature of natural language, enabling the application of a more sophisticated analysis of the runtime behavior of Linux. To improve the state-of-art, we believe that a mathematical model, based on well-defined criteria that describe, in a deterministic way, the Linux behavior is required, which leads this research to the application of another area from the computer science: formal methods.

## 1.2 FORMAL METHODS

Formal methods consist of a collection of mathematical techniques to rigorously state the specification of a system. The specifications of a system can then be used for multiple purposes. For example, to provide a rigorous proof that the implemented program satisfies some properties. The advantage of using mathematical notation is that it removes the ambiguous nature of natural language while enabling automatic verification of the system.

Despite the arguments in favor of formal methods, its application is generally restricted to specific sectors. The most commonly mentioned reasons for that are the complexity of the mathematical notation used in the specifications and the limitations of computational space and processing time required for the verification of a system using formal methods. Regarding performance, a common problem faced by practitioners in the usage of formal methods is the *state explosion* problem. As exemplified in (CLARKE; EMERSON; SIFAKIS, 2009), the composition of a model of a system with $n$ tasks, with each task with $m$ states, will result in a model with $m^n$ states. Moreover, it is also necessary to consider the level of expressiveness of the specification notation.

Generally, a more concise notation is likely to be more efficient than feature-rich notation. Regarding the complexity, it is a challenge for this work to find a formal specification notation that, at the same time, can be easily interpreted by kernel developers, useful to demonstrate the timing behavior of tasks, and able to verify the Linux kernel behavior appropriately.

### 1.2.1 Formal models

A model is an *abstraction* (a set of mathematical equations) of a system, whereas a system is *something real*, e.g., an amplifier, a car, a factory, a human body, and so on. The process of modeling a system involves defining a set of measurable variables associated with the given system. Often, the model only approximates the complete behavior of the system. The *adequacy* is a crucial aspect during the definition of the level of abstraction used in the model. The *adequacy* of a model determines how effectively it represents the underlying behavior of the system (O'REGAN, 2017). The terms system and model can be interchangeably used when an *adequate* model is found.

It is then a challenge for this work to define an *adequate* level of abstraction that, at the same time, explains the runtime behavior of real-time tasks of Linux while avoiding the well-known limitation of formal methods.

Regarding the level of abstractions, the developers of Linux observe and debug the timing properties of Linux using the tracing features present in the kernel (ROSTEDT, S., 2011; SPEAR; LEVY; DESNOYERS, 2012; TOUPIN, 2011; BRANDENBURG, Bjorn B.; ANDERSON, James H., 2007). They interpret a chain of events, trying to identify the states that cause large *scheduling latencies* delays, and then try to change kernel algorithms to avoid such delays. For instance, they use `ftrace` (ROSTEDT, Steven, 2010) or `perf`[1] to trace kernel events like interrupt handling, wakeup of a new thread, context switch, etc., while `cyclictest` runs.

The notion of *events*, *traces* and *states* used by developers are common to discrete event system (DES). The admissible sequences of events that a DES can produce or process can be formally modeled through a *language*, using the automata formalism. One of the key features of the automata formalism is its directed graph or state transition diagram representation. This graphical representation hides the complexity of the language and was very welcome by the Linux kernel community during discussions about the development of this thesis. Moreover, the automata are amenable to composition operations, allowing a modular development of the model. The modular approach allows the model development based on a set of subsystems and the specifications that synchronizes these subsystems. Moreover, it enables a set of analyses as well, considering the finite-state case, including runtime verification (RV).

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

### 1.2.2 Runtime verification

Among the techniques available to the application of formal methods, RV was chosen because of the runtime nature of the proposed model. RV presents a lightweight, yet rigorous, method that complements classical exhaustive verification techniques (such as model checking and theorem proving) with a more practical approach. At the price of a limited execution coverage, that analyses a single execution *trace* of a system, RV can give precise information on the runtime behavior of the monitored system (FALCONE et al., 2018).

In this context, a monitor is a program that can parse both the formal specification and the trace of the system, connecting them. The monitor then verifies the runtime behavior of the system by comparing the trace of its execution against the formal specification, reporting an error in the case of a model violation.

Monitors can be classified as *offline* and *online* monitors. *Offline* monitors process the traces generated by a system after the occurrence of the events, generally by reading the trace execution from a permanent storage system. *Online* monitors process the trace during the execution of the system.

## 1.3 GOALS OF THIS THESIS

Aiming to improve the runtime analysis and verification of the real-time Linux kernel, this thesis proposes the creation of a formal model of the Linux task, including the synchronization primitives that influence their timing behavior. This model should enable the formal verification of the *logical* behavior of the system, as well as the formal analysis of its *timing* behavior.

This primary goal is divided into sub-goals, presented as follows.

### 1.3.1 A formal model for Linux tasks

This thesis proposes the creation of a formal model of the Linux task, including the synchronization primitives that influence their timing behavior. A fundamental step in the development of a model is the precise definition of its purpose. Thus, the purpose of creating an explicit model of the Linux tasks are:

- to promote the unambiguous understanding of the system from its formal specifications;
- to enable the validation of the specifications against the real execution of the system: a fundamental step to strengthen the trustworthy and accuracy of the model;
- to formally verify the runtime behavior of the system.

Given the foreseeing complexity of such a model, a modeling methodology should be carefully defined, in such a way to avoid the well-known limitations of the practical application of formal methods. The methodology should also comprise the development of an automatic model validation tool.

### 1.3.2 Runtime verification of the logical behavior of Linux

Giving that the model aims to formalize the runtime behavior of tasks of Linux, a practical formal verification method should be developed employing RV techniques. To be practical, the verification method should be able to connect the *model* and the *trace* without requiring extensive manual development of the monitoring tool, maximizing the automatic code generation from the model. The monitor should also be able to generate output that helps the developers to debug the conditions that caused the undesired behavior of the system. It is known that the trace of the system impacts the timing behavior of the system, so to be practical, the verification software should also be efficient. By efficient, it means that the monitoring tool must minimize the overhead, not only to viable values but in such a way to avoid impacting the timing aspects of the system as much as possible.

### 1.3.3 Runtime analysis of the timing behavior of Linux

As the last goal, the formal modeling of the kernel behavior is used as an intermediary step, facilitating the understanding of the rules and properties that rule the timing behavior of Linux tasks. To demonstrate the effectiveness of the goal, part of the Linux behavior should be explained with a level of formalism and granularity similar to the practices in the real-time scheduling theory. To be effective, the explanation should result in a set of practical variables that could be measured and serve as the base for improving the real-time features of Linux. Finally, tools to measure the value for these variables must be developed, also taking into consideration the efficiency of the approach in such a way to avoid impacting the timing aspects of the system as much as possible.

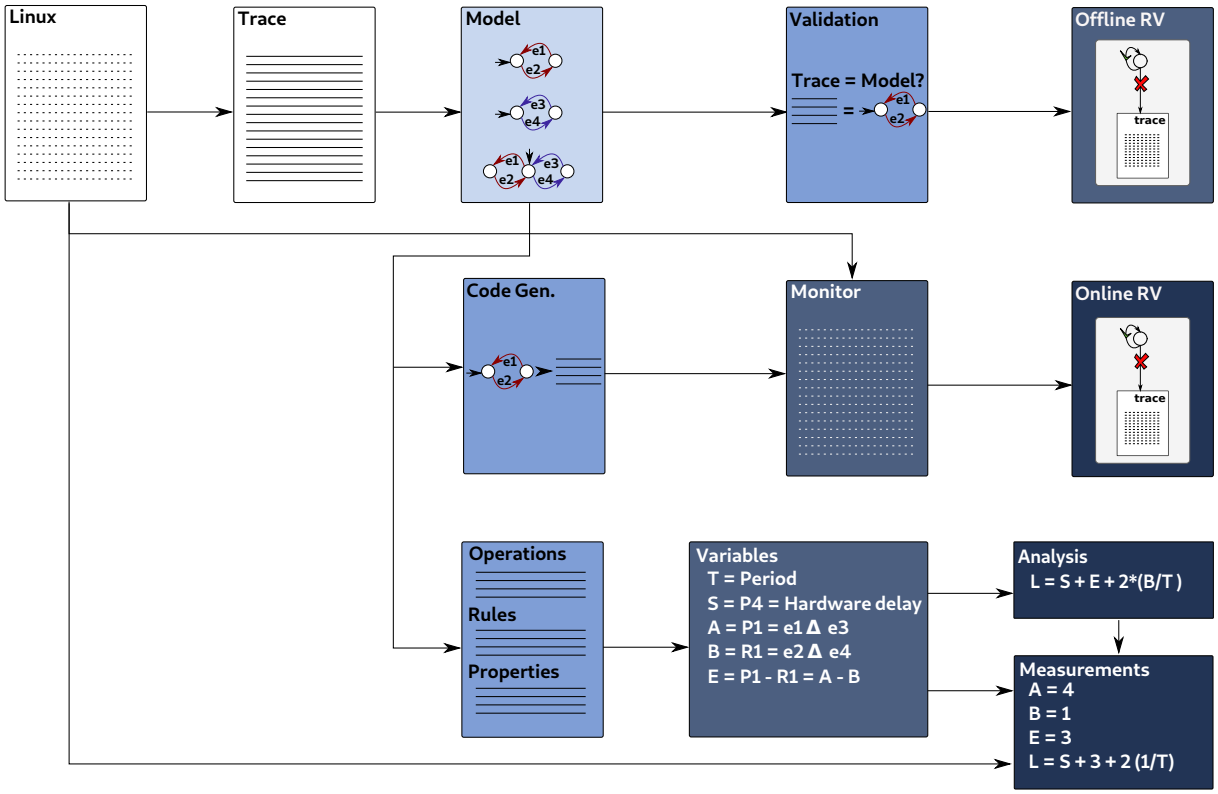
## 1.4 CONTRIBUTIONS OF THIS THESIS

The original contributions of this thesis can be organized in three stages, as presented in Figure 1. The major contributions of each stage are detailed in the next sections.

### 1.4.1 First stage: modeling the timing behavior of tasks on real-time Linux

This stage focused on the creation of the *thread synchronization model for the PREEMPT_RT Linux kernel*, and the main contributions are:

Figure 1 – Thesis approach and contributions.




- the definition of a formal modeling methodology using the automata theory and the modular approach;
- the thread synchronization model for the PREEMPT_RT Linux kernel;
- an *offline* runtime verification tool that can be used both in the model validation and the runtime verification of the kernel.

### 1.4.2 Second stage: efficient runtime verification for the Linux kernel

This stage focused on the development of an *efficient runtime verification for the Linux kernel*, based on the modeling formalism presented in the previous stage.

The main contributions of this phase are:

- the development of a dynamic online runtime verification approach for the Linux kernel, enabling the monitoring of the system *on-the-fly*;
- the development of an automatic kernel code generation from an automaton model;
- the performance analysis of the method, demonstrating its low impact on the performance of the system.

### 1.4.3 Third stage: formal definition of the latency components

The last stage leverages the thread model to extract a set of rules and properties that defines the timing behavior of *scheduling latency*. The main contributions of this phase are:

- the definition of a set of rules and properties about the basic synchronization dynamics of the Linux kernel, necessary for the formal definition of the latency;
- the definition of a set of variables and the subsequent *scheduling latency* analysis;
- an efficient *scheduling latency* measurement tool.

## 1.5 ORGANIZATION OF THIS THESIS

The next chapter presents the background topics, covering from essential elements from the real-time theory to the runtime verification methods, passing by an explanation about real-time Linux and its synchronization methods, formal methods, and formal models. A set of related work is then presented in Chapter 3.

The development of the thesis is presented in Chapters 4, 5 and 6, where the development of the *thread synchronization model for the PREEMPT_RT Linux kernel*, the *efficient runtime verification for the Linux kernel* and the formal definition of the *scheduling latency* are, respectively, described in details.

Finally, the conclusions, set of publications, and future work are presented in Chapter 7.

# 2 BACKGROUND

This chapter presents the background information useful for a more comprehensive understanding of the concepts exposed in this thesis. It is divided into three main sections. The first section covers useful definitions from the classical real-time systems theory. The second section describes Linux as a real-time operating system, from basic definitions of tasks and the mutual exclusion mechanism to tracing features present in the kernel, and an early tentative of framing Linux in the existing real-time research literature. The third section presents the fundamentals of formal methods, the modeling of DES using the automata theory, and runtime verification. Finally, Section 2.4 presents the final remarks, summarizing the background while pointing to the directions that will be followed in the next chapters.

## 2.1 REAL-TIME SYSTEMS

Real-time systems are computing systems where the correct behavior does not depend only on the functional behavior, but also on the timing behavior. In other words, the response to a request is only correct if the logical result is correct and produced within a given *deadline*. Otherwise, the system will be showing a defect (BUTTAZZO, 2011). Real-time systems can be classified according to the effect of a timing defect. *Critical* or *hard real-time systems* are those systems for which a timing defect may result in catastrophic consequences. For instance, a failure in the hard real-time braking system of a car can potentially cause loss of lives. In contrast, *non-critical* or *soft real-time systems* are those systems in which temporal requirements describe the desired behavior. Still, if not met, they do not invalidate the results nor have catastrophic consequences, although the application's utility is reduced (LIU, 2000).

### 2.1.1 Real-time scheduling theory

In real-time systems, the most basic scheduling unit is the *task*. A task is a computation that must be sequentially executed by a processor. When a task becomes ready to run, it is *dispatched*, and so it becomes *active*. Any time a task is activated, it is said that a new *job* of the task was dispatched. A job can be either *running* when the scheduling algorithm selects it to run or *ready* to run when it is ready to run but not running. Jobs ready but still waiting to run are maintained in the *ready queue*. Generally, the *i*th task of a system is denoted by $\tau_i$.

Real-time tasks are also characterized by the activation pattern of their jobs. *Periodic* tasks are those in which jobs are dispatched at a constant rate, with each activation taking place after a fixed period of time. *Sporadic* tasks are those in which a new job arrives after, at least, a minimum time after the arrival of the previous job of

the same task, i.e., they are characterized by a pre-specified minimum inter-arrival time among consecutive jobs. Those tasks that do not have a regular activation pattern are said to be *aperiodic* tasks.

Tasks are also characterized by their *deadline*. A task has *implicit deadline* if its deadline is equal to its activation period or minimum inter-arrival time. A task has *constrained deadline* if its deadline is less than or equal to the activation period or minimum inter-arrival time of the task. Otherwise, the deadline is said to be *arbitrary*.

Individually, each real-time task $\tau_i$ has a set of timing values that specifies its behavior according to the previous patterns. For example:

- **Period** $P_i$: the activation pattern of task $\tau_i$;
- **Relative deadline** $D_i$: the relative deadline (the specification) of task $\tau_i$;
- **Computation time** $C_i$: the computation time of the task $\tau_i$, without interruptions. The $C_i$ value often represents the worst-case execution time (WCET) of a task.

After activated, some runtime parameters of the $m$th job the task $\tau_i$ are also used to describe its execution. For instance:

- **Arrival time** $a_{i,m}$: is the time instant when a job becomes ready for execution;
- **Release time** $r_{i,m}$: is the time instant when a job is queued in the ready queue;
- **Absolute deadline** $d_{i,m} = a_{i,m} + D_i$: is the time instant when the job should already have finished its activation, i.e., it is the runtime parameter of a task;
- **Starting time** $s_{i,m}$: is the time instant when a job starts is execution;
- **Finishing time** $f_{i,m}$: is the instant when a job finishes its execution;
- **Response time** $R_{i,m}$: is the difference between the finishing time and the arrival time: $R_{i,m} = f_{i,m} - a_{i,m}$.

When the number of tasks is higher than the number of processors, central processing unit (CPU) time needs to be shared among concurrent tasks. Tasks are selected to run on processors according to a *scheduling policy*. The set of rules that define which task runs on which CPU at a given time is called *scheduling algorithm*.

In the real-time scheduling theory a system is modeled as a set of $n$ tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. These tasks are scheduled on a set of $m$ processors $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$ and they may share $q$ resources $\sigma = \{\sigma_1, \sigma_2, ..., \sigma_q\}$ which require mutual exclusion. In this context, the main goal of the scheduler is to somehow assign the time of processors from $\rho$ and resources from $\sigma$ to tasks from $\tau$ in order to finish all jobs of all tasks from $\tau$ while meeting the timing constraints of each task.

A schedule is said to be *feasible* when all tasks are able to accomplish their jobs while respecting all constraints. A task set is said to be *schedulable* if there is at least one algorithm that can produce a *feasible schedule*.

If during the execution, a ready task is interrupted to give place to another task, the running job is said to be *preempted*. The *preempted job* is placed in the ready queue, while another job becomes the running one. When it is possible to preempt a job at any point, the scheduler is said to be *preemptive*. In contrast, when the scheduler cannot interrupt at an arbitrary point, the scheduler is said to be *non-preemptive* or *cooperative*. The switch between one job and another is named *context switch*.

### 2.1.2 Response-time analysis

In order to guarantee timing correctness while executing real-time tasks, one has to know whether a given set of tasks will complete within their respective deadlines. There are several analytic methods to obtain this guarantee, depending on the execution model of the system. As shown later in Chapter 6, part of the Linux behavior can be seen as a real-time system that schedules tasks using a fixed-priority scheduler, together with several mutual exclusion protocols. For a theoretical system with this characteristic, it is possible to verify the schedulability using the method of response-time analysis (RTA) (JOSEPH; PANDYA, 1986; LEHOCZKY; SHA; DING, 1989; AUDSLEY et al., 1993).

The RTA considers that a system is composed of a set of *n sporadic* tasks $\tau$, which in turn are described by a set of algebraic variables related to their timing behavior. In addition to the period $P_i$, the relative deadline $D_i$, and the worst-case execution time $C_i$, the RTA also considers the *release jitter* and the *blocking time* of a task, where:

- **Release Jitter** $J_i$: is the delay at the beginning of the execution of a task $i$;
- **Blocking time** $B_i$: is the worst-case blocking time, which is a delay caused by a lower priority task, generally because the lower priority task holds some resources required by the task $i$.

Based on these variables, the RTA is used to define the value of $I_i$ and $W_i$, where:

- **Interference** $I_i$: is the interference caused by higher priority;
- **Busy window** $W_i$: is the busy-window.

The interference $I_i$ of a task $\tau_i$ is the sum of the computation time of tasks in the set $hp(i)$ that were activated during the busy period of task $i$, where $hp(i)$ is the set of tasks with priorities higher than $\tau_i$. Formally:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil . C_j \quad (1)$$

The busy period $W_i$ of a task $\tau_i$ corresponds to the sum of its computational time, blocking time, and interference. It is given by Equation 2.

$$W_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil . C_j \quad (2)$$

It is essential to notice that $W_i$ appears on both sides of the equation, due to its use in the definition of the interference. This dependence implies the use of an iterative method to determine $W_i$. In this case, the initial value of $W_i$ is the worst-case execution time $C_i$, and Equation 2 is used interactively $x$ times, until $W_i^{x+1} = W_i^x$ or $W_i^{x+1} > D_i$.

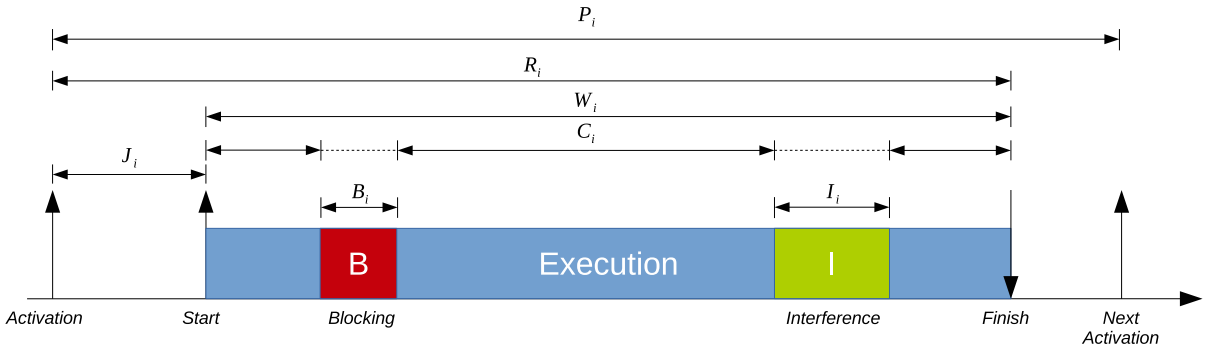
After obtaining $W_i$, it is used to determine the response time $R_i$, that equals to its busy period $W_i$ added to its release jitter $J_i$:

$$R_i = W_i + J_i \quad (3)$$

A system is said to be schedulable if, for every task $\tau_i$, the maximum response time $R_i$ is less than or equal to its deadline $D_i$.

A common way to represent the behavior of a real-time task is using a *time-line* format. Figure 2 shows how each abstraction used in the response-time analysis composes the response time of a real-time task.

Figure 2 – Response-time analysis abstractions in a timeline.




## 2.2 LINUX

Linux is a full-featured general-purpose OS, that has been adapted and improved over the last decade to be used as a RTOS, becoming a viable option as a RTOS for

many relevant workloads. This section shortly presents the background for real-time Linux, along with a description of the tracing subsystem, that is frequently used in the analysis of the timing behavior of the kernel.

### 2.2.1 Linux as a real-time operating system

Real-time Linux has been a recurring topic in both research (CALANDRINO et al., 2006; PALOPOLI et al., 2009; BRANDENBURG; GÜL, 2016) and industry (DUBEY; KARSAI; ABDELWAHED, 2009; GUTIÉRREZ et al., 2018; CUCINOTTA et al., 2009; CORBET, J., 2010), for more than a decade now. From the different initiatives for enabling real-time Linux, such as RTAI (MANTEGAZZA et al., 2000) and Xenomai (GERUM, 2004; BROWN; MARTIN, 2010), the PREEMPT_RT became the *de facto* standard. The difference between the other approaches and the PREEMPT_RT is that, rather than trying to run a real-time OS in parallel with Linux, the PREEMPT_RT aims to transform the Linux kernel into a RTOS.

Linux has three preemption models for kernel space activities. The preemption models range from the *non-preemptive* mode, in which the kernel code schedules only on predefined preemption points, to the *preemptive* mode, in which the kernel code is preemptive by default, unless when the preemption is explicitly disabled. It is worth noting that the user-space code is always preemptive, independently of the preemption model.

In addition to the preemption models present in the vanilla kernel[1], The PREEMPT_RT patchset adds the *fully-preemptive* model, which improves the *preemptive* mode. Regarding the *fully-preemptive* mode, Linux developers have extensively reworked the Linux kernel to reduce the code sections that could delay the scheduling of the highest-priority thread. While PREEMPT_RT improves the responsiveness, it reduces the throughput of the system, and that justifies the maintenance of the multiple preemption modes on Linux.

The `cyclictest` is the primary tool adopted in the evaluation of the *fully-preemptive mode* of PREEMPT_RT Linux (CERQUEIRA; BRANDENBURG, 2013), and it is used to compute the time difference between an expected activation time and the actual start of execution of the high-priority thread running on a CPU. By configuring the measurement thread with the highest priority and running a background task set to generate disturbance, `cyclictest` is used in practice to measure the *scheduling latency* of each CPU of the system. Maximum observed latency values generally range from a few microseconds on single-CPU systems to 250 us on non-uniform memory access systems, which are acceptable values for a vast range of applications with sub-millisecond timing precision requirements. In this way, PREEMPT_RT Linux closely fulfills theoretical

[1] Vanilla kernel is the Linux kernel as-is from its main repository.

fully-preemptive systems assumptions that consider atomic scheduling operations, with neglectable overheads.

In the *fully-preemptive mode*, there are three different execution contexts: non-maskable interrupt (NMI), maskable interrupts (IRQs), and threads. Both the NMI and the IRQs are asynchronous interrupts (INTs), i.e., mechanisms used to deliver events coming either from external hardware or by code running on other CPUs via inter-processor interrupts. The interrupt controller manages interrupts, both queueing and dispatching one NMI per-CPU and multiple IRQs. For each CPU, the NMI is the highest-priority interrupt, so it postpones and preempts IRQs. As a design choice, Linux (in the *fully-preemptive mode*) handles IRQs with IRQs disabled. Hence an maskable interrupt (IRQ) cannot preempt another IRQ. Threads have no control over the NMI, but they can delay the execution of IRQs by temporarily disabling (masking) them. Note that when IRQs are masked, their occurrence is anyway stored in the interrupt controller. When IRQs are enabled again, the kernel is notified about the occurrence of such interrupts, and they are executed as soon as possible, possibly preempting the currently executing thread. Given the potential interference on threads execution, one of the design goals of the PREEMPT_RT was to reduce the code that executes in the interrupt context to the bare minimum, by moving most of it to thread context (OLIVEIRA; OLIVEIRA, 2016).

Despite the existence of different memory contexts in which a regular program can run, like user programs in the kernel-space, e.g., during a system call, kernel threads, or the process context in the user-space, from the scheduling viewpoint, they are all threads. Linux has not one but five schedulers, which are provided to fit the requirements of the manifold different applicative scenarios in which Linux is used. When invoked, the set of schedulers are queried in a fixed order. The following schedulers are checked:

- STOP_MACHINE: a pseudo-scheduler used to execute kernel facilities;
- SCHED_DEADLINE (LELLI et al., 2016): An earliest deadline first (EDF) like real-time scheduler;
- SCHED_FIFO and SCHED_RR: the fixed-priority real-time scheduler;
- SCHED_OTHER: the completely fair scheduler (CFS);
- IDLE: a pseudo-scheduler that runs the *idle thread*.

Since schedulers are queried in order, the querying order implements the first level of priority among the threads handled by each scheduler. Every time the scheduling-related code is executed by the kernel, the highest-priority thread is selected for a context switch. When no ready threads are available, the IDLE scheduler returns the *idle thread*, a particular thread always ready to run. For simplicity, we refer hereafter

with the term *scheduler* when mentioning the kernel code handling all the scheduling operations related to all five schedulers. The scheduler is called either voluntarily by a thread leaving the processor, or involuntarily, to cause a preemption. Any thread currently executing can postpone the execution of the scheduler while running in the kernel context by either disabling preemption or the IRQs.

### 2.2.2 Task abstraction and context synchronization

As mentioned in the previous section, there are three main contexts in which code can run in the PREEMPT_RT. Two of them are INTs: The NMI, the IRQ, and the other one is the thread context.

Interrupts are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing thread that requires the attention of a processor. An interrupt occurrence results in a forced transfer of execution from the currently running thread to the interrupt handler. The interrupt handling scheduling is managed by the interrupt controller, provided by hardware[2]. For instance, in the Intel processors, the prioritization of IRQs is done in the local advanced programmable interrupt controller (APIC) of each CPU.

A thread can postpone the execution of an IRQ by temporarily disabling (or masking) the IRQs in a given processor. Linux API provides two main methods to disable IRQs. The first is through the functions `local_irq_disable()` and `local_irq_enable()`. The second is through the functions `local_irq_save()` and `local_irq_restore()`, these functions (actually macros) save the processor flags only to be restored lately, which allows nesting calls to disable/enable interrupts (LOVE, 2010). The processor can also temporarily disable IRQs without OS intervention. Indeed, the processor temporarily disables all the maskable interrupts before dispatching an interrupt handler. In the PREEMPT_RT case, the interrupt handler continues executing with the interrupts disabled until the end return of the handler. Although the IRQ handlers context exists in the PREEMPT_RT, for most of them, their function is no longer to deal with the hardware but to wake up the kernel threads that execute the code of interrupt handlers. For the sake of completeness, it is worth mentioning that IRQs and threads cannot postpone an NMI execution.

Linux processes are composed of a memory context and a set of one or more threads. Generally, a thread runs in the process memory context in user-space. However, when a thread makes a *system call* or causes an *trap*, for example, with a *page fault*, it changes its execution context from user-space to kernel-space, executing the kernel code in kernel-space on behalf of the process (CORBET; RUBINI; KROAH-HARTMAN, 2005). There is also the particular case of threads that only run in kernel-space, the so-called kernel threads, or `kthreads`.

[2] See Intel® 64 and IA-32 Architectures Software Developer's Manual.

Threads are activated by events that change their state in the scheduler, from *sleeping* to *ready to execute*. Ideally, when a lower priority thread awakens a higher priority thread, the scheduler should be called and promptly start the execution of the thread with higher priority. However, when preemption or interrupts are disabled, the lower priority thread runs until preemption or IRQ is enabled again, and the scheduler can decide to run a thread with higher priority. Differently from interrupts, Linux is responsible for scheduling threads. The scheduling decisions and the context switch takes place inside the `__schedule()` function. The `__schedule()` function consults all the schedulers, as described previously, and changes the context to the next selected thread as needed.

The preemption of a processor can be disabled via the *preempt_disable()* function, and then enabled again with the function *preempt_enable()*. For each call of *preempt_disable()* there should be a call to *preempt_enable()*. These calls can be nested, the number of nesting can be retrieved with the function *preempt_count()* (CORBET; RUBINI; KROAH-HARTMAN, 2005). The function *preempt_enable()*, when called, checks whether the preemption counter is 0, that is, whether the preemption system will be active again. When enabling preemption, if a higher priority task may be ready to run, the scheduling routine will be called.

### 2.2.3 Mutual exclusion

The Linux kernel has several mechanisms for mutual exclusion. There are two reasons for these different mechanisms. The first comes from the needs of the diverse execution contexts, which have different constraints. For instance, in interrupt context, the code cannot use methods that put the interrupt handler to sleep, while a thread can sleep, allowing other threads to execute while the blocked task waits for the resource. In addition to the restrictions imposed by the execution contexts, the methods of mutual exclusion are optimized for some instances, some to improve performance, others seeking determinism.

The next sections introduce the principal mutual exclusion primitives available in the Linux kernel, presenting the motivation for their usages and the behavior change in the PREEMPT_RT kernel. It is worth mentioning that this section does not aim to present an in-depth explanation of all synchronization methods but to introduce the terminology and some aspects that are later covered in the thesis.

#### 2.2.3.1 Spinlock

In a section protected by a spinlock, only one task is allowed access to a specific critical region. The behavior of a spinlock depends on whether the kernel is configured for single-core or multicore systems. In either case, the preemption is disabled before

attempting to acquire a lock and is enabled after the release of the lock. In the single-core case, this action is already enough to protect the critical section.

In the multicore case, when a task tries to acquire a spinlock that is not held by any other task, the lock is acquired. Otherwise, the task needs to busy-wait for the resource to be released by a task running on another CPU. Although busy-waiting consumes CPU time in vain, it avoids a more complex control, involving changing the task from *ready* to *sleeping*, calling the scheduler routines, causing context switch to another thread, and so on. Thus, the busy-waiting kernel spinlock is beneficial in the case of small critical sections. An important detail is that, before attempting to acquire a spinlock, the current task disables the preemption of the processor, enabling it again only after releasing the lock.

The `spin_lock()` and `spin_unlock()` are the main functions for acquiring and releasing a spinlock. The application programming interface (API) of the spinlocks also implements versions that disable interrupts. Such functions are necessary to prevent deadlocks. For instance: if a thread acquires a spinlock, then an interrupt arrives and tries to acquire the same spinlock, a deadlock will happen because the thread will be waiting for the return of the interrupt handler, never releasing the spinlock. The spinlock is used mainly in parts of the kernel where a task can not sleep, such as interrupt handlers and non-preemptive sections. In the kernel with PREEMPT_RT, spinlocks are mostly converted to real-time (RT) mutexes. The reason for this change is described in Section 2.2.3.7.

#### 2.2.3.2 Read-write spinlocks

In some cases, critical sections are accessed multiple times for data reads, but fewer times for an update. To improve the throughput, exclusive access to these data is required only when writing the data, while allowing concurrent accesses to read the data. In this case, there is contention only when a task waits to write, or tasks wait for a data being written.To acquire the read-write lock for reading one uses functions `read_lock()` and `read_unlock()` For writing it uses functions `write_lock()` and `write_unlock()`.

The vanilla kernel uses spinlocks to protect the write access. Thus the read-write spinlocks disable preemption. The read-write spinlocks also have versions that disable interrupts and softirqs. It is not possible to upgrade the `read_lock()` to a `write_lock()`, as this causes a deadlock.

An important detail is that the readers always take precedence over the writers. While there is a reader in the critical section, the writer can not run. Since readers can get the lock concurrently, even if a writer is waiting for the lock, new readers may acquire the lock and thus postpone indefinitely the acquiring of the lock by the writer.

In the kernel with PREEMPT_RT, control access to critical sections is made with the RT mutex, avoiding disabling the IRQs and preemption.

#### 2.2.3.3 Semaphores

Unlike spinlocks, semaphores do not use busy-waiting. When a task tries to acquire an unavailable semaphore, the task is placed on a waiting list, its status changes to sleeping, and the task leaves the processor. When the semaphore becomes available, tasks in the queue are awakened accordingly with the lock acquired, continuing its execution. As the kernel has some restrictions on where a piece of code can sleep, semaphores can not be used in the context of interrupts.

Semaphores accept various tasks in its critical section. A counter, created at the initialization time, controls the access to a critical section. To implement mutual exclusion using a semaphore, one should initialize the semaphore with a counter equals to one. Two basic functions can be used to acquire a semaphore: `down()` and `down_interruptible()`. The difference between the two modes is the way that the task is put to sleep: state interruptible or uninterruptible.

If a thread in the interruptible state receives a signal, it awakens immediately and the signal delivered to the task. On the other hand, a task in state uninterruptible is not waked up, thus delivering the signal is delayed until the task is awake and acquires the semaphore. Of these two, it is more common to use the so-called `down_interruptible()`. Function `up()` releases the semaphore.

When compared with spinlocks, semaphores have an advantage: semaphores do not disable preemption throughout the critical section. However, semaphores cause higher overhead because they put the task to sleep and wake it up after some time. In cases of small critical sections, this overhead can be higher than the critical section itself, so it is advised only for large critical sections.

Another side effect is that by making the task to sleep, a high-priority task can suffer unlimited priority inversion.

#### 2.2.3.4 Read-write semaphores

Semaphores also have a read-write version. Read-write semaphores do not have counters. The rule is the same as read-write spinlocks: a writer requires mutual exclusion, but several concurrent readers are possible. The precedence of the readers over the writers is the same as with the read-write spinlocks. Hence, writers can be blocked indefinitely.

The function to acquire the semaphore for reading is `down_read()`. For writing it is used the function `down_write()`. With read-write semaphores it is possible to downgrade the state `writer` to the state `reader`. This is done with the function `downgrade_write()`.

In the kernel with PREEMPT RT, control access to critical sections is made with the RT mutex, and the read-write logic is disabled: all read-write spinlocks end up

converted to regular RT mutex.

#### 2.2.3.5 Mutex

The mutex option was implemented as a simple mutual exclusion to put tasks on contention to sleep, mainly to replace semaphores initialized with a count of one. Despite having a behavior similar to a semaphore with a count of one, the mutex has a simpler interface, better performance, and more use restrictions, which facilitates system debugging (LOVE, 2010).

To acquire a mutex, it is used the function `mutex_lock()`. If the mutex is not available, the task is put to sleep. To release a mutex, the function used is `mutex_unlock()`. In comparison with spinlocks, mutexes have the same benefits and problems of counting semaphores initialized to one.

#### 2.2.3.6 RT mutex

The RT mutexes extend the semantics of mutexes with the priority inheritance protocol. In an RT mutex, when a low-priority task holds an RT mutex, and this RT mutex is blocking a task of higher priority, the low-priority task inherits the higher-priority task priority. If the task that inherited the priority blocks on another RT mutex, this propagates the priority to another task until the task that holds the RT mutex releases the mutex that blocked the highest-priority task. This approach helps to reduce the blocking time of high-priority tasks, avoiding unbounded priority inversion.

#### 2.2.3.7 Spinlocks and RT mutex in the PREEMPT RT

In the PREEMPT RT, spinlocks and mutexes are converted to RT mutexes. Spinlocks are converted to RT spinlocks, using the RT mutex to implement mutual exclusion. This is possible because in the PREEMPT_RT many sections of the kernel, which were initially in interrupt context, were converted to threads running in the address space of the kernel, so the spinlocks used in these sections can be converted to RT mutex. In parts of the kernel that it can not sleep even with the PREEMPT_RT, the original spinlocks are used, with the prefix `raw_`, for example, `raw_spin_lock()`.

A major benefit of transforming spinlocks in RT mutexes comes from the fact that the RT mutexes do not disable preemption, reducing the latency. In fact, the use of RT mutexes instead of spinlocks and the execution of device interrupt handlers and softirqs in the context of threads are the two major causes for the decrease of latency in PREEMPT_RT, when compared to the vanilla kernel.

Figure 3 – `ftrace` output.

```
sh-2038  [002] d... 16230.043339: ttwu_do_wakeup <-try_to_wake_up
sh-2038  [002] d... 16230.043339: check_preempt_curr <-ttwu_do_wakeup
sh-2038  [002] d... 16230.043340: resched_curr <-check_preempt_curr
sh-2038  [002] d... 16230.043343: sched_wakeup: comm=cat pid=2040 prio=120 target_cpu=003
```

### 2.2.4 Linux tracing

Linux has an advanced set of tracing methods, which are mainly applied in the runtime analysis of kernel latencies and performance issues[3]. The most popular tracing methods are the `function tracer` that enables the trace of kernel functions (ROSTEDT, Steven, 2010), and the `tracepoint` that enables the tracing of hundreds of events in the system, like the *wakeup* of a new thread or the occurrence of an interrupt. But there are many other methods, like `kprobes` that enable the creation of `dynamic tracepoints` in arbitrary places in the kernel code, and composed arrangements like using the `function tracer` and `tracepoints` to examine the code path from the time a task is woken up to when it is scheduled.

An essential characteristic of the Linux tracing feature is its efficiency. Nowadays, almost all Linux based operating systems (OSes) have these tracing methods enabled and ready to be used in production kernels. Indeed, these methods have nearly zero overhead when disabled, thanks to the extensive usage of runtime code modification techniques that allow for greater efficiency than using conditional jumps when tracing is disabled. For instance, when the `function tracer` is disabled, a `no-operation` assembly instruction is placed right at the beginning of all traceable functions. When the `function tracer` is enabled, the `no-operation` instruction is overwritten with an instruction that calls a function that will *trace* the execution, for instance by appending information into an in-kernel trace buffer. This is done at runtime, without any need for a reboot. A `tracepoint` works similarly, but using a jump label (CORBET, Jonathan, 2010). The mentioned tracing methods are implemented in such a way that it is possible to specify how an event will be handled dynamically, at runtime. For example, when enabling a `tracepoint`, the function responsible for handling the event is specified through a proper in-kernel API.

Currently, there are two main interfaces by which these features can be accessed from user-space: `perf` and `ftrace`. Both tools can hook to the trace methods, processing the events in many different ways. The most common action is to record the occurrence of events into a trace-buffer for post-processing or human interpretation of the events. Figure 3 shows the output of the `ftrace` tracing functions and `tracepoints`. The recording of events is optimized by the usage of *per-CPU* lock-less trace buffers. Furthermore, it is possible to take actions based on events. For example, it is possible

[3] See *Linux Tracing Technologies*: https://www.kernel.org/doc/html/latest/trace/index.html.

Table 1 – Mapping between mechanisms of the Linux kernel and abstractions of the response-time analysis.

| Mechanism | Abstraction |
|---|---|
| NMI Handler | Task |
| IRQ Handler | Task |
| Thread | Task |
| IRQ Control | Release Jitter |
| Preemption Control | Release Jitter |
| Spinlock | Blocking |
| RW Spinlocks | Blocking |
| Semaphores | Blocking |
| RW Semaphores | Blocking |
| Mutex | Blocking |
| RT Mutex | Blocking |

to record a *stacktrace*.

These tracing methods can also be leveraged for other purposes. Similarly to `perf` and `ftrace`, other tools can also hook a function to a tracing method, non-necessarily, to provide a trace of the system execution to the user-space. For example, the Live Patching feature of Linux uses the `function tracer` to hook and deviate the execution of a problematic function to a revised version of the function that fixes a problem (POIMBOEUF, 2014).

### 2.2.5 Characterization of real-time Linux tasks timeline

In (OLIVEIRA; OLIVEIRA, 2016), the authors described the timeline of the tasks in the PREEMPT_RT kernel, using the abstractions of the response-time analysis. The description of the timeline was aided by empirical information obtained by tracing the kernel with a custom-developed tool, based on `ftrace`. The next sections present a summary of this research that serves at the same time background and related work. The limitations and the lessons learned are presented at the end of the section, motivating for the development presented in this thesis.

#### 2.2.5.1 Kernel mechanisms and the response time analysis

This Section presents the mapping of the Linux kernel synchronization mechanisms to the abstractions used in the response time analysis. This mapping is described in Table 1. The three execution contexts of Linux, i.e., NMI, IRQs, and threads, were mapped into the *task* abstraction. The preemption and interrupt control, that can delay the start of the execution of the *task* context, were mapped into the release *jitter abstaction*. Finally, all the mutual exclusion methods were mapped into the *blocking* abstraction.

Table 2 – Linux events used in the parallel with the response-time analysis.

| | |
|---|---|
| NMI Handlers entry | Start of NMI handler execution |
| NMI Handlers exit | Finish of NMI handler execution |
| IRQ Handlers entry | Start of IRQ handler execution |
| IRQ Handlers exit | Finish of IRQ handler execution |
| IRQ Enable event | Disable IRQ handler execution |
| IRQ Disable event | Disable IRQ handler execution |
| Sched Wakeup | Wakeup of a thread |
| Sched Wakeup New | Wakeup of a new thread |
| Schedule function entry | Start of the schedule function |
| Schedule function exit | Finish of the schedule function |
| Preempt disable | Disables the preemption of the current thread |
| Preempt enable | Enable the preemption of the current thread |
| Context switch | Context switch of threads |
| Lock Acquire | The task wants to acquire the lock |
| Lock Acquired | The task acquired the lock |
| Lock Contended | The task contended waiting the lock |
| Lock Release | The task released the lock |

*Scheduling overhead*

An important concept in RTOS design is the overhead imposed by the scheduler. The scheduling overhead is associated with selecting the next task to be scheduled and the context switching overhead. A Limitation of the classic response-time analysis model is the absence of scheduling overhead that resembles Linux behavior. Moreover, many theoretical studies, often the scheduling overhead, are considered negligible or considered part of the task's execution time of the task (SOUTO et al., 2015) (BLOCK; ANDERSON, 2006). However, many empirical studies consider the measurements of scheduling overhead. These overheads are measured primarily to determine an upper bound, or to compare different implementations of schedulers (KENNA et al., 2011) (BASTONI; BRANDENBURG; ANDERSON, 2010). To demonstrate the impact of the scheduler overhead in the execution of threads, the work presented in (OLIVEIRA; OLIVEIRA, 2016) also considers the `__schedule()` execution in the timeline description.

#### 2.2.5.2 Trace-timeflow: empircal observation of the system

The `ftrace` tool (ROSTEDT, Steven, 2010) was adapted to trace the occurrence of interesting events in Linux. The adapted trace tool was called `trace timeflow`. The `trace timeflow` traces the events listed on Table 2. Sich events served to observe the execution of the three different kinds of tasks and the actions that could impact their timing behavior. For instance, the IRQ Handlers entry and exit report the *starting time* and the *finishing time* of a job of a given IRQ.

By tracing the execution of threads and interrupt handlers using the `trace timeflow`, it was possible to characterize the execution of these tasks using the real-time scheduling theory abstractions. An example of the `trace timeflow` output is shown in

Figure 4. In this example, the thread `pi`, with process identification number (PID) 838, was running in the CPU 0, when it caused a *read* system call in Line 5 and 6. The task then called the function `_raw_spin_lock()` to acquire a spinlock in Line 8. Then the preemption is disabled, and the lock `mr_lock` is acquired without contention, returning to the thread execution in 989 nanoseconds (Lines from 9 to 12). The thread then calls the function `_raw_spin_unlock()`, releases the lock, and enables the preemption, in Lines from 13 to 16. Finally, the thread finishes its execution returning from the system call.

Figure 4 – Trace `timeflow` output example.

```
1  # tracer: timeflow
2  #
3  #  TASK-PID  PRIO  CPU      TIME         DURATION           FUNCTION CALLS
4  #   |    |    |     |        |  |          |   |             |   |   |   |
5     pi-838  [  9]  [00]  83.910612437  ---------->
6     pi-838  [  9]  [00]  83.910612437                 sys_read() {
7     pi-838  [  9]  [00]  83.910613717                   run_higher in: c = 1 inc = 11287
8     pi-838  [  9]  [00]  83.910613955                   _raw_spin_lock() {
9     pi-838  [  9]  [00]  83.910614175                     sched_preempt_disable: at _raw_spin_lock
10    pi-838  [  9]  [00]  83.910614368                     lock_acquire: ffffffffa02d6578 &mr_lock
11    pi-838  [  9]  [00]  83.910614750                     lock_acquired: ffffffffa02d6578 &mr_lock
12    pi-838  [  9]  [00]  83.910615092  0.989 us         } _raw_spin_lock ()
13    pi-838  [  9]  [00]  83.910616591                   _raw_spin_unlock() {
14    pi-838  [  9]  [00]  83.910616786                     lock_release: ffffffffa02d6578 &mr_lock
15    pi-838  [  9]  [00]  83.910617153                     sched_preempt_enable: at _raw_spin_unlock
16    pi-838  [  9]  [00]  83.910617315  0.571 us         } _raw_spin_unlock ()
17    pi-838  [  9]  [00]  83.910617648                   run_higher out: c = 1 inc = 11288
18    pi-838  [  9]  [00]  83.910618197  5.609 us       } sys_read ()
19    pi-838  [  9]  [00]  83.910618197  <----------
```

#### 2.2.5.3 Characterization of interrupt handlers timeline

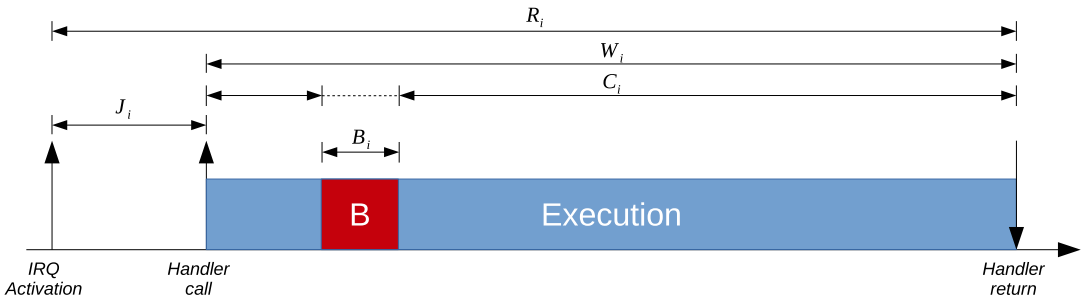
Due to the different restrictions imposed on maskable and non-maskable interrupts, it was necessary to characterize the interrupts for these two different modes.

A non-maskable interrupt can be enabled at any time, and therefore must obey a set of very strict rules. For example, a non-maskable interrupt handler can not use mutual exclusion mechanisms, except when it is used only in this context, for synchronization with other NMIs running on another CPU. The code of NMI handlers can not be reentrant, i.e., a second NMI will not be handled during the execution of an NMI.

From these restrictions and the trace of interrupts, it is possible to characterize the execution of NMIs as in Figure 5, where the major quantities of interest, as introduced in Section 2.1 are visually highlighted.

For NMIs, the response time $R_i$ is given by the delay between the IRQ activation and the return of the NMI handler. The release jitter $J_i$ will occur if the system is already handling a non-maskable interrupt. In this case, it is safe to assume the worst case: that the second NMI was activated right after the first NMI was activated.

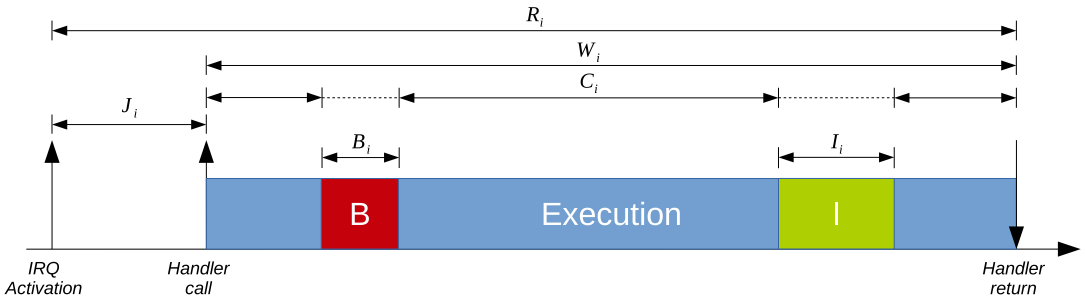
Figure 5 – Non-maskable interruption timeline.




The busy window $W_i$ is defined as the time that the NMI held the CPU during its execution, being determined by the time interval between the call and the return of the IRQ handler. The blocking represented by variable $B_i$ must be implemented as busy waiting, which should occur only for synchronization between non-maskable interrupts in different processors. Finally, the runtime $C_i$ is determined by the busy window, discounting the time that the NMI may have been blocked by another NMI.

The characterization of the maskable interrupt handlers is shown in Figure 6.

Figure 6 – Maskable interruption timeline.



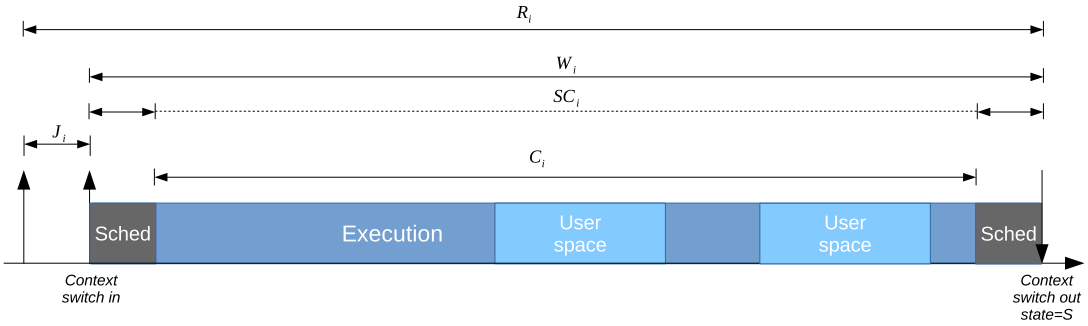

For maskable interrupts, the response time $R_i$ is determined by the time interval between the activation and the return of the interrupt handler. The release jitter $J_i$ can happen if the system has interrupts disabled, either by an action of the operating system or by the action of the processor itself, e.g., if it is already handling an interrupt. In this case, it is safe to assume that in the worst-case activation took place immediately after the disabling of interrupts.

Unlike non-maskable interrupts, maskable interrupts can suffer interference $I_i$, caused by a non-maskable interrupt. The busy window $W_i$ is defined as the time that the interruption held the CPU during its execution, being determined by the time interval from start to finish of the interrupt handler. Blocking $B_i$ is always implemented as busy waiting. Lastly, the runtime $C_i$ is determined by the busy window discounting blocking and interference from other interrupt handlers.

#### 2.2.5.4 Characterization of the threads timeline

The characterization of real-time threads is more complex than that of the interrupt handlers. Therefore, it was made in parts. The first part considers activation without blocking and interference. The second one identifies the different forms of blocking and interference, showing how they can affect the timing behavior of real-time threads. Figure 7 describes the execution of a real-time thread without interference or blocking.

Figure 7 – Real-time thread.




For threads, the response time $R_i$ is the time between the thread activation by the event `sched_wakeup`, and the context switch when the thread leaves the processor, suspending its execution in state `S`. The busy window $W_i$ is the time interval between the first context switch after the activation of the task and the context switch in which the task leaves the processor to sleep, finishing its execution. The release jitter $J_i$ can be associated with two reasons: preemption or interrupts being disabled by process of lower priority, and by a scheduler execution that removes the current task. Both must happen at the processor on which the task was activated.

After a task starts its execution, the scheduling routine that had suspended the task runs until it returns to the application code. The scheduling overhead is associated with the variable $SC_i$, comprising the exit-scheduling overhead, i.e., the time between the calling of function `schedule()` and the context switch, and the entry-scheduling overhead, that is the time between the context switch and the return of function `schedule()`.

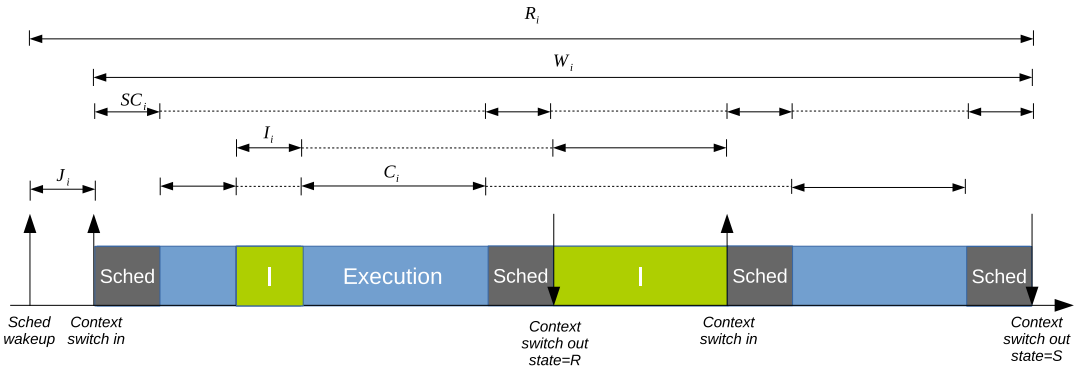
Finally, the computation time $C_i$ is the time that the thread has executed its own code, which can be either in user space or kernel space, excluding scheduling overhead, blocking and interference.

Regarding the interference, $I_i$, Figure 8 describes the two forms of interference that a task can suffer: from an interrupt handler and from another thread.

Since the interrupt handlers are activated by the hardware, they do not need to be scheduled. The interference of an interrupt handler is given by the busy window $W_i$ of the interrupt handler.

Differently from the interference of interrupt handlers, the interference caused by threads adds scheduling overhead to the currently running task. This overhead

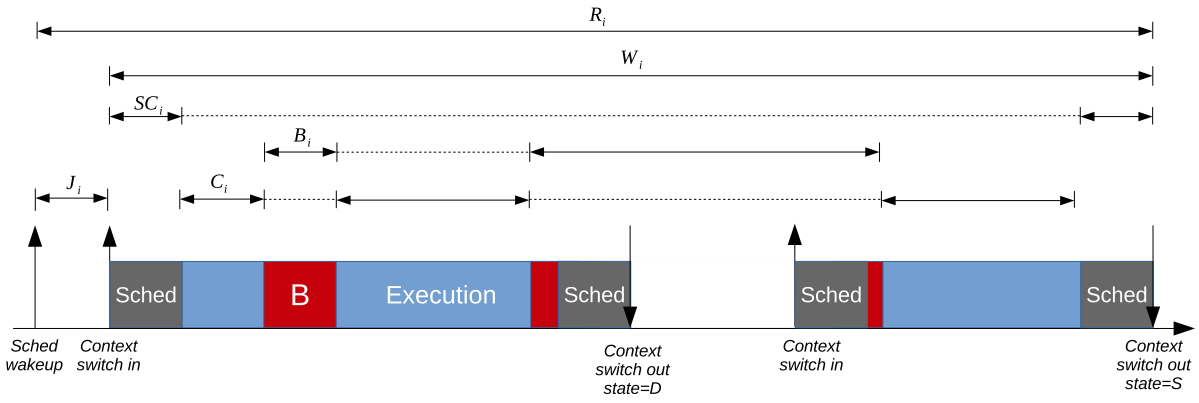
Figure 8 – Forms of thread interference.




increases the scheduling overhead of the task itself. The interference of a high-priority thread is given by the time interval between the context switch that removes the current thread from the processor, and the context switch that gives back the processor to the thread. It is possible to identify whether a thread is suffering interference by the state that it is leaving the processor. When a real-time thread leaves the processor in `R` state, it is suffering interference.

Regarding locks, one thread can experience two forms of blocking: implemented as busy waiting, or implemented by suspending the execution of the thread. Figure 9 demonstrates both cases.

Figure 9 – Forms of thread blocking.




The first example of $B_i$ (left) is a busy-waiting lock, where the task keeps running on its context until another thread releases the lock. In the trace, it is possible to identify this blocking by the tracepoint `lock_contended`. After acquiring access to the critical section, tracepoint `lock_acquired` is shown. Thus, the blocking time is given by the time interval between these two tracepoints.

The second example is (right) the type of blocking that suspends the execution of the thread until it acquires the critical section. In this case, as the scheduling overhead happens due to the mutual-exclusion mechanism, it is considered that this time is part

of the blocking time of the task, and the measurement is made in a manner analogous to the mechanisms that do not suspend execution: the time interval between tracepoints `lock_contended` and `lock_acquired`.

#### 2.2.5.5 Final remarks

Although the work presented in (OLIVEIRA; OLIVEIRA, 2016) successfully describes the execution of tasks on Linux using terms from real-time scheduling theory, it can be improved to cover all the possible states better a task can have in a single execution. For example, it shows the kinds of the *blocking* a task may suffer, but it lacks the description of how many times a task can block, or that, a task can block during the scheduling actions or the possibility of nested locking. The interpretation can also be improved by adding a new set of variables that can better describe the particular behavior of Linux. To improve the state-of-art, we believe that a mathematical model is required. The model should be based on well-defined criteria that describe, in a deterministic way, enabling the analysis of all possible states a task can have on Linux.

Toward the formalization of Linux execution tasks, the next section introduces concepts from formal methods theory, the modeling of DES, followed by the modeling approaches applied in the modeling of complex systems.

## 2.3 FORMAL METHODS

Formal methods are fundamental in the development of cyber-physical systems. Mainly to systems that can cause major damage, including loss of life, which are the case of safety-critical hard real-time systems. Formal methods consist of a collection of mathematical methods to rigorously state the specification of a system. The specifications of a system can then be used for multiple purposes, including the formal verification of the correctness of a program. The advantage of the usage of mathematical notation is that it removes the ambiguous nature of natural language while enabling automatic verification of the system.

Spivey (SPIVEY, 1989) defines formal specification as:

- *"Formal specification is the use of mathematical notation to describe in a precise way the properties that an information system must have, without unduly constraining the way in which these properties are achieved."*

While O'Regan (O'REGAN, 2017) defines the properties of a system as follows:

- *"The properties are a logical consequence of the mathematical definition, and the requirements may be amended where appropriate."*

Several different specification formalisms can be used in the specifications of computing systems. These formalism can be classified into two different approaches: the *axiomatic or algebraic approach* (CENTER et al., 1976) and the *model-oriented approach* (CLARKE; EMERSON; SIFAKIS, 2009).

The *axiomatic or algebraic approach* dates from the mid-70s. This technic uses different mathematical structures from the modern algebra, such as groups, rings, fields, in the specification of data types, using then the equational logic as a specification mechanism (EHRIG et al., 1992). The calculus communicating systems (CCS) and the communicating sequential processes (CSP) are example of specification languages using the *axiomatic or algebraic approach*.

The *model-oriented approach* uses a mathematical model for the specification of a system. In this context, a model is a mathematical abstraction of a system that exists in the real world, containing only the necessary details to explain a given behavior of the system. Discrete mathematics elements, such as set theory, functions and relations are applied to the modeling of computer systems, both directly, or as part of more complex forms, such as finite-state machines and automata. Many sorts of formal verification methods can be then applied to formal models of systems, such as *model checking* and RV.

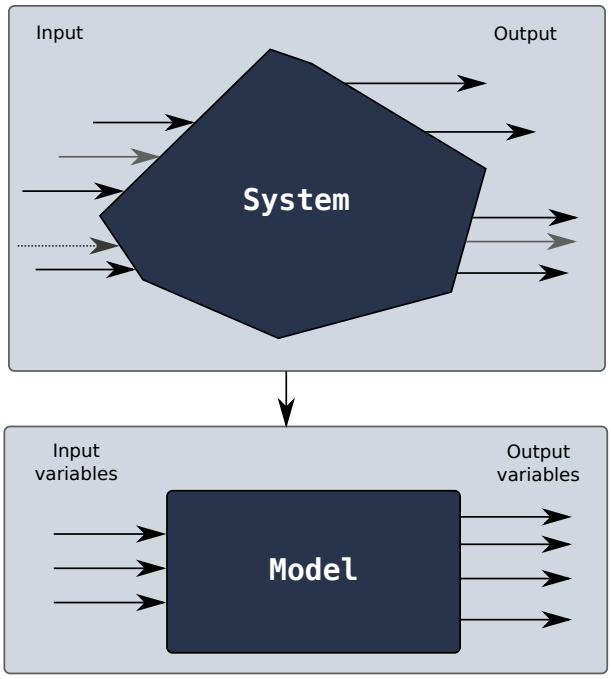
As presented in Chapter 3, the usage of formal models for Linux has gained attention in the last year, motivating many different theoretical and practical work. Motivated by this trend, along with other arguments explained in the next sections, this thesis follows the *model-oriented* approach.

### 2.3.1 Models

A model is an *abstraction* (a set of mathematical equations) of a system, whereas a system is *something real*, e.g., an amplifier, a car, a factory, a human body, and so on. The process of modeling a system involves the definition of a set of measurable variables associated with the given system. The subset of variables acting on the system from outside is considered input variables. The subset of these variables, which is possible to measure while varying the input directly, is defined as the set of output variables. The input variables can be thought as the *stimulus*, which produces a *response* on output variables, as in Figure 10. It is possible that some variables are not associated with either the input or the output, these are sometimes referred to as suppressed output variables. To complete a model, it is reasonable to postulate that there exists some mathematical relationship between the input and the output (CASSANDRAS; LAFORTUNE, 2010).

It is essential to emphasize flexibility in the modeling process since no unique way to select input and output variables is imposed. Thus, it is the task of the modeler to identify these variables, depending on a particular point of view or on the constraints

Figure 10 – System and Model.




imposed by a specific application. Generally, the model only approximates the complete behavior of the system, and it is a good practice to try to simplify the model, following the "Ockham's Razor" (law of parsimony) approach. The *adequacy* is a crucial aspect during the definition of the level of abstraction used in the model. The *adequacy* of a model determines how effectively the model represents the underlying behavior of the system (O'REGAN, 2017). When an *adequate* model is found, the terms system and model can be interchangeably used. The following items are desirable characteristics of a software reliability model:

- good theoretical foundation;
- realistic assumptions;
- good empirical support;
- as simple as possible;
- trustworthy and accurate.

A fundamental step in the development of a model is the clear definition of its purpose. As described in Section 2.1.1, in the real-time scheduling theory, a system is modeled as a set of tasks. Based on this point of view, this work endeavors the creation of an explicit formal model for the tasks of Linux, including the variables that influence the timing behavior of the tasks.

Generally, the specifications of computing systems are created in an early stage of the process and are used as a reference for the development and testing of the

system. However, Linux already exists, and it is unrealistic to think about a complete redesign of the core features of the kernel. Thus, the purpose for the creation of an explicit model of the task of Linux are:

- to promote the unambiguous understanding of the system from its formal specifications: useful in the definition of the timing behavior of Linux;
- to enable the validation of the model against the real execution of the system: a fundamental step to strengthen the trustworthy and accuracy of the model;
- to verify violations of the specifications during the execution of the system: useful in the formal verification of Linux behavior.

Despite the arguments in favor of the usage of formal methods, its application is generally restricted to specific sectors. The most commonly mentioned reasons for that are the complexity of the mathematical notation used in the specifications, along with limitations of computational space and processing time required for the verification of a system using formal methods. Regarding the complexity, it is a challenge for this work to find a formal specification notation that, at the same time, can be easily interpreted by kernel developers, useful to demonstrate the timing behavior of tasks, and able to verify the Linux kernel behavior appropriately. Regarding performance, a common problem faced by practitioners in the usage of formal methods is the *state explosion* problem. As exemplified in (CLARKE; EMERSON; SIFAKIS, 2009), the composition of a model of a system with $n$ tasks, with each task with $m$ states, will result in a model with $m^n$ states. Moreover, it is also important to consider the level of expressiveness of the specification notation. Generally, a more concise notation is likely to be more efficient than feature-rich notation.

Hence, as important as defining the goals of a model, it is determining what are not the goals of a model, limiting its scope to avoid hitting the limitations of formal modeling. In this regard, a common motivation to create a model is to predict future behavior (EPSTEIN, 2008) In the case of the real-time system scheduling theory, the prediction of the future generally means the definition of the schedulability of a system, with many articles that explore this possibility (FERSMAN; MOKRUSHIN, et al., 2006; NORSTROM; WALL; WANG YI, 1999; AMNELL, T. et al., 2002; ABDEDDAÏM; MALER, 2001). However, it is essential to clarify that this is not the main goal of this project. The justification for such choice is avoiding well-known restrictions on the use of formal methods with complex software such as the Linux kernel.

From all available methods to formally describe a system, the methods used the DES demonstrated to be an appropriate choice for this research, mainly because:

1. real-time systems models are generally formalized using discrete math methods;

2. OS developers are familiar with state-machine like descriptions;
3. Linux possesses logical and event-based behavior;
4. Linux has a vast set of events that can be traced efficiently;
5. the previous attempt to describe Linux behavior was made using tracing;
6. a vast set of formal verification methods are available for DES, such as RV and model checking.

The next sections present the basics notions of DES and automata theory used in the development of the model.

### 2.3.2 Discrete event systems

A set of appropriate models for DES was developed to adequately describe the behavior of these systems and provide a framework for analytic techniques to meet design, control, and performance evaluation goals. This section presents the most basic format for the description of a DES: the *language* formalism.

The evolution of a DES can be thought of as a sequence of visited states, and the associated events causing transitions. The description of the evolution is then done as a sequence of events, for instance, $e_1, e_2, e_3, ..., e_n$, that describes the logical behavior of the system. All possible sequences of events define the language that describes the system. The issue of representing a language using appropriate modeling formalism is the key to do the analysis, control, and hence performance evaluation of a DES.

The starting point of a DES is the underlying finite event set $E$ associated with it. The set $E$ represents the "alphabet" used to form "strings" (or "words" or even "trace") of event sequences that compose a language. This framework can be used either to define the language to be performed by a new system or to identify the language spoken by an existing system formally.

A string composed of no events is called the empty string, and it is denoted by $\varepsilon$. 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 $s$.

#### 2.3.2.1 Language definition

A language defined over an event set $E$ is a set of finite-length strings formed from events in $E$.

For example, let the set $E$ be composed of $\{a, b, c\}$. It is possible to define the language $L_1$ with the following four strings:

$$L_1 = \{\varepsilon, a, ab, acc\} \tag{4}$$

It is also possible to define the language $L_2$ with four strings as following:

$$L_2 = \{\textit{strings with two elements starting with } a\} = \{\varepsilon, aa, ab, ac\} \tag{5}$$

Moreover, it is also possible to have a set with infinite elements. For instance:

$$L_3 = \{\textit{all strings starting with } a\} \tag{6}$$

The concatenation is the base operation of the composition of a string, hence a language. Taking $L_1$ as example, *acc* can be seen as the concatenation of *a*, *c* and *c* again, or the concatenation of *a* and *cc*, or even the concatenation of *ac* and *c*. Regarding the empty string, since $a\varepsilon = \varepsilon a = a$ it is possible to classify $\varepsilon$ as the identity element of concatenation.

To be able to formally define concatenation, the Kleene-closure operation, denoted by $^*$, must be presented. The Kleene-closure of a set $E$ is the set $E^*$ composed of all finite strings of elements from $E$. Therefore, it is possible to state that $E$ is a subset of $E^*$. It is noteworthy that $E^*$ contains an infinity number of elements.

Formally, the Kleene-closure of a language $L$ is defined as: Let $L \subseteq E^*$, then

$$L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \cup ... \tag{7}$$

Therefore, concatenation is defined as follows:
Let $L_a, L_b \subseteq E^*$, then

$$L_aL_b := \{s \in E^* : (s = s_as_b) \textit{ and } (s_a \in L_a) \textit{ and } (s_b \in L_b)\} \tag{8}$$

Another important operation is the prefix-closure. A language $L$ is said to be prefix-closed, denoted by $\overline{L}$, if any prefix of any string in $L$ is also an element of $L$. Formally, Let $L \subseteq E^*$, then

$$\overline{L} := \{s \in E^* : (\exists t \in E^*)[st \in L]\} \tag{9}$$

Finally, the following three terms are often used on strings operations:
Let $s = abc$, then:

- $a$ is a prefix of $s$,

Figure 11 – Example of automaton.

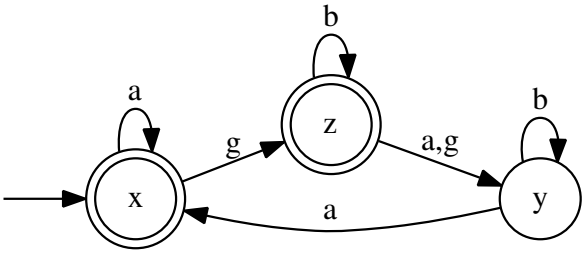
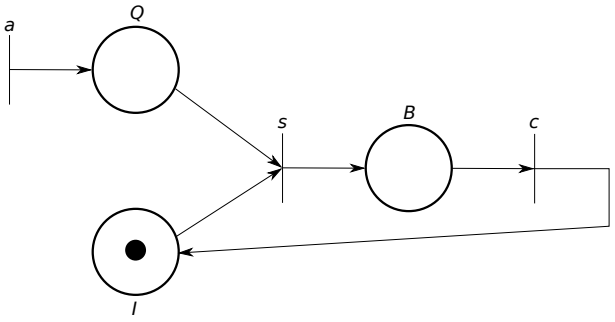



Figure 12 – Example of Petri net.




- *b* is a substring of *s*, and
- *c* is a suffix of *s*.

It is important to note that $\varepsilon$ and *s* are both prefix, suffix and a substring of *s*. Moreover, since languages are sets, usual set operations such as union, intersection, difference, and complement with respect to $E^*$ can be used.

#### 2.3.2.2 DES modeling formalism

Although languages enable the formal modeling of a DES by describing all possible sequences of events that a DES can produce or process, the absence of an additional level of structures to describe the logical behavior of a system makes its usage not always easy when modeling arbitrarily complex systems. However, it is possible to surpass this limitation with two formalism: *Petri nets* and *automata*. Both formalisms can represent a language using a graphical state transition format, and enable a modular modeling approach, turning the development more intuitive and flexible. However, both formalism have different concepts and representations, as shown Figures 11 and 12.

Both automata and Petri nets were tried with preliminary versions of the IRQ model (OLIVEIRA, D. B. de et al., 2017). In both cases, it was possible to create simple models using a modular approach[4] and conduce non-functional verification of the models.

However, the automata format showed to be a better option for this project. The automata format is simpler and more intuitive for OS developers, facilitating the

[4] See Section 2.3.3.2.

goal of promoting the unambiguous understanding of Linux, which was confirmed with developers during the development of models. For example, preliminary versions of the models were presented to the real-time Linux kernel community at the Real-time Linux Summit 2018 (OLIVEIRA, 2018e) and the general community at the Linux Plumbers Conference (OLIVEIRA, 2018f,d).

Moreover, the more sophisticated format of Petri nets resulted in longer processing times for composition and non-functional verification of the models, even for models with less than one hundred states in its automata version. Foreseeing a large number of states for the desired thread model, confirmed by the final thread model in Chapter 4 with more than nine thousand states, the choice for the automata formalism was made.

The next section presents the basics of automata theory, including the modeling approach used later in the project.

### 2.3.3 Automata theory

By using automaton formalism, it is possible to define a language according to well-defined rules. Automata are intuitive and easy to use, they are 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, let the event set be $E = \{a, b, c\}$. Then, consider the state transition diagram in Figure 11, where nodes represent 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

$$G = \{X, E, f, x_0, X_m\} \tag{10}$$

where:

- $X$ is the set of states
- $E$ is the finite set of events
- $f : X \times E \rightarrow X$ is the transition function. It defines the state transition in the occurrence of a event from $E$ in the state $X$.
- $x_0$ is the initial state
- $X_m \subseteq X$ is the set of marked states

For instance, the automaton $G$ represented in Figure 11 can be defined as follows:

- $X = \{x, y, z\}$
- $E = \{a, b, g\}$
- $f =$
  - $f(x, a) = x$
  - $f(y, a) = x$
  - $f(z, b) = z$
  - $f(x, g) = z$
  - $f(y, b) = y$
  - $f(z, a) = f(z, g) = y$
- $x_0 = x$
- $X_m = \{x, z\}$

It is also possible to add to the automaton the active event function (or feasible event function) $\Gamma : X \implies 2^E$. $\Gamma(x)$ is the set of all events $e$ for which $f(x, e)$ is defined in the state $x$.

The automaton works as follows. It starts at the initial state $x_0$ and upon the occurrence of an event $e \subseteq E$ with $f(x_0, e)$ defined, the state transition from $x_0$ to $f(x_0, e)$ will take place. This process continues based on the transitions for which $f$ is defined. For the sake of convenience, $f$ is always extended from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

$$f(x, \varepsilon) = x \tag{11}$$

$$f(x, se) = f(f(x, s), e) \text{ for } s \in E^* \text{ and } e \in E \tag{12}$$

Informally, following the graph of Figure 11 it is possible to see that the occurrence of the event $a$, followed by the event $g$ and $a$ will lead from the initial state to state $y$. Using automaton formalism, the same can be formally specified as follows:

$$\begin{aligned} f(x, aga) = f(f(x, ag), a) = f(f(f(x, a), g), a) \\ = f(f(x, g)a) = f(z, a) = y \end{aligned} \tag{13}$$

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, formally:

$$\mathcal{L}(G) = \{s \in E^* | f(x_0, s) \text{ is defined}\} \tag{14}$$

Therefore, $s \in \mathcal{L}(G)$ if and only if it is an admissible path in the state transition diagram, here $\mathcal{L}(G)$ is prefix-closed by definition. Another language generated by an automaton 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. Formally:

$$\mathcal{L}_m(G) = \{s \in \mathcal{L}(G) | f(x_0, s) \in X_m\} \tag{15}$$

$\mathcal{L}_m(G)$ does not need to be prefix-closed. 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 state for a system, and $\mathcal{L}_m(G)$ is the set of possible final states for a system.

#### 2.3.3.1 Operations with automata

An important automaton operation is the *Parallel Composition*. The Parallel composition allows the merge of two or more automata models into one single model. The standard way of building a model of the entire system from models of individual systems components is by parallel composition (CASSANDRAS; LAFORTUNE, 2010).

Before describing the parallel composition, the operation *Accessible Part* needs to be presented. From the definition of $\mathcal{L}(G)$ and $\mathcal{L}_m(G)$, it is possible to see that all states from $G$ that are not accessible, or reachable, from $x_0$ (the initial state) can be deleted. When deleting a state, all the transitions attached to that event are also deleted. The operation that deletes states not accessible from $x_0$ from a given automaton $G$ is defined as $Ac(G)$, formally:

$$\begin{aligned} Ac(G) &:= (X_{ac}, E, f_{ac}, x_0, X_{ac,m}) \text{ where} \\ X_{ac} &= \{x \in X : (\exists s \in E^*)[f(x_0, s) = x]\} \\ X_{ac,m} &= X_m \cap X_{ac} \\ f_{ac} &= f|_{X_{ac} \times E \to X_{ac}} \end{aligned} \tag{16}$$

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 $G_1$ and $G_2$, their parallel composition is defined as:

$$G_1 \parallel G_1 := Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_{1\parallel 2}, (x_{01}, x_{02}), X_{m1} \times X_{m2}) \tag{17}$$

where:

$$f((x_1, x_2), e) := \begin{cases} (f_1(x_1, e), f_2(x_2, e)) & \text{if } e \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, e), x_2) & \text{if } e \in \Gamma_1(x_1) \backslash E_2 \\ (x_1, f_2(x_2, e)) & \text{if } e \in \Gamma_2(x_2) \backslash E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In words, in the parallel composition, a private event, that is an event present in only one automaton, can execute whenever possible. On the other hand, 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.

#### 2.3.3.2 Modeling approaches

There are two possible approaches to model a system, the *monolithic* and the *modular* approach (RAMADGE; WONHAM, 1987).

*The monolithic approach*

In the monolithic approach, firstly, an uncoordinated model of the system is built. This model should comprise all possible chains of events, including undesired sequences. This model is a named generator $G$. Then, based on a set of *specification*, a specification model $S$ is modeled. The specification $S$ controls the generator $G$ by disabling all undesired events in a given state. Finally, the specification $S$ and the generator $G$ are merged using the parallel composition $S \parallel G$. The composition $S \parallel G$ represents all the chain of events produced, or accepted, by the system. The resulting model is denoted by $S/G$, which is the specification $S$ controlling the generator $G$. The language $L(S/G)$ is the language generated by $S/G$.

For example, suppose that three mechanisms compose a washer and dryer machine. One mechanism controls the door: it opens and closes the door. The second mechanism is the washing machine, it accepts two commands: to start washing and to finish washing. Finally, the drying mechanism accepts two commands, to start drying and to stop drying.

The alphabet of the system is defined in Table 3. The initial state of the system is neither washing nor drying and with the door closed. The initial state is also a safe state. The generator $G$ of the washer and drying machine, including all possible combinations of events, is presented in Figure 13.

The specification model should be composed of the following specifications:

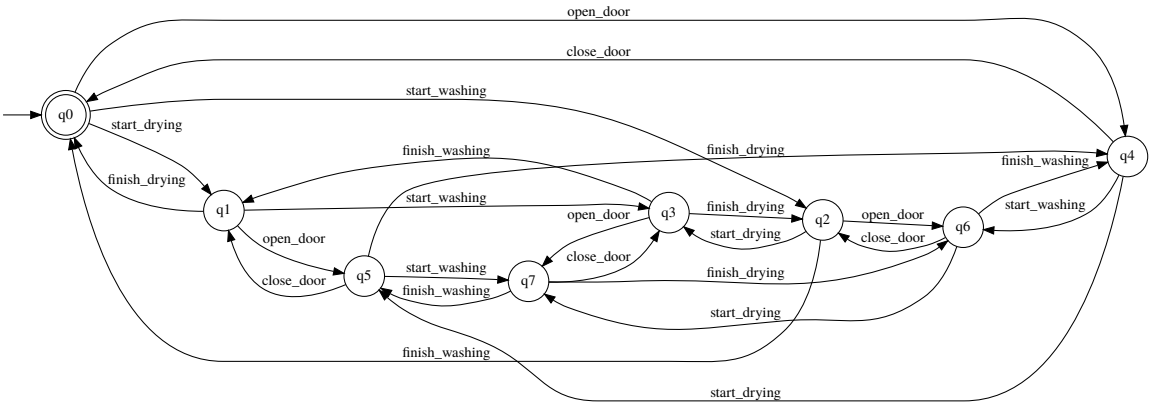
$S_1$: both washer and dryer should not start working with the door open;

$S_2$: the door should not open while either washing or drying;

Table 3 – Events of the washer and dryer machine.

| **Event** | **Sub-system** | **Description** |
|---|---|---|
| close_door | door | Closes the door |
| open_door | door | Opens the door |
| start_washing | washing | Starts to wash |
| stop_washing | washing | Stops to wash |
| start_drying | drying | Starts to dry |
| stop_drying | drying | Stops to dry |

Figure 13 – Monolithic generator $G$ of the washer and dryer machine.




$S_3$: once start washing, the machine should finish washing before staring drying; and

$S_4$: once start drying, the machine should finish drying before starting washing again.

By disabling undesired transitions, it is possible to find the specification model $S$ presented in Figure 14.

Then, given the generator $G$ and the specification $S$, the model $S/G$ is found by the synchronous product of $S \parallel G$, as in Figure 15.

*The modular approach*

Although the monolithic approach is good for simple systems, it is not efficient in the modeling of complex systems, as the number of states increases exponentially. Moreover, it is not easy to understand which specification disabled a specific event. For systems composed of many independent sub-systems, with several specifications, the modular approach becomes more efficient.

In the modular approach, rather than specifying a single generator $G$, the generator is modeled as a set of independent sub-systems, where each sub-system has its own alphabet. The generator $G$ is then composed by the parallel composition of

Figure 14 – Monolithic specification model $S$ of the washer and dryer machine.

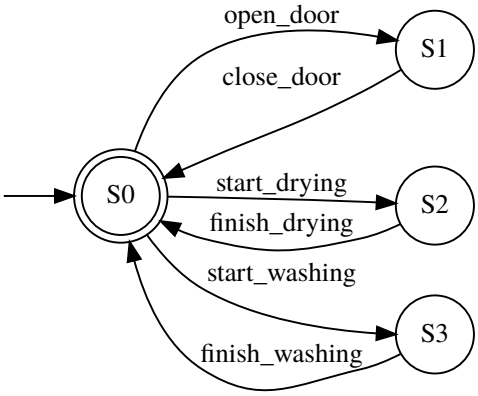



Figure 15 – $S/G$ of the washer and dryer machine.

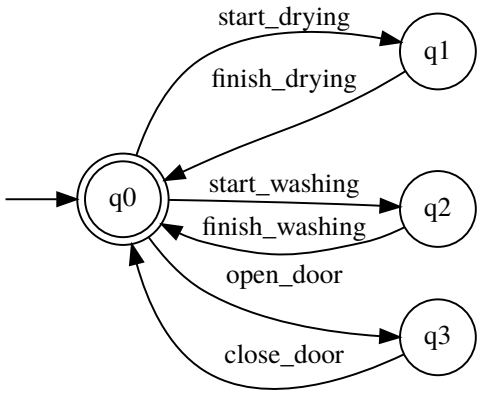



Figure 16 – Generator: $G_{door}$.

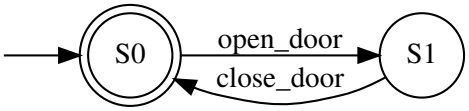



Figure 17 – Generator: $G_{wash}$.

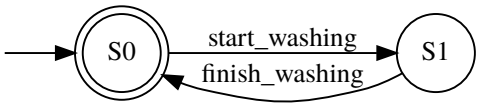
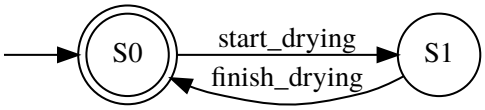



Figure 18 – Generator: $G_{dry}$.




all sub-systems. Similarly, rather than modeling a single specification $S$ to satisfy all specifications, each specification is modeled independently, using the alphabet of the sub-systems of the generator $G$ it aims to control. The parallel composition of all sub-systems then composes the generator $G$. This approach allows the module modeling of complex systems in a simpler way.

Using the same example, the generator can be naturally divided into three sub-systems: 1) door control; 2) washing; and 3) drying. These three generators are presented in Figure 16, 17 and 18, respectively.

In the same way, each of the specifications can be modeled separately. In Figure 19, the specification $S_1$ is modeled by blocking washing and drying start while the door is open. In Figure 20, the specification $S_2$ is modeled by blocking the door opening while the machine is either washing or drying. In Figure 21, the starting of the dryer is blocked while washing, and in Figure 22, the starting of the washer is blocked while drying. These two later models are the modes for the specification $S_3$ and $S_4$, respectively.

The generator $G$ is then defined as $\{G_{door} \parallel G_{wash} \parallel G_{dry}\}$, resulting in the

Figure 19 – Specification: $S_1$.

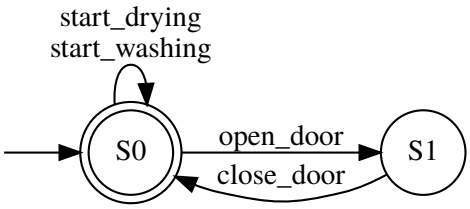



Figure 20 – Specification: $S_2$.

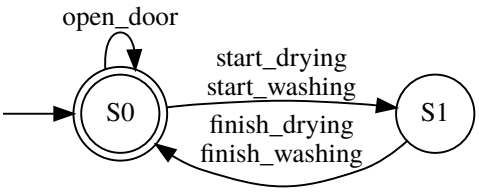



Figure 21 – Specification: $S_3$.

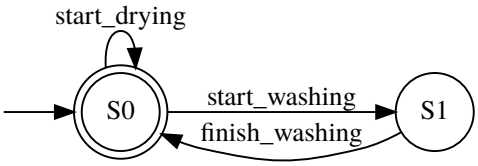



Figure 22 – Specification: $S_4$.

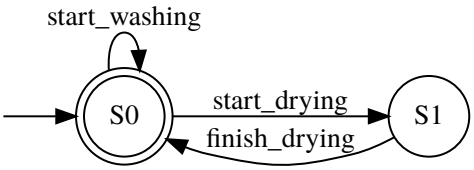
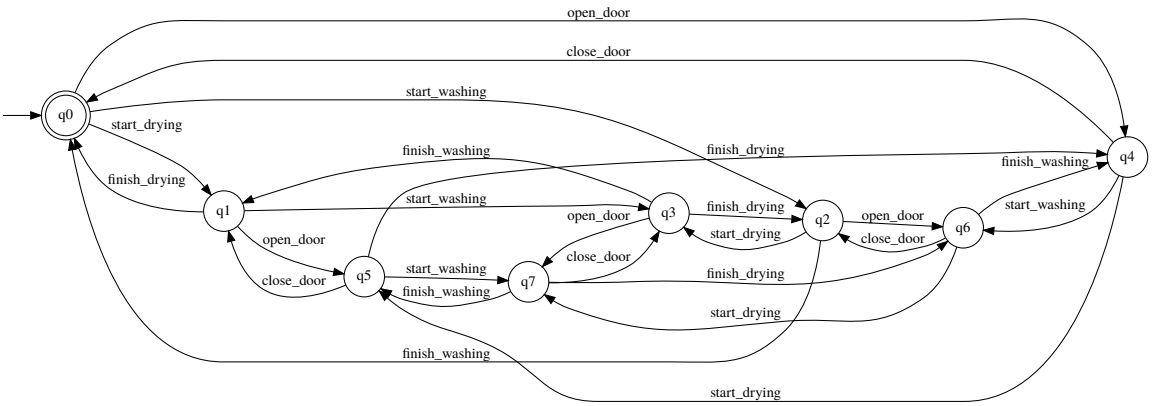



Figure 23 – Modular generator $G$ of the washer and dryer machine.




automaton in Figure 23. Note that the Modular $G$ and Monolithic $G$ (presented in Figure 13) are equivalent models, but developed using different approaches.

The modular specification $S$ is then defined as $\{S_1 \parallel S_2 \parallel S_3 \parallel S_4\}$, resulting in the automaton in Figure 24. Which is synchronized with $G$ to compose the controlled system in Figure 25.

As it is possible to see in Figure 25, the results of both monolithic and modular approaches are the same. However, the modular approach is more intuitive, helping in the understanding of each part of the system, and the specifications as well.

The automaton modeling can be done using specialized software, like Supremica. Supremica is an integrated development environment used in the modeling of large scale models. Supremica implements monolithic and modular verification and synthesis algorithms for solving non-blocking, controllability, and combined non-blocking and

Figure 24 – Modular specification $S$ of the washer and dryer machine.

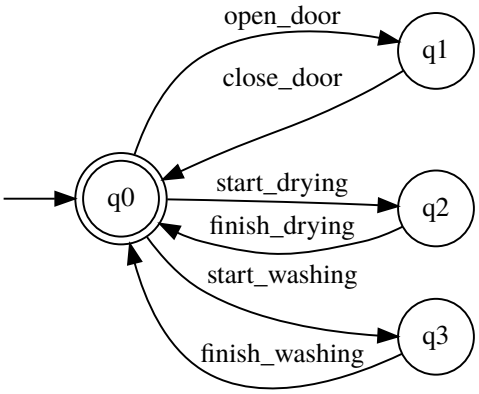
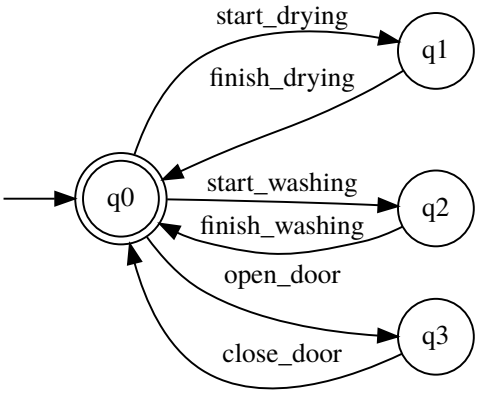



Figure 25 – Modular model $S/G$ of the washer and dryer machine.




controllability problems (AKESSON et al., 2006).

### 2.3.4 Runtime verification

Among the techniques available to the application of formal methods, Runtime Verification (RV) was chosen because of the runtime nature of the proposed model. RV presents a lightweight, yet rigorous, method that complements classical exhaustive verification techniques (such as model checking and theorem proving) with a more practical approach. At the price of a limited execution coverage, that analyses a single execution *trace* of a system, RV can give precise information on the runtime behavior of the monitored system (FALCONE et al., 2018).

One of the crucial points that motivate the usage of RV is the usage of the *events* and *trace* the abstractions to analyze the runtime behavior of the system. Indeed, real-time Linux developers use the *trace* features of Linux in the analysis of the runtime behavior of the system, and the work presented in Section 2.2.5 also bases in the *trace* execution, which are the same abstractions used in previously described DES. In RV, a property can be abstractly described as a set of traces, and a specification is a concrete (textual) object describing property, and therefore it denotes a set of traces.

When considering how to use RV to check if the runtime behavior of a system conforms to some specification, there are three necessary steps to be taken:

1. the specification of the desired, or undesired, behavior of the system;
2. the generation of a monitor that interprets the specification;
3. the connection between the monitor and the system to be observed.

The monitor of the system is the tool that connects the specification written in a mathematical format and the events generated by the system. The connection

between the system and the monitor is made through the instrumentation of the system. The instrumentation exhibits the behavior of the system by generating a trace of its execution.

Monitors can be classified as *offline* and *online* monitors. *Offline* monitors process the traces generated by a system after the occurrence of the events, generally by reading the trace execution from a permanent storage system. *Online* monitors process the trace during the execution of the system. *Online* monitors are said to be *synchronous* if the processing of an event is attached to the system execution, blocking the system during the event monitoring. On the other hand, an *asynchronous* monitor has its execution detached from the system. Each type of monitor has a set of advantages. For example, *offline* monitors can be executed on different machines but require operations to save the log to a file. *Asynchronous online* monitors generally result in lower overhead by avoiding both blocking the execution of the system and the manipulation of files (CASSAR; FRANCALANZA, 2015). However, only the *synchronous online* method can take actions at the exact moment a violation occurs, enabling additional data collection in the precise state in which the system shows a defect.

In the context of this work, the runtime verification will have a dual goal: initially, to verify the correctness of the model against the real execution of the system. Once a satisfactory model is found, the model will be used to verify the runtime behavior of the Linux kernel. To make this possible, this work aims to develop methods to connect the mathematical model specifying the behavior of tasks on Linux to the real trace of the Linux kernel, using Linux tracing features such as `perf` and `ftrace`.

## 2.4 FINAL REMARKS

The success of Linux as a general-purpose operating system (GPOS) motivated its extension to be a RTOS. Over the last decade, Linux has gained a lot of real-time features and improvements. Nowadays, Linux can perform scheduling decisions in the microseconds order, and this is motivating its usage on a new class of safety-critical systems. However, the timing analysis of Linux is based on an empirical evaluation, and such results are not enough to satisfy the safety requirements necessary in such a class of systems.

To improve the runtime analysis and verification of the Linux kernel, a new set of methods needs to be developed, based not only on empirical testing but using sophisticated techniques from the theory of formal methods and real-time scheduling systems.

The timing analysis of the PREEMPT RT Linux kernel using the real-time scheduling terminology, presented in Section 2.2.5, represented an improvement in the understanding of Linux and validated the idea of using the tracing features of Linux to enlighten its timing behavior. The weak point of the approach was the informal method

used in the inspection and explanation of the trace.

As seen in Section 2.3, the usage of formal methods is highly recommended in the specifications of safety-critical systems, mainly on the verification of the properties of the system. In the model-based approach, the verification is conducted using an abstract model of the real system. The model is a simplified representation of the system. It aims to reduce its complexity, enabling the usage of more specialized methods that would not be possible in a complete and complex description of the system.

The previous usage of tracing in the analysis of Linux witnesses the event-based nature of kernel. It also evidences the possibility of using the discrete event systems theory to create an abstract model of tasks on Linux, focusing on the events that influence the timing behavior of tasks, from the real-time scheduling theory point of view. From the formalisms available for the modeling of discrete event systems, the automata showed to be a viable option during early experiments. The graphical representation of an automaton hides the complexity of its formal notation, simplifying the understanding and modeling of a system. Moreover, automata operations enable creating a complex model from the composition of smaller and specialized models. The modular approach uses these operations to model a system from a set of independent generators and specifications.

The abstract model of the Linux tasks has the potential to reduce the complexity of kernel code while enabling more sophisticated reasoning about the timing behavior of the system, and the application of runtime verification for the Linux kernel. Before starting exploring the creation of the model and its usage in the analysis and verification of Linux, a set of related work is presented in the next chapter.

# 3 RELATED WORK

This Chapter presents prior literature relevant to the work being presented in this thesis, spanning across three main areas: 1) formal methods applied to operating systems kernel; 2) use of automata in real-time and operating systems analysis; and 3) real-time Linux latency. The first two topics are fundamental for the development of the work presented in Chapters 4, and 5, while the last shows the prior work relevant to Chapter 6.

## 3.1 FORMAL METHODS FOR OS KERNELS

A particularly challenging area is the one of verification of an operating system kernel and its various components. Some works that addressed this problem include the BLAST tool (HENZINGER et al., 2002), where control flow automata have been used, combining existing techniques for a state-space reduction based on abstraction, for model checking verification of `C` code using 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, the 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, inserting a conditional jump to an error handling piece of code, and a model of the surrounding kernel behavior to allow the model checker to verify whether or not the faulty code could ever be reached.

The static code analyzer SLAM (BALL; RAJAMANI, 2002b) shares the primary objectives with BLAST, in that it allows for analyzing `C` programs to detect violation of certain conditions. The SLAM uses a specification language named *specification language for interface checking (SLIC)* (BALL; RAJAMANI, 2002a). SLIC rules are composed by an static set of *states* (organized as `C` structures), a set of *events* and and its associated handler function, and a set of *annotations* that connects the specification with the objects instance in the code.

The Microsoft static driver verifier (SDV) toolset uses SLAM to verify the correct usage of Windows API on kernel modules, especially device drivers. SDV aims to be a completely automatic tool, to be used at compile time by developers. Anytime SDV finds a problem in a driver, it shows the execution path in the driver that leads to a rule violation. Despite the high level of automatization, the SDV authors do not claim it is a *push buttom* solution for verification. Instead, they mention that first, the specification of the properties to be verified needs to be modeled, which is not a straightforward step, requiring the understanding, and even the debugging, of the code. Secondly, the verification tooling needs to be integrated into the development environment to be prac-

tical. Last but not least, the problem of state space explosion reduces the application of the method for specific cases. The SDV authors clarify that the tool does not verify the complete operating system kernel, but sole the composition of some components of the system, such as the device drivers, with the highly abstract environmental model of the procedures of the Windows kernel (BALL; LEVIN; RAJAMANI, 2011).

Different authors also proposed a similar approach for the Linux kernel, resulting in three different toolsets: DDVerify, Avinux, and Linux driver verification (LDV).

(WITKOWSKI et al., 2007) proposed the DDVerify toolset, extending the capabilities of BLAST and SLAM, e.g., supporting interrupts, deferred tasks, timers, etc. It also includes specification rules about the usage of the synchronization primitives and the correct way to initialize variables before usage. The specifications are written in `C` code. It also relies on a set of self-developed scripts used in the extraction of information of the source files to generate a collection of information about the modules to be analyzed.

The Avinux (POST; KÜCHLIN, 2007) proposes an *public domain like* reimplementation of the SDV for the device drivers of Linux, using an extended version of the SLIC specification language, named SLICx.

Both DDVerify and Avinux use the CBMC (CLARKE; KROENING; LERDA, 2004) static verification tool, but DDVerify also supports SATABS (CLARKE; KROENING; SHARYGINA, et al., 2005). Moreover, both toolsets also present some integration with Eclipse IDE.

The LVD toolset aims to enable static code analysis for a diverse set of kernel modules (ZAKHAROV, I. S. et al., 2015). The LDV is a newer project and tries to overcome the limitations of the previously mentioned toolsets. For example, instead of supporting a fixed set of static verification tools, they proposed a dynamically and configurable set, including, but not limited to, BLAST, CPAchecker (BEYER; KEREMOGLU, 2011), UFO (ALBARGHOUTHI et al., 2012), and CBMC. Although it is mostly centered on Linux, its methodology can be extended for other OS kernels too. Regarding the specifications, they are done using the PI-process (ZAKHAROV, I. et al., 2013). The commit logs of kernel patches fixing problems found by the LDV are generally tagged with the following message:

```
Found by Linux Driver Verification project (linuxtesting.org).
```

This tag enables the collection of some interesting metrics. As of the version `5.6-rc6` of the Linux kernel, there are at least 351 kernel patches fixing problems found by LDV, from 18 different authors, with dates ranging from 2011 to 2019 [1]. These numbers confirm the practical approach of LDV.

[1] Run `git log -grep "Found by Linux Driver Verification project"` in a git tree of the kernel to see the kernel commit logs related to LDV.

Chaki et al. (CHAKI, S. et al., 2004) 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 the 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 (CHAKI, Sagar et al., 2005), 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 (KLEIN et al., 2009), 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 (BLACKHAM et al., 2011). These findings were possible thanks to the simplicity of the seL4 micro-kernel features, e.g., semi-preemptability.

*Remarks*

The works presented in this section, employing the static code analyzers of part of OS kernels, evidence the applicability of formal methods in this field. The usage of such technique is generally limited either to a specific interface between the kernel and their modules/drivers (SDV, DDVerify, Avinux, and LVD); or to a single function of the system (MAGIC); or applied on specialized/embedded OS kernels (seL4). Such restriction, as reported in (BALL; LEVIN; RAJAMANI, 2011), is a constraint to avoid state explosion.

However, the model proposed in this thesis is not limited to a single subsystem of the kernel, or a single function, but are the synchronization mechanisms embedded in the entire code. There are also relations between the events that are not written in the code. For instance, there is no annotation limiting the usage of mutexes to the preemptive sections of the code. Such cases would have to be handled via a manual specification of the system, which is precisely the purpose of this work.

Finally, the suggestion of using static code analysis to automatize the creation of the model was frequently raised during the development of this thesis. The answer

to this question received special attention and is specially addressed in Section 4.3.1.

### 3.1.1 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 (CORBET, J., 2006) 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 detecting the mistaken order of acquisition of multiple (nested) locks throughout multiple kernel code paths and detecting 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 have to be kept by the kernel is reduced by applying the technique based on locking classes rather than individual locks.

In (ALGLAVE et al., 2018), 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 the form of kernel code, which can be directly executed and tested.

Moreover, the well-known TLA+ formalism (LAMPORT, 1994) 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 (MARINAS, 2018). These recent results raised interest in the potential of the usage of formal methods in the development of Linux.

*Remarks*

`Lockdep` is used on a daily bases in the kernel development, and is responsible for countless bug fixes, and mainly bug prevention, as developers generally use it even before proposing patches. Although `lockdep` does not use a formal specification format, it effectively creates a monitor and conduces a runtime check of the system. Such a runtime verification like approach is needed because the context, and the order, in which the locks are taken is highly dependant on the workload of the system, and the complexity to create a synthetic workload that represents all the user-cases

is, at least, unreasonable. During the discussions with the Linux kernel community[2], developers agreed with the possibility of a formal version of `lockdep`, based on the approach presented in Chapter 5. Indeed, part of the model proposed in this thesis already overlaps with `lockdep`, as shown with the problem found with the model in Section 4.5.3.

The LKMM model and the usage of TLA+ confirms the trend of using model-based verification of the kernel, motivating the creation of other models, like the model proposed in this thesis.

## 3.2 AUTOMATA-BASED REAL-TIME SYSTEMS ANALYSIS

Automata and discrete-event systems theory have been extensively used to verify the timing properties of real-time systems. For example, in (WANG; LI; WONHAM, 2016), 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 (YOVINE, 1997; BOUAJJANI; TRIPAKIS; YOVINE, 1997; DAWS; YOVINE, 1995), the Kronos tool is used for checking properties of models based on multi-rate and parametric/symbolic timed automata.

In (CIMATTI; PALOPOLI; RAMADIAN, 2008), 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 the symbolic analysis of timed automata (FERSMAN; PETTERSSON; YI, 2002), 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 (LE et al., 2013), where verification has been carried out using the UPPAAL model checker (LI et al., 2004). A similar methodology can be found in (ARTHO; ÖLVECZKY, 2014), 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 (ANDRÉ et al., 2012). Similar in purposes is also the work in (LARSEN et al., 2014), 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 (LAMPKA; PERATHONER; THIELE, 2013) considered composability of automata-based timing specifications, so that timing properties of a complex real-time system can be verified with reduced complexity. Interesting work is presented in (SUN; LIPARI, 2014), where the authors use *Linear Hybrid Automaton* to model an exact schedulability analysis for the fixed priority scheduler, considering preemptive tasks in

[2] Discussions held at the Linux Plumbers Conference 2019 and the Kernel Recipes 2019.

a multiprocessor system.

Similarly to the approach of UPPAAL (LI et al., 2004), the TIMES tool has been used (AMNELL T. A. F., E. et al., 2004) with an automata-based formalism to describe a network of distributed real-time components for analyzing their temporal behavior from the viewpoint of schedulability.

Further works exist introducing mathematical frameworks for analyzing real-time systems, such as in (KAYNAR et al., 2003; LYNCH; SEGALA; VAANDRAGER, 2003), making use of Hybrid and Timed Input/Output Automata to prove safety, liveness and performance properties of a system.

*Remarks*

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, 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 commonly used in the real-time analysis literature.

### 3.2.1 Automata-based models for Linux

In (POSADAS et al., 2010), 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 model, respectively.

Matni and Dagenais (MATNI; DAGENAIS, 2009) proposed using automata to analyze traces generated by the kernel of an operating system. Automata is used to describe patterns of problematic behavior. An off-line analyzer checks for their occurrences. It uses the Linux Trace Toolkit next generation (LTTng) to search for three scenarios: chroot jail escape, locking validation, and real-time constraints checking.

The usage of trace and automata to verify conditions in the kernel is also presented in (MATNI; DAGENAIS, 2009). 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 (SPEAR; LEVY; DESNOYERS, 2012).

An important area that makes use of a formal definition of the system is the State based/Stateful robustness testing (LEI et al., 2010). Robust testing is a fault tolerance technique (PULLUM, 2001) also applied in the OS context. In (COTRONEO; DI LEO,

et al., 2011), 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 (COTRONEO; LEO, et al., 2013), 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 TIMEOUT approach (SHAHPASAND; SEDAGHAT; PAYDAR, 2016) 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.

*Remarks*

The model presented (POSADAS et al., 2010), is a high-level one and does not consider the internal details of the Linux kernel. Similarly, the models presented in (MATNI; DAGENAIS, 2009) are proof of concept 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.

The possibility of extracting models from the operating system, like SABRINE and TIMEOUT, 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, or inside the values of each variable introduced in Chapter 6.2. For example, the model presented in this paper identifies the synchronization events that cause a 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. It is worth to note that the overhead involved in tracing all functions is not negligible, and can considerably influence in the results of

the timing analysis, even more than the sole verification of the system, as presented in Chapter 5. A further discussion about the automatic creation of models is presented in Section 4.3.1.

These prior works demonstrate that the usage of automata format to test, verify, and explain the core dynamics of an operating system is not a new idea. Instead, it is a known field and motivates this thesis, which proposes a different model for different goals.

## 3.3 REAL-TIME LINUX LATENCY

Abeni et al. (ABENI et al., 2002) defined a metric similar to `cyclictest`, called OS latency that quantifies the difference between the actual schedule produced by the kernel and the ideal schedule, evaluating various OS latency components of several standards and real-time Linux kernels existing at the time (2002). They also examined the effects of the OS latency on a real application, an audio/video player, concluding that full kernel preemptability can significantly improve the real-time performance of Linux. During the 18 years since that paper indeed several aspects of Linux preemptability were enhanced.

Cerqueira and Brandenburg (CERQUEIRA; BRANDENBURG, 2013) described experiments with `cyclictest` to evaluate the scheduling latency experienced by real-time tasks under LITMUS$^{RT}$, vanilla Linux and Linux with the PREEMPT_RT patch. Three scenarios were considered: idle system, CPU-bound background workload, and I/O-bound background workload. Several histograms of observed scheduling latency in different scenarios and test conditions are presented. These histograms are the results of measuraments made with `cyclictest` adpted to LITMUS$^{RT}$. The authors also discussed the advantages and limitations of using `cyclictest` as a metric for estimating a system's capability to provide temporal guarantees. A similar experimental study is presented in (FAYYAD-KAZAN; PERNEEL; TIMMERMAN, 2013).

Reghanzani et al. (REGHENZANI; MASSARI; FORNACIARI, 2017) empirically measured the latencies of a real-time Linux system under stress conditions in a mixed-criticality environment. They also indicate the use of tracing (i.e., `feathertrace`) to obtain fine-grained measurements. No detailed description of the source of the measured delays is provided in the paper.

Herzog et al. (HERZOG et al., 2018) presented a tool that systematically measures interrupt latency, at run-time, in the Linux vanilla kernel. It is based on tracepoint. Examples are provided for two distinct platforms (ARM and Intel). The tool is used to measure interrupt latencies, identify jitter causes, and reveal inter-dependencies between interrupt handlers and user-space workloads. No attempt is made to model Linux kernel scheduling. Regnier et al. (REGNIER; LIMA; BARRETO, 2008) presented an evaluation of the timeliness of interrupt handling in Linux.

The `ftrace`'s `preemptirqsoff` tracer (ROSTEDT, Steven, 2009) enables the tracing of function when either preempt or IRQs are disabled, trying to capture the longest window.

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 (B. B. BRANDENBURG, 2011). 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. These figures are injected into overhead-aware real-time analysis techniques.

*Remarks*

The scheduling latency is, undoubtedly, the main metric for the PREEMPT_RT, and so motivated many different works. However, most of them considered Linux as a *black-box*, trying to observe the consequences of the synchronization in the delay, but not the root causes, as in (ABENI et al., 2002; CERQUEIRA; BRANDENBURG, 2013; REGHENZANI; MASSARI; FORNACIARI, 2017).

Some other works try to find the root cause but based only on empirical analysis of the system. The approach in (ROSTEDT, Steven, 2009) does not differentiate between interference due to interrupts and the contribution due to different code segments disabling preemption or interrupts. However, by adding tracing of functions, it adds overhead to the measurement, thus potentially heavily affecting the result, often mispointing the real source of the latency.

The discussion about outliers in (B. B. BRANDENBURG, 2011), along with the explicit admission of the need for removing some of them manually throughout the experimentation, witnesses the need for a more insightful model that provides more accurate information of those overheads.

Regarding latency, this thesis targets explaining, at a finer-grained level of detail, what these scheduling overheads are, where they originate from, and why. This work aims to enable a more analytical view of Linux, fulfilling the part of the gap that exists between real-time Linux and real-time scheduling theory.

## 3.4 FINAL REMARKS

This set of related work demonstrates the value of the endeavor of applying formal methods to improve Linux. SDV, LDV are practical examples of the potential of such techniques. It is also clear that no single solution will solve all the problems: the state explosion requires a meticulous selection of the goals and acceptable limitations

of each methodology. The runtime verification-like method used on `lockdep` inspires the development of a similar approach based on a formal set of specifications.

The usage of automata for the description and analysis of Linux was also explored. However, the vast majority of the work uses straightforward models for very specialized specifications. The complexity of the presented models is not comparable with the ones presented by this thesis, so they do not require the use of a more sophisticated approach. Still, the previous work demonstrates that the path taken was not unknown and motivates further exploration.

Finally, the fact that real-time latency is still measured using a *black-box* approach shows that the constraints imposed by the synchronization to the Linux tasks are indeed complex. The complexity, as shown in the next chapter, is given by the impressive number of events and possible sequences that they can have. The translation of the informal knowledge of Linux, to the level of precision utilized in the real-time scheduling theory, requires an intermediary step, removing the code complexity, while enabling the formal definition of the specification and properties of the synchronization of the task of Linux.

# 4 A THREAD SYNCHRONIZATION MODEL FOR THE PREEMPT_RT KERNEL

Despite all developments made on real-time Linux presented in Section 2.2, 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* (BRANDENBURG, B. B.; ANDERSON, J. H., 2007; CALANDRINO et al., 2006) 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 putting in practice (GLEIXNER, 2010). On the other hand, Linux was not designed as a real-time 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 (BRANDENBUG; ANDERSON, 2009).

The developers of Linux observe and debug their timing properties using the tracing features present in the kernel (ROSTEDT, S., 2011; SPEAR; LEVY; DESNOYERS, 2012; TOUPIN, 2011; BRANDENBURG, Bjorn B.; ANDERSON, James H., 2007). 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` (ROSTEDT, Steven, 2010) 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.

This chapter presents 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 also covers kernel code related to locking mechanisms, such as mutexes, read/write semaphores, and read/write locks, including the possibility of nested locks, such as in the locking primitives' own code (OLIVEIRA, Daniel Bristot de et al., 2018).

This chapter 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*

[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`

*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.

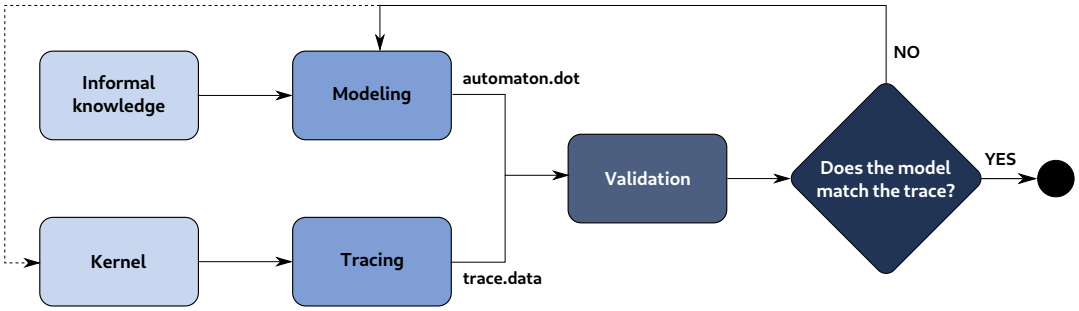
## 4.1 MODELING APPROACH

Following the approach presented in Figure 26, 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 (ROSTEDT, Steven, 2010), the kernel code analysis, academic documentation about Linux and real-time systems (OLIVEIRA; OLIVEIRA, 2016) (B. B. BRANDENBURG, 2011), and hardware documentation. 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 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*.

## 4.2 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

Figure 26 – Modeling approach.




level of abstraction used in the model 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. The level of abstraction was also highly influenced by the previous experience, described in Section 2.2.5.

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 (OLIVEIRA, 2018e) and The Linux Plumbers Conference 2018 (OLIVEIRA, 2018f,d).

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

Table 4 – Interrupt 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 |

Table 5 – 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 |

new `tracepoints` were added to the kernel.

Tables 4, 5 and 6 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) (CORBET, J., 2018). 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 (MCKENNEY, 2005). Moreover, the scheduling and locking related events are implementation independent. For instance,

Table 6 – 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 |

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.

## 4.3 MODELING

The automata model was developed using the Supremica IDE (AKESSON et al., 2006). 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` (ELLSON et al., 2002), for example.

The model was developed using the modular approach. All generators and specifications 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 27 and 28[3]. 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 7 shows statistics about the generators and specifications that compose the Model. The complete Model is generated from the parallel composition of all gen-

[3] The *generators* and *specifications* of the model are presented during the development of next Chapters, as required.

Table 7 – 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 27 – Examples of generators: *G05* need resched (left) and *G04* Scheduling context (right).

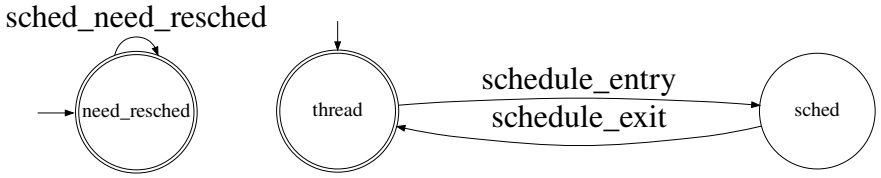
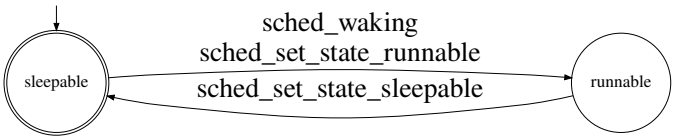



Figure 28 – Examples of generators: *G01* sleepable and runnable.




erators 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.

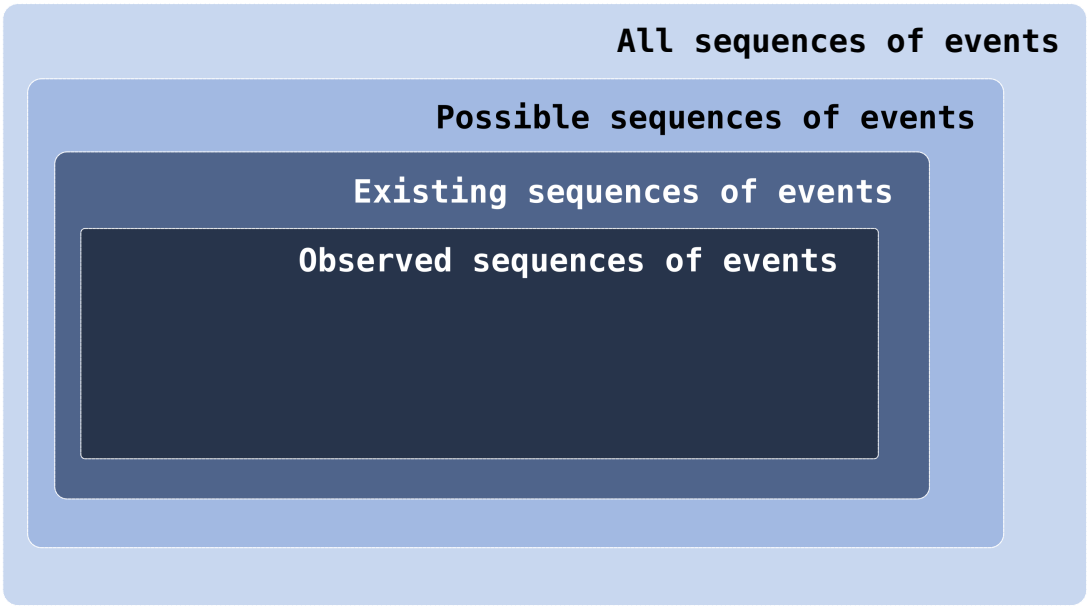
### 4.3.1 Automate or not to automate the model creation?

One frequent question made during the development of this work was: *Is it possible to automatically create a model from the trace output? For instance, tools such as SABRINE (COTRONEO; LEO, et al., 2013) 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.

To help in the explanation, cosider the set of events presented in Figure 29 as a reference:

- *all sequences of events* is the cartesian product of all events.
- the *possible sequences of events* are the sequences that can occur in the system

Figure 29 – Sets of sequences of event.



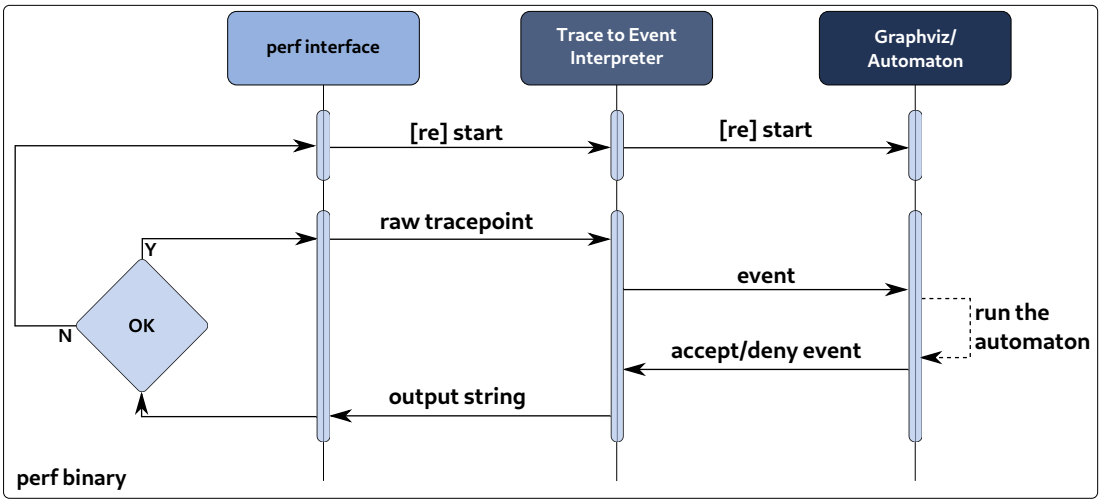

without violating any specification.

- the *existing sequences of events* are the sequences that exist in the current code of Linux;
- the *observed sequences of events* are the sequences that a given trace of the execution of the system shows.

The modeling approach presented in this work 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.

The validation approach of this work 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*, as they do not violate any of the specifications. 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

Figure 30 – `perf task_model report` dynamic.




as interrupts get enabled before calling the scheduler, because the kernel will not break if such a sequence appears in the future.

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

## 4.4 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

Figure 31 – `perf_tool` definition inside the `task_model` structure.

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

Figure 32 – `task_model` and *perf_tool* initialization.

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

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, 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 30. When starting, `perf` interface opens the trace file, and uses the `Graphviz` library[4] 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 31 and 32. Two callbacks implement the tool, one to handle tracing samples, and another to no-

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

Figure 33 – `perf thread_model`: Events to callback mapping.

```
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 34 – Handler for the `irq_vectors:nmi_entry` `tracepoint`.

```
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;
}
```

tify 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 33.

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 34 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 35, is used to run the automaton. If the automaton 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 32), 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

Figure 35 – *process_event*: trying to run the automata.

```
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 36 – Example of the `perf thread_model` output: a thread activation.

```
 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
```

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 36.

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. 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 4.5. The source code of the model in the format used by `Supremica`, the kernel patch with kernel and

Figure 37 – Kernel trace excerpt.

```
 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]
```

`perf` modifications and more information about how to use the model and reproduce the experiments are available at the paper's page[5].

## 4.5 OFFLINE RUNTIME VERIFICATION

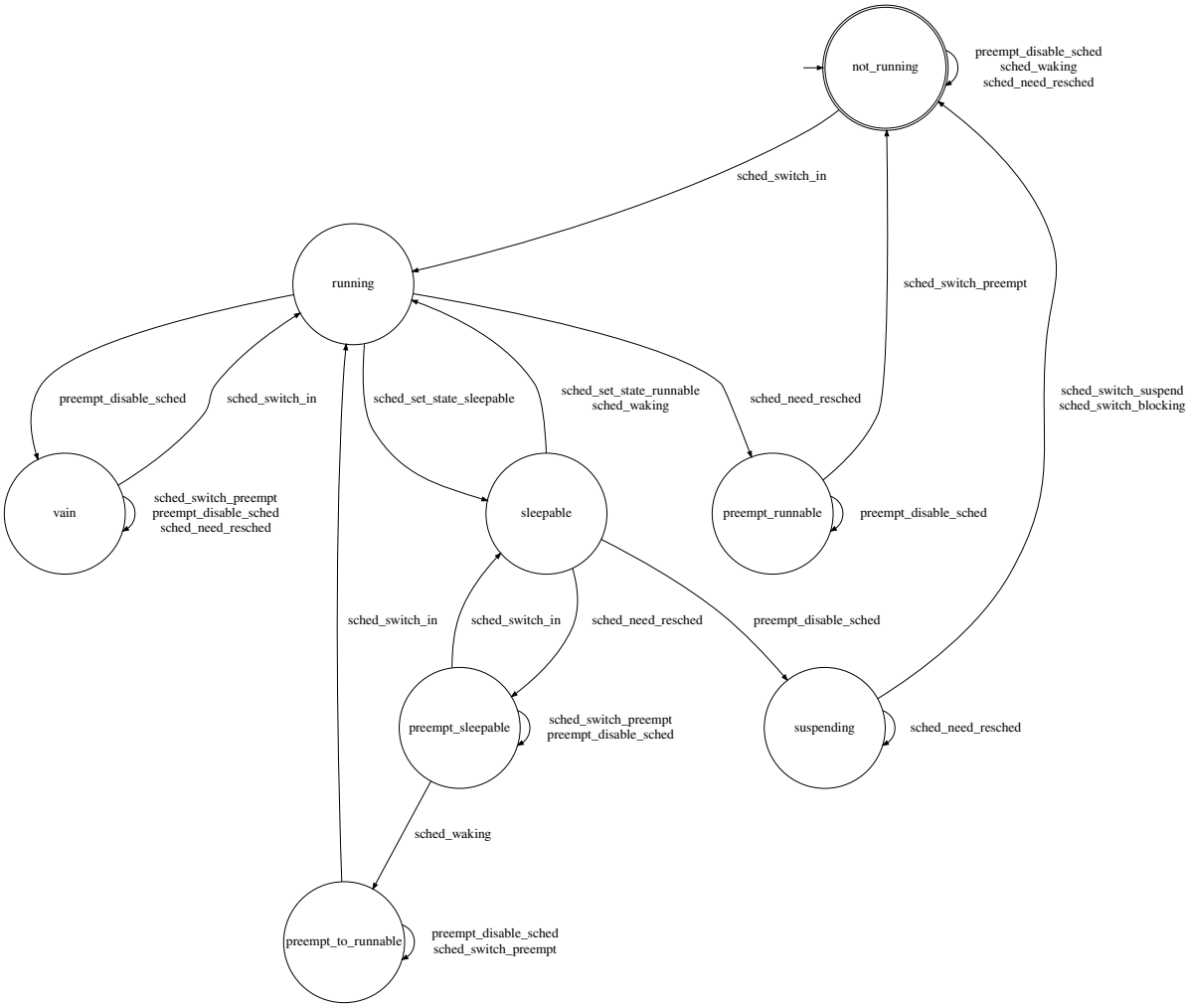
This section presents three problems found in the kernel while validating the model, in an example that validates the usage of the model for offline runtime verification of the kernel. The first problem shows an optimization case, the second is a problem in the tracing, and the third is regarding an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

### 4.5.1 Scheduling in vain

In Linux, the main scheduler function (`__schedule()`) is always called with preemption disabled, as shown in Figure 70. In the model, it can be seen as the event that precedes a scheduler call. The specification in Figure 38 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

[5] See: `https://bristot.me/linux_task_model/`

Figure 38 – *S18* Scheduler call sufficient and necessary conditions.




occurrence of the event `sched_set_state_runnable`, and so the scheduler will not *necessarily* be called.

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 `preempt_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 37, considering the thread `kworker/0:2` in analysis, and the model in Figure 38 in the initial state, the events and state transitions of Table 8 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

Table 8 – Events and state transitions of Figure 37.

| Line | Event | New state |
|---|---|---|
| 1 | sched_switch_in | running |
| 2 | sched_set_state_sleepable | sleepable |
| 3 | sched_need_resched | preempt_sleepable |
| 4 | preempt_disable_sched | preempt_sleepable |
| 6 | sched_switch_preempt | preempt_sleepable |
| 7 | sched_waking | preempt_to_runnable |
| 8 | sched_switch_in | running |
| 9 | preempt_disable_sched | vain |

switched the context in preemption and left the processor. At Line 7 the thread is awakened, switching the state to `preempt_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 37, 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 (OLIVEIRA, 2018a).

### 4.5.2 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 39.

Both mentioned events require the preemption and IRQs to be disabled, as modeled in Figure 74, 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 40 shows the raw trace from kernel for the case evaluated in Figure 39, 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

Figure 39 – Missing kernel events: the output of `perf thread_model`.

```
. 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 40 – Missing kernel events: the output of kernel `tracepoints`.

```
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 41 – Pseudo-code of tracing recurrence.

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

                /* Trace Recursion */
```

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 (OLIVEIRA, 2018c). 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 41.

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 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

Figure 42 – Trace excerpt with comments of where the IRQ context is identified in the trace.

```
 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 43 – `mutex_lock` not permitted with interrupts disabled.

```
 + 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 =====
```

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 42. 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 (OLIVEIRA, 2019b). It involves detecting the current task context before executing any `C` code.

### 4.5.3 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 43 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 67 and 45. The raw trace showed that a real-time mutex was being taken in the timer interrupt, as shown in Figure 46. The interrupt in case was the timer interrupt,

Figure 44 – *S12* Events blocked in the IRQ context.

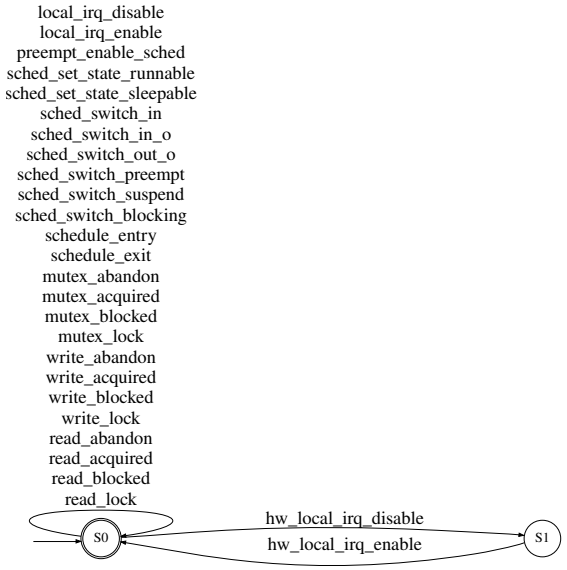



Figure 45 – *S22* Lock while interruptible.

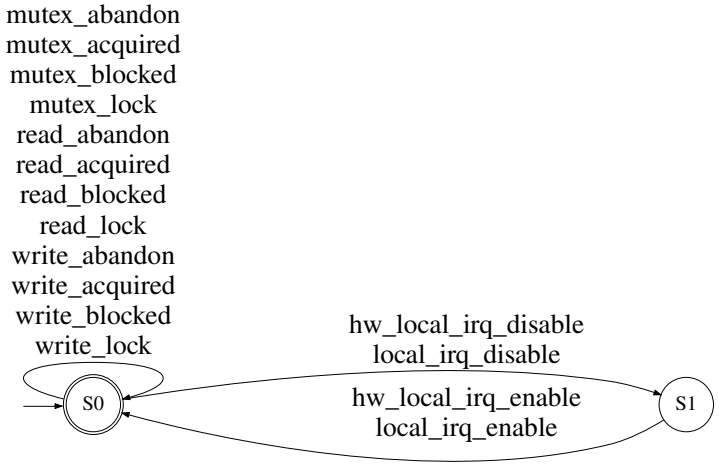



Figure 46 – Trace of `mutex_lock` taken in the timer interrupt handler.

```
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 47 – Function stack, from the timer IRQ to the `mutex_lock`, used in the report for the Linux kernel developers.

```
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()    {
```

while running the watchdog timer. Figure 47 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 verified against the *4.14-rt* kernel (OLIVEIRA; CUCINOTTA; OLIVEIRA, 2019), 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 (OLIVEIRA, 2019a).

## 4.6 FINAL REMARKS

Linux is a sophisticated 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.

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 4.5 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.

The idea of using the automata model to verify the kernel was presented to the leading Linux kernel developers, and there is a consensus that the approach should be integrated into the kernel code, mainly to improve testing of the logical correctness of the kernel (OLIVEIRA, 2018d), but also for timing regressions, with the creation of new metrics for the PREEMPT_RT kernel (OLIVEIRA, 2018b). The sole limitation to offline verification using `perf` is its efficiency: the need to copy a large volume of trace data to the user-space is unpractical for many use-cases. The next chapter presents a method that transforms the offline verification presented in this chapter into online runtime verification, processing the events in-kernel, with overheads that turn its usage possible even in production systems.

# 5 ONLINE RUNTIME VERIFICATION

Real-time variants of the Linux OS have been successfully used in many safety-critical and real-time systems belonging to a wide spectrum of applications, going from sensor networks (DUBEY; KARSAI; ABDELWAHED, 2009), robotics (GUTIÉRREZ et al., 2018), factory automation (CUCINOTTA et al., 2009) to the control of military drones (CONDLIFFE, 2014) and distributed high-frequency trading systems (CORBET, J., 2010; CHISHIRO, 2016), just to mention a few. However, for a wider adoption of Linux in next-generation cyber-physical systems, like self-driving cars (LINUX FOUNDATION, 2016), automatic testing and formal verification of the code base is increasingly becoming a non-negotiatable requirement. However, Linux lacks a methodology for runtime verification that can be applied broadly throughout all of the in-kernel subsystems.

The approach presented in Section 4.5 relies on tracing events into an in-kernel buffer, then moving the data to user-space where it is saved to disk, for later post-processing. Although functional, when it comes to tracing high-frequency events, the act of in-kernel recording, copying to user-space, saving to disk and post-processing the data related to kernel events profoundly influences the timing behavior of the system. For instance, tracing scheduling and synchronization-related events can generate as many as 900000 events per second, and more than 100 MB per second of data, per CPU, making the approach non-practical, especially for big muti-core platforms.

An alternative could be hard-coding the verification in the Linux kernel code. This alternative, however, is prone not to become widely adopted in the kernel. It would require a considerable effort for acceptance of the code on many subsystems. Mainly because complex models can easily have thousands of states. A second alternative would be maintaining the verification code as an external *patchset*, requiring the users to recompile the kernel before doing the checking, what would inhibit the full utilization of the method as well. An efficient verification method for Linux should unify the flexibility of using the dynamic tracing features of the kernel while being able to perform the verification with low overhead.

This chapter presents an efficient automata-based verification method for the Linux kernel, capable of verifying the correct sequences of in-kernel events as happening at runtime, against a theoretical automata-based model that has been previously created. The method starts from an automata-based model, as produced through the well-known Supremica modeling tool, then it auto-generates `C` code with the ability of efficient transition look-up time in *O(1)* for each hit event. The generated code embedding the automaton is compiled as a module, loaded *on-the-fly* into the kernel and dynamically associated with kernel tracing events. This enables the run-time verification of the observed in-kernel events, compared to the sequences allowed by the model, with any mismatch being readily identified and reported. The verification is carried out

Figure 48 – Verification approach.

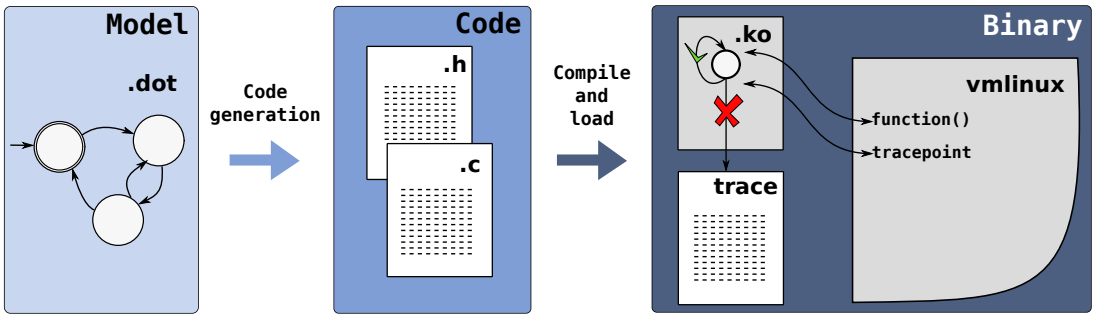
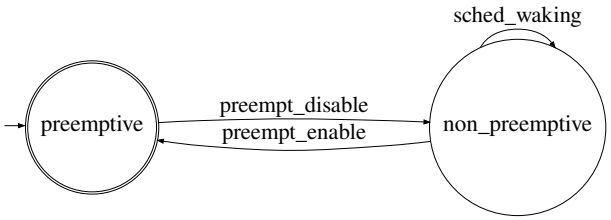



Figure 49 – Wake-up In Preemptive (*WIP*) Model.




in kernel space way more efficiently than it was possible to do in user-space, because there is no need to store and export the whole trace of occurred events. Indeed, results from performance analysis of a kernel under verification show that the overhead of the verification of kernel operations is very limited, and even lower than merely activating tracing for all of the events of interest.

## 5.1 EFFICIENT FORMAL VERIFICATION FOR THE LINUX KERNEL

An overarching view of the approach being proposed in this paper is displayed in Figure 48. It has three major phases. First, the behavior of a part of the Linux kernel is modeled using automata, using the set of events that are available in the tracing infrastructure[1]. The model is represented using the *.dot* `Graphviz` format (ELLSON et al., 2002). The *.dot* format is open and widely used to represent finite-state machines and automata. For example, the Supremica modeling tool (AKESSON et al., 2006) supports exporting automata models using this format.

Figure 49 presents the example of an automaton for the verification of in-kernel scheduling-related events. The model specifies that the event *sched_waking* cannot take place while preemption is enabled, in order not to cause concurrency issues with the scheduler code.

In the second step, the `.dot` file is translated into a `C` data structure, using the `dot2c` tool [2]. The auto-generated code follows a naming convention that allows it to be

[1] These can be obtained for example by running: `sudo cat /sys/kernel/debug/tracing/available_events`.

[2] The tools, the verification modules, the BUG report, high-resolution figures and FAQ are available in the companion page (OLIVEIRA, 2019c).

linked with a kernel module skeleton that is already able to refer to the generated data structures, performing the verification of occurring events in the kernel, according to the specified model. For example, the automaton in Figure 49 is transformed into the code in Figure 50.

Figure 50 – Auto-generated code from the automaton in Figure 49.

```
enum states {
        preemptive = 0,
        non_preemptive,
        state_max
};

enum events {
        preempt_disable = 0,
        preempt_enable,
        sched_waking,
        event_max
};

struct automaton {
        char *state_names[state_max];
        char *event_names[event_max];
        char function[state_max][event_max];
        char initial_state;
        char final_states[state_max];
};

struct automaton aut = {
        .event_names = { "preempt_disable", "preempt_enable",
                         "sched_waking" },
        .state_names = { "preemptive", "non_preemptive" },
        .function = {
                        { non_preemptive,         -1,             -1 },
                        {             -1, preemptive, non_preemptive },
                    },
        .initial_state = preemptive,
        .final_states = { 1, 0 }
};
```

The `enum states` and `events` provide useful identifiers for states and events. As the name suggests, the `struct automaton` contains the automaton structure definition. Its corresponding `C` version contains the same elements of the formal definition. The most critical element of the structure is `function`, a matrix indexed in constant time *O(1)* by `curr_state` and `event` (as shown in the `get_next_state()` function in Figure 51). Likewise, for debugging and reporting reasons, it is also possible to translate the event and state indexes into strings in constant time, using the `state_names` and `event_names` vectors.

Regarding scalability, although the matrix is not the most efficient solution with respect to the memory footprint, in practice, the values are reasonable for nowadays common computing platforms. For instance, the thread synchronization model from Chapter 4, with 9017 states and 20103 transitions, resulted in a binary object of less than 800KB, a reasonable value even for nowadays Linux-based embedded systems. The automaton structure is static, so no element changes are allowed during the ver-

Figure 51 – Helper functions to get the next state.

```
char get_next_state(struct automaton *aut, enum states curr_state,
                    enum events event) {
        return aut->function[curr_state][event];
}
```

ification. This simplifies greatly the needed synchronization for accessing it. The only information that changes is the variable that saves the *current state* of the automata, so it can easily be handled with atomic operations, that can be a single variable for a model that represents the entire system. For instance, the model in Figure 49 represents the state of a CPU (because the preemption enabling status is a *per-cpu* status variable in Linux), so there is a *current state* variable *per-cpu*, with the cost of (*1 Byte * the number of CPUs of the system*). The simplicity of automaton definition is a crucial factor for this method: all verification functions are *O(1)*, the definition itself does not change during the verification and the sole information that changes has a minimal footprint.

In the last step, the auto-generated code from the automata, along with a set of helper functions that associate each automaton event to a kernel event, are compiled into a kernel module (a `.ko` file). The model in Figure 49 uses only `tracepoints`. The *preempt_disable* and *preempt_enable* automaton events are connected to the `preemptirq:preempt_disable` and `preemptirq:preempt_enable` kernel events, respectively, while the *sched_waking* automaton event is connected to the `sched:sched_waking` kernel event. The Sleeping While in Atomic (*SWA*) model in Figure 52 also uses tracepoints for *preempt_disable* and *enable*, as well as for *local_irq_disable* and *enable*. But the *SWA* model also uses `function tracers`.

One common source of problems in the `PREEMPT_RT` Linux is the execution of functions that might put the process to sleep, while in a non-preemptive code section. The event *might_sleep_function* represents these functions. At initialization time, the *SWA* module *hooks* to a set of functions that are known to eventually putting the thread to sleep.

Note that another noteworthy characteristic of the proposed framework is that, by using user-space probes (DRONAMRAJU, 2019), it is also possible to perform an integrated automata-based verification of both user and kernel-space events, without requiring code modifications.

The kernel module produced as just described can be loaded at any time during the kernel execution. During initialization, the module connects the functions that handle the automaton events to the kernel tracing events, and the verification can start. The verification keeps going on until it is explicitly disabled at runtime by unloading the module.

The verification output can be observed via the tracing file regularly produced by

Figure 52 – Sleeping While in Atomic (*SWA*) model.

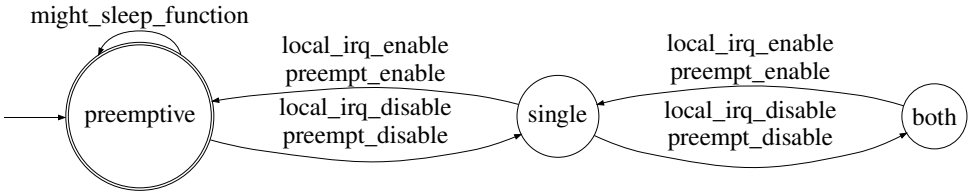



Figure 53 – Example of output from the proposed verification module, as occurring when a problem is found.

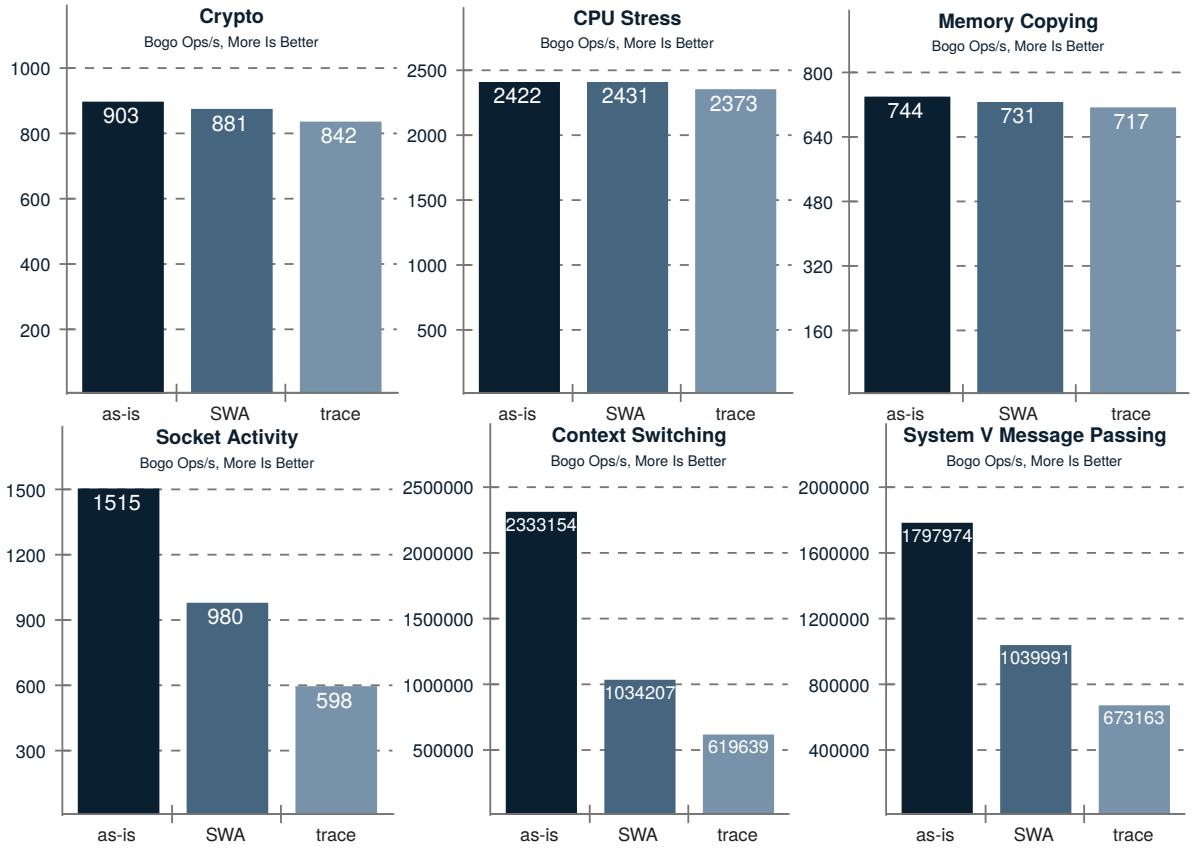
```
bash-1157  [003] ....2..   191.199172: process_event: non_preemptive -> preempt_enable = \\
                                         preemptive safe!
bash-1157  [003] dN..5..   191.199182: process_event: event sched_waking not expected in \\
                                         the state preemptive
bash-1157  [003] dN..5..   191.199186: <stack trace>
=> process_event
=> __handle_event
=> ttwu_do_wakeup
=> try_to_wake_up
=> irq_exit
=> smp_apic_timer_interrupt
=> apic_timer_interrupt
=> rcu_irq_exit_irqson
=> trace_preempt_on
=> preempt_count_sub
=> _raw_spin_unlock_irqrestore
=> __down_write_common
=> anon_vma_clone
=> anon_vma_fork
=> copy_process.part.42
=> _do_fork
=> do_syscall_64
=> entry_SYSCALL_64_after_hwframe
```

`Ftrace`. As performance is a major concern for runtime verification, debug messages can be disabled of course. In this case, the verification will produce output only in case of problems.

An example of output is shown in Figure 53. In this example, in Line 1 a *debug* message is printed, notifying the occurrence of the event *preempt_enable*, moving the automaton from the state *non_preemptive* to *preemptive*. In Line 2, *sched_waking* is not expected in the state *preemptive*, causing the output of the *stack trace*, to report the code path in which the problem was observed.

The problem reported in Figure 53 is the output of a *real bug* found in the kernel while developing this approach. The bug was reported to the Linux kernel mailing list, including the verification module as the test-case for reproducing the problem [2].

Figure 54 – Phoronix Stress-NG Benchmark Results: *as-is* is the system without tracing nor verification; *SWA* is the system while verifying *Sleeping While in Atomic* automata in Figure 56 and with the code in Figure 50; and the *trace* is the system while tracing the same events used in the *SWA* verification.




## 5.2 PERFORMANCE EVALUATION

Being efficient is a key factor for a broader adoption of a verification method. Indeed, an efficient method has the potential to increase its usage among Linux developers and practitioners, mainly during development, when the vast majority of complex testing takes place. Therefore, this section focuses on the performance of the proposed technique, by presenting evaluation results on a real platform verifying models, in terms of the two most important performance metrics for Linux kernel (and user-space) developers: *throughput* and *latency*.

The measurements were conducted on an HP ProLiant BL460c G7 server, with two six-cores Intel Xeon L5640 processors and 12GB of RAM, running a Fedora 30 Linux distribution. The kernel selected for the experiments is the Linux PREEMPT_RT version *5.0.7-rt5*. The real-time kernel is more sensible to synchronization as the modeled preemption and IRQ-related operations occur more frequently than in the mainline kernel.

### 5.2.1 Throughput evaluation

Throughput evaluation was made using the *Phoronix Test Suite* benchmark (PHORONIX, 2020), and its output is shown in Figure 54. The same experiments were repeated in three different configurations. First, the benchmark was run in the system *as-is*, without any tracing nor verification running. Then, it was run in the system after enabling verification of the *SWA* model. Finally, a run was made with the system being traced, only limited to the events used in the verified automaton. It is worth mentioning that tracing in the experiments means only recording the events. The complete verification in user-space would still require the copy of data to user-space and the verification itself, which would add further overhead.

On the CPU bound tests (Crypto, CPU Stress and Memory Copying), both trace and verification have a low impact on the system performance. In contrast, the benchmarks that run mostly on kernel code highlights the overheads of both methods. In all cases, the verification performs better than tracing. The reason is that, despite the efficiency of tracing, the amount of data that has to be manipulated costs more than the simple operations required to do the verification, essentially the cost of looking up the next state in memory in *O(1)*, and storing the next state with a single memory write operation.

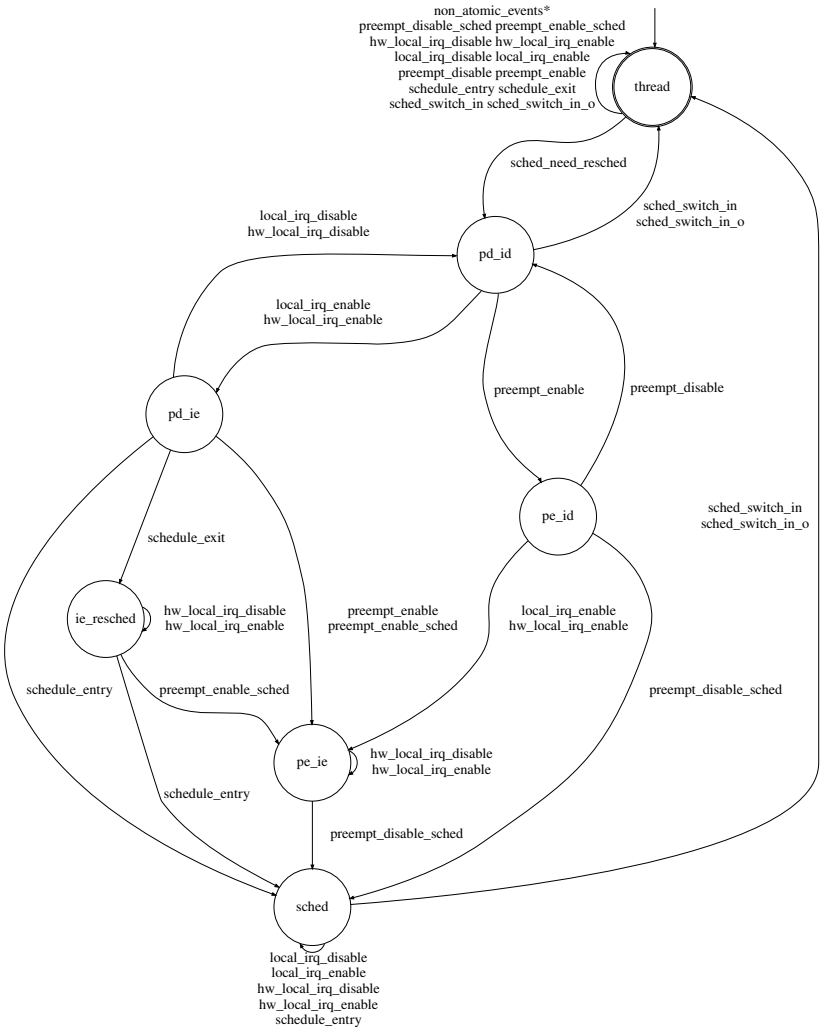
### 5.2.2 Latency evaluation

Latency is the main metric used when working with the PREEMPT_RT kernel. The latency of interest is defined as the delay the highest real-time priority thread suffers from, during a new activation, due to in-kernel synchronization. Linux practitioners use the `cyclictest` tool to measure this latency, along with `rteval` as background workload, generating intensive kernel activation.

Two models were used in the latency experiment. Similarly to Section 5.2.1, the *SWA* model was evaluated against the kernel *as-is*, and the kernel simply tracing the same set of events. In addition, the *Need Re-Schedule* (*NRS*) model in Figure 77 was evaluated. It describes the synchronization events that influence the latency, and is modeled as the *specification 19* from Chapter 4. The *NRS* measurements were made on the same system but configured as a single CPU.

Consistently with the results obtained in the throughput experiments, the proposed verification mechanism is more efficient than the sole tracing of the same events. This has the effect that the `cyclictest` latency obtained under the proposed method, shown in Figure 56 (*SWA*/*NRS* curves), is more similar to the one of the kernel *as-is* than what is obtained while just tracing the events.

Figure 55 – Need re-sched forces scheduling (*NRS* model).

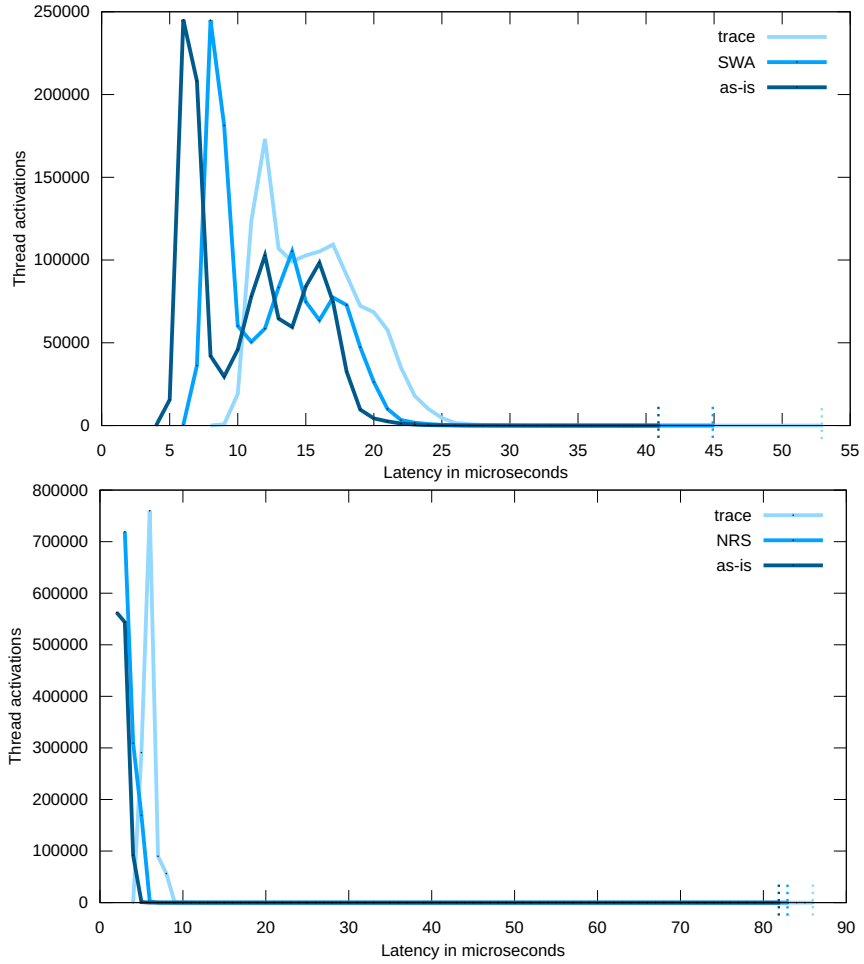



## 5.3 FINAL REMARKS

The increasing complexity of the Linux kernel code-base, along with its increasing usage in safety-critical and real-time systems, pushed towards a stronger need for applying formal verification techniques to various kernel subsystems. Nonetheless, two factors represent a barrier in this regard: 1) the need for complex setups, even including modifications and re-compilation of the kernel; 2) the excessively poor performance exhibited by the kernel while under tracing, for collecting data needed in the verification, typically carried out in user-space.

The solution for both problems seemed to be controversial: the usage of in-kernel tracing along with user-space post-processing reduces the complexity of the setup, but incurs the problem of having to collect, transfer to user-space and process large amounts of data. On the other hand, the inclusion of verification code “hard-coded” in the kernel requires more complex setups, with the need for applying custom patches and recompiling the kernel, with said patches being quite cumbersome to maintain as the kernel evolves over time.

Figure 56 – Latency evaluation, using the *SWA* model (top) and the *NRS* model (bottom).




This chapter tackled these two problems by using the standard tracing infrastructure available in the Linux kernel to dynamically attach verification code to a non-modified running kernel, by exploiting the mechanism of dynamically loadable kernel modules. Furthermore, the verification code is semi-automatically generated from standard automata description files, as can be produced with open editors. The presented benchmark results show that the proposed technique overcomes standard tracing and user-space processing of kernel events to be verified in terms of performance. Moreover, the proposed technique is more efficient than merely tracking the events of interest just using tracing features available in the kernel.

The results of this research attracted the attention of the Linux kernel development community. In addition to the academic conferences, the results of this chapter were presented at the main open-source conference venues, including the Linux Plumbers Conference 2019 and the Embedded Linux Conference Europe 2019, being an invited talk at the Kernel Recipes conference in 2019.

# 6 LATENCY ANALYSIS

This chapter presents the final results of the thesis, being the motivation for the development of the model. It aims to validate that the model developed in Section 4 has the potential to describe the PREEMPT_RT execution unambiguously, overcoming the main limitation of the timeline presented in (OLIVEIRA; OLIVEIRA, 2016), starting by the principal PREEMPT_RT metric: the *scheduling latency*.

Regarding the *scheduling latency*, Linux developers have extensively reworked the Linux kernel to reduce the code sections that could delay the scheduling of the highest-priority thread. `Cyclictest` is the primary tool adopted in the evaluation of the fully-preemptive mode of PREEMPT_RT Linux (CERQUEIRA; BRANDENBURG, 2013), and it is used to compute the time difference between an expected activation time and the actual start of execution time of a high-priority thread running on a CPU. By configuring the measurement thread with the highest priority and running a background taskset to generate disturbance, `cyclictest` is used in practice to measure the *scheduling latency* of each CPU of the system. Maximum observed latency values generally range from a few microseconds on single-CPU systems to 250 us on non-uniform memory access systems (RED HAT. INC, 2020), which are acceptable values for a vast range of applications with sub millisecond timing precision requirements. In this way, PREEMPT_RT Linux closely fulfills theoretical fully-preemptive systems assumptions that consider atomic scheduling operations, with neglectable overheads.

Despite its practical approach and the contributions to the current state-of-art of real-time Linux, `cyclictest` has some known limitations. The main limitation arises from the opaque nature of the latency value provided by `cyclictest` (BRANDENBUG; ANDERSON, 2009). Indeed, it only informs about the latency value, without providing insights on its root causes. Tracing features of Linux are often applied by developers to help in the investigation. However, the indiscriminate usage of tracing is not enough to solve the problem: the tracing overhead can easily mask the real sources of latency, and the excessive amount of data often drives the developer to conjunctures that are not the actual cause of the problem. For these reasons, the debug of a latency spike on Linux generally takes a reasonable amount of hours of very specialized resources.

A common approach in the real-time systems theory is the categorization of a system as a set of independent variables and equations that describe its integrated timing behavior. However, the complexity of the execution contexts and fine-grained synchronization of the PREEMPT_RT make it difficult the application of classical real-time analysis for Linux. Linux kernel complexity is undoubtedly a barrier for both expert operating system developers and real-time systems researchers. The absence of a theoretically-sound definition of Linux behavior is widely known, and it inhibits the application of the rich arsenal of already existing techniques from the real-time theory. Also,

it inhibits the development of theoretically sound analysis that fits all the peculiarities of the Linux task model (GLEIXNER, 2010).

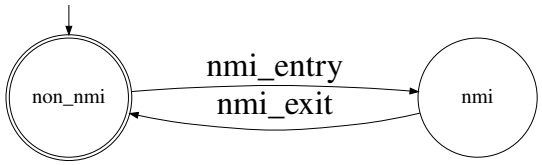
In this Chapter, the model presented in Chapter 4 is used to derive a set of properties and rules defining the Linux kernel behavior from a scheduling perspective. These properties are then leveraged to derive a theoretically-sound bound to the scheduling latency that comprehensively considers the sources of delays, including the all possible synchronization flows in the kernel code. The analysis builds upon a set of practically-relevant modeling variables inspired by the foundational principles behind the development of the PREEMPT_RT Linux Kernel. This chapter also presents an efficient tracing method, using the same approach presented in Chapter 5, to observe the kernel events, interpreting them to define observed values for the variables used in the analysis while reducing the runtime overhead and storage space to values that make its use feasible in practice. The tool also analyses the trace, serving to distinguish the various sources of the latency. Moreover, by exploring the interference caused by adopting different interrupt characterizations, it also derives possible latency bounds based on real trace execution. Finally, the experimental section compares the results obtained by the `cyclictest` and the proposed tool, showing that the proposed method can find sound bounds faster with acceptable overhead.

## 6.1 SYSTEM MODEL

This work uses the *Thread Synchronization Model*, presented in Chapter 4, as the description of a single-cpu PREEMPT_RT Linux system configured in the fully-preemptive mode. The advantages of using the model is many-fold: (1) it was developed in collaboration with kernel developers, and widely discussed with both practitioners (OLIVEIRA, 2018e,f) and academia (OLIVEIRA; CUCINOTTA; OLIVEIRA, 2019; OLIVEIRA, D. B. de et al., 2017); (2) the model is deterministic, i.e, in a given state a given event can cause only one transition; (3) the model was extensively verified, both for functional and non-functional properties; indeed, the non-functional runtime verification of the model was capable of identifying bugs in the Linux kernel that no other kernel mechanism could discover (OLIVEIRA; OLIVEIRA; CUCINOTTA, 2020) but mainly; (4) it abstracts the code complexity by using a set of small automata, each one precisely describing a single behavior of the system.

The model's task set is composed of a single NMI $\tau^{\text{NMI}}$, a set $\Gamma^{\text{IRQ}} = \{\tau_1^{\text{IRQ}}, \tau_2^{\text{IRQ}}, \ldots\}$ of maskable interruptions (IRQ for simplicity), and a set of threads $\Gamma^{\text{THD}} = \{\tau_1^{\text{THD}}, \tau_2^{\text{THD}}, \ldots\}$. NMIs, IRQs, and threads are subject to the scheduling hierarchy discussed in Section 2.2, i.e., the NMI has always a higher priority than IRQs, and IRQs always have higher priority than threads. Given a thread $\tau_i^{\text{THD}}$, at a given point in time, the set of threads with a higher-priority than $\tau_i^{\text{THD}}$ is denoted by $\Gamma_{\text{HP}_i}^{\text{THD}}$. Similarly, the set of tasks with priority lower than $\tau_i^{\text{THD}}$ is denoted by $\Gamma_{\text{LP}_i}^{\text{THD}}$. Although the schedulers might have

Figure 57 – NMI generator (O1).




threads with the same priority in their queues, only one among them will be selected to have its context load, and consequently, starting to run. Hence, when scheduling, the schedulers elect a single thread as the highest-priority one.

The system model is formalized using the modular approach, where the *generators* model the independent action of tasks and synchronization primitives, and the *specification* models the synchronized behavior of the system. The next sections explains the *generators* as the basic operations of the system, and the specifications as a set of *rules* that explains the system behavior.

### 6.1.1 Basic Operations

This section describes *generators* relevant for the scheduling latency analysis, starting by the interrupt behavior:

- **O1**: The NMI context starts with the entry of the NMI handler (`nmi_entry`), and exits in the return of the handler (`nmi_exit`). This operation is modeled as in Figure 57.
- **O2**: Linux allows threads to temporarily mask interrupts (`local_irq_disable`), in such a way to avoid access to shared data in an inconsistent state. Threads need to unmask interrupts (`local_irq_enable`) at the end of the critical section, as modeled in Figure 58.
- **O3**: To enforce synchronization, the processor masks interrupts before calling an interrupt handler on it. IRQs stays masked during the entire execution of an interrupt handler (`hw_local_irq_disable`). Interrupts are unmasked after the return of the handler (`hw_local_irq_enable`), as shown in Figure 59. In the model, these events are used to identify the begin and the return of an IRQ execution.

The reference model considers two threads: the thread under analysis and an arbitrary other thread (including the idle thread). The corresponding operations are discussed next.

- **O4**: The thread is not running until its context is loaded in the processor (`sched_switch_in`). The context of a thread can be unloaded by a suspension (`sched_switch_suspend`), blocking (`sched_switch_blocking`), or preemption (`sched_switch_preempt`), as in Figure 60.

Figure 58 – IRQ disabled by software (O2).

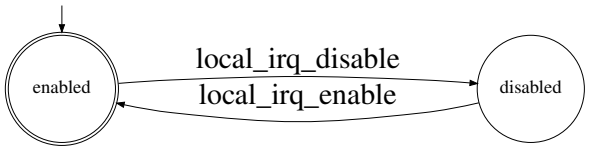



Figure 59 – IRQs disabled by hardware (O3).

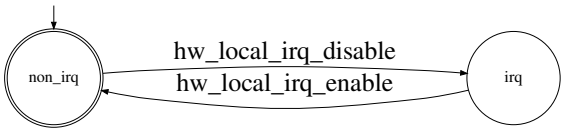



Figure 60 – Context switch generator (04).

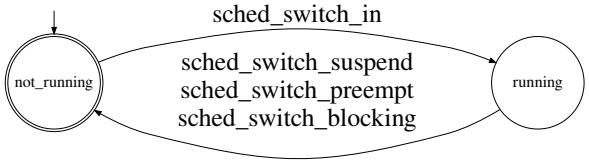



Figure 61 – Context switch generator (05).

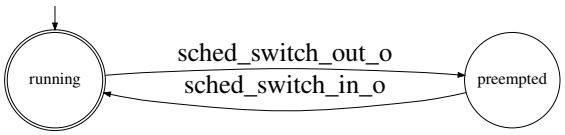



Figure 62 – Preempt disable (06).

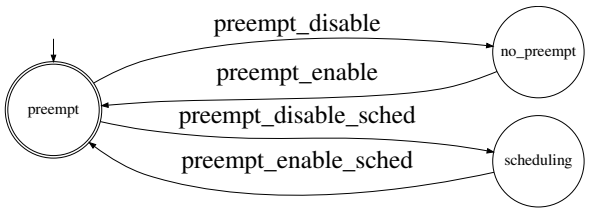



Figure 63 – Scheduling context (07).

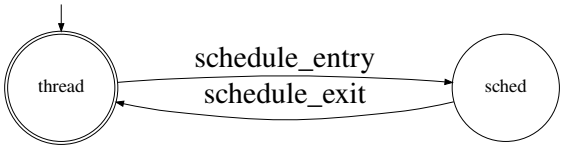



Figure 64 – Thread runnable/sleepable (08).

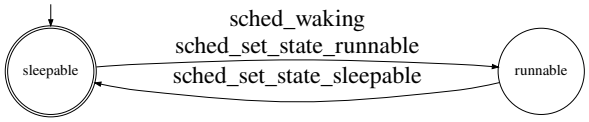
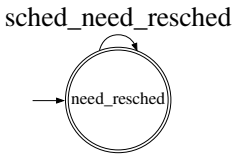



Figure 65 – Need re-schedule operation (09).




- **O5**: The model considers that there is always *another thread* ready to run. The reason is that, on Linux, the *idle state* is implemented as a thread, so at least the *idle thread* is ready to run. The other thread can have its context unloaded (`sched_switch_out_o`) and loaded (`sched_switch_in_o`) in the processor, as modeled in Figure 61.

- **O6**: The preemption is enabled by default. Although the same *function* is used to disable preemption, the model distinguishes the different reasons to disable preemption, as modeled in Figure 62. The preemption can be disabled either to postpone the scheduler execution (`preempt_disable`), or to protect the scheduler execution of a recursive call (`preemp_disable_sched`). Hereafter, the latter mode is referred to as *preemption disabled to call the scheduler* or *preemption disabled to schedule*.

- **O7**: The scheduler starts to run selecting the highest-priority thread (`schedule-`

`_entry`, in Figure 63), and returns after scheduling (`schedule_exit`).

- **O8**: Before being able to run, a thread needs to be awakened (`sched_waking`). A thread can set its state to *sleepable* (`sched_set_state_sleepable`) when in need of resources. This operation can be undone if the thread sets its state to runnable again (`sched_set_state_runnable`). The state-machine that illustrates the interaction among these events is shown in Figure 64.
- **O9**: The need re-schedule (`sched_need_resched`) notifies that the currently running thread is not the highest-priority anymore, and so the current CPU needs to re-schedule, in such way to select the new highest-priority thread (Figure 65).

### 6.1.2 Rules

The *Thread Synchronization Model* (OLIVEIRA; OLIVEIRA; CUCINOTTA, 2020) includes a set of *specifications* defining the synchronization rules among *generators* (i.e., the basic operations discussed in Section 6.1.1). Next, we summarize a subset of rules extracted from the automaton, which are relevant to analyze the scheduling latency. Each rule points to a related specification, graphically illustrated with a corresponding figure.

**IRQ and NMI rules.** First, we start discussing rules related to IRQs and NMI.

- **R1**: There is no specification that blocks the execution of a NMI (**O1**) in the automaton.
- **R2**: There is a set of events that are not allowed in the NMI context (Figure 66), including:
  - **R2a**: set the need resched (**O9**).
  - **R2b**: call the scheduler (**O7**).
  - **R2c**: switch the thread context (**O4** and **O5**)
  - **R2d**: enable the preemption to schedule (**O6**).
- **R3**: There is a set of events that are not allowed in the IRQ context (Figure 67), including:
  - **R3a**: call the scheduler (**O7**).
  - **R3b**: switch the thread context (**O4** and **O5**).
  - **R3c**: enable the preemption to schedule (**O6**).
- **R4**: IRQs are disabled either by threads (**O2**) or IRQs (**O3**), as in the model in Figure 68. Thus, it is possible to conclude that:

Figure 66 – NMI blocks all other operations (R2).

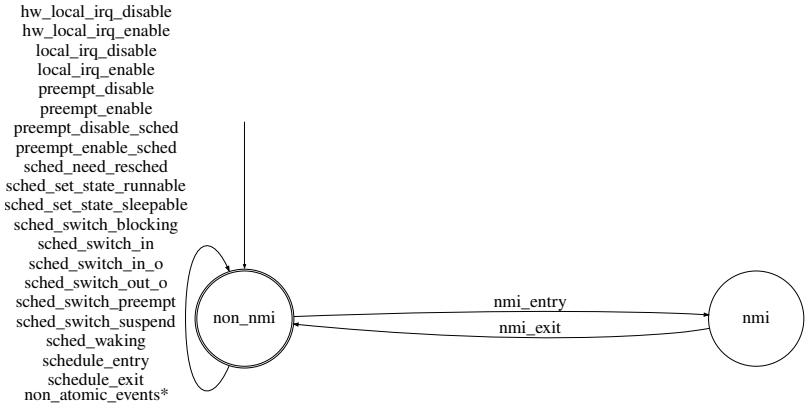



Figure 67 – Operations blocked in the IRQ context (R3).

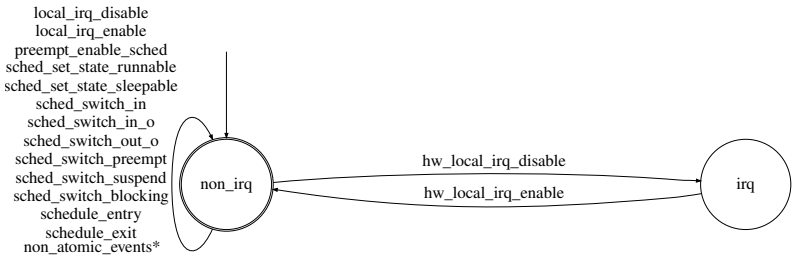



Figure 68 – IRQ disabled by thread or IRQs (R4).

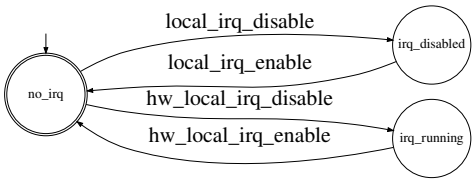



Figure 69 – The scheduler is called with interrupts enabled (R5).

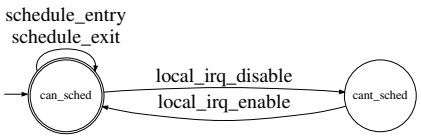
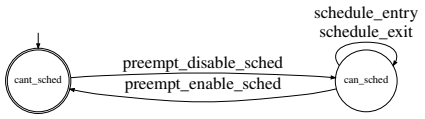



Figure 70 – The scheduler is called with preemption disabled to call the scheduler (R6).




- **R4a**: by disabling IRQs, a thread postpones the beginning of the IRQ handlers.
- **R4b**: when IRQs are not disabled by a thread, IRQs can run.

**Thread context.** Next, synchronization rules related to the thread context are discussed. We start presenting the necessary conditions to call the scheduler (**O7**).

**Necessary conditions to call the scheduler.**

- **R5**: The scheduler is called (and returns) with interrupts enabled (Figure 69).

Figure 71 – The scheduler context does not enable the preemption (R7).

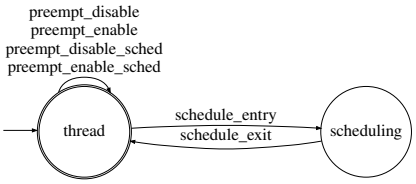



Figure 72 – The context switch occurs with interrupts and preempt disabled (R8).

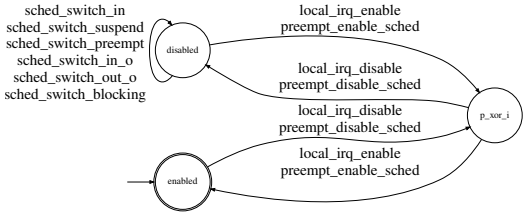



Figure 73 – The context switch occurs in the scheduling context (R9).

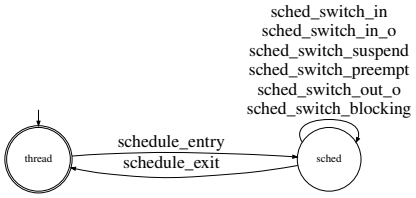



Figure 74 – Wakeup and need resched requires IRQs and preemption disabled (R10 and R11).

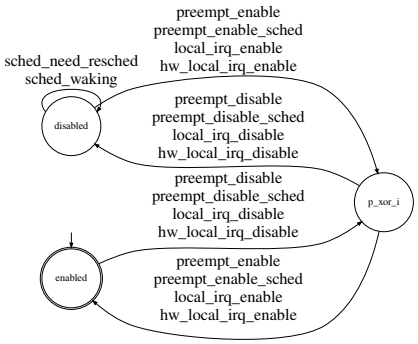



Figure 75 – Disabling preemption to schedule always causes a call to the scheduler (R12).

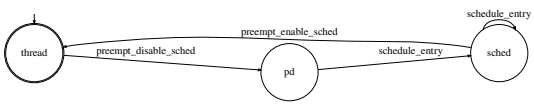
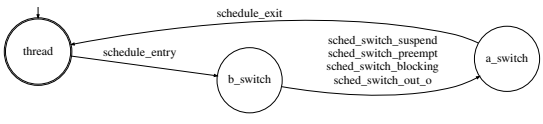



Figure 76 – Scheduling always causes context switch (R13).



- **R6**: The scheduler is called (and returns) with preemption disabled to call the scheduler (Figure 70).

  Rule R7 ensures that the scheduler always run with the preemption disabled.

- **R7**: The preemption is never enabled by the scheduling context (Figure 71).

  Regarding the context switch (**O4** and **O5**), the following conditions are required.

**Necessary conditions for a context switch.**

- **R8**: The context switch occurs with interrupts disabled by threads (**O2**) and preemption disabled to schedule (**O6**, Figure 72).
- **R9**: The context switch occurs in the scheduling context (**O7**, Figure 73).

The necessary conditions to set the need resched (**O9**) and to wakeup a thread (**O8**) are the same. They are listed below, and showed in Figure 74.

**Necessary conditions to set the need resched and to wakeup a thread.**

- **R10**: preemption should be disabled, by any mean (**O6**).
- **R11**: IRQs should be masked, either to avoid IRQ (**O2**) or to postpone IRQs (**O3**).

Until here, we were mentioned necessary conditions. From now on, we will be considering sufficient conditions.

**Sufficient conditions to call the scheduler and to cause a context switch.**

- **R12**: disabling preemption to schedule (**O6**) always causes a call to the scheduler (**O7**, Figure 75).
- **R13**: calling the scheduler (**O7**) always results in a context switch (**O4**,**O5**). Recall that if the system is idle, the idle thread is executed after the context switch (Figure 76).
- **R14**: setting need resched (**O9**) always results in a context switch (**O4**,**O5**, Figure 77).

Figure 77 – Setting need resched always causes a context switch (R14).

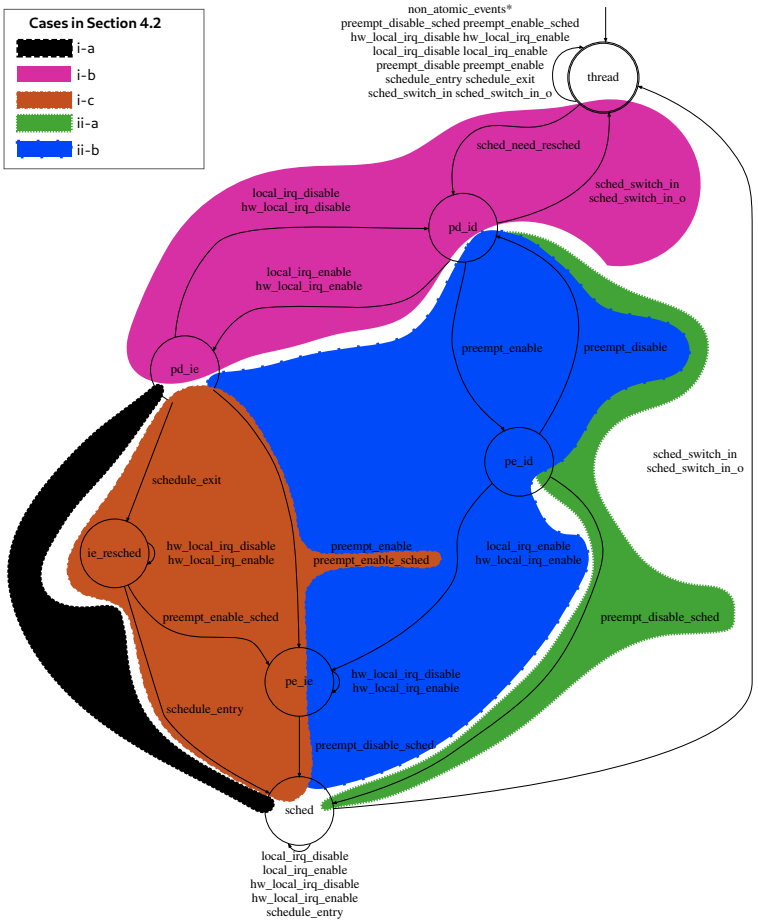
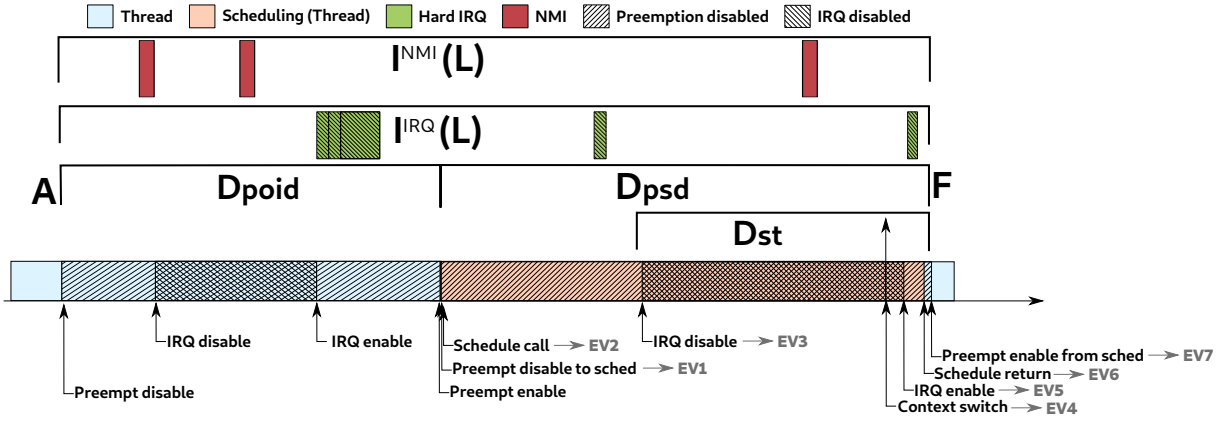



Figure 78 – Reference Timeline.



## 6.2 DEMYSTIFYING THE REAL-TIME LINUX SCHEDULING LATENCY

### 6.2.1 Problem statement

We start defining the scheduling latency (hereafter only latency) and then we leverage the rules presented in Section 6.1 and the related automaton model to derive an upper bound reflecting all the peculiarities of Linux. The *latency* experienced by a *thread instance* (also called *job*) may be informally defined as the maximum time elapsed between the instant in which it becomes ready while having the highest-priority among all ready threads, and the time instant in which it is allowed to execute its own code after the context switch has already been performed. By extension, the latency of a thread is defined as reported in Definition 1.

**Definition 1** (Thread Scheduling Latency)**.** *The scheduling latency experienced by an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ *is the longest time elapsed between the time A in which* any *job of* $\tau_i^{THD}$ *becomes ready and with the highest priority, and the time F in which the scheduler returns and allows* $\tau_i^{THD}$ *to execute its code, in any possible schedule in which* $\tau_i^{THD}$ *is not preempted by any other thread in the interval* [*A*, *F*]*.*

For brevity, we refer next to the event that causes any job of $\tau_i^{\text{THD}}$ becoming ready *and* with the maximum priority as $\texttt{RHP}_i$ event[1]. With Definition 1 in place, this paper aims at computing a theoretically-sound upper bound to the latency experienced by an arbitrary $\tau_i^{\text{THD}} \in \Gamma^{\text{THD}}$ under analysis. To this end, we extract next some formal properties and lemmas from the operations and rules presented in Section 6.1. We begin determining which types of entities may prolong the latency of $\tau_i^{\text{THD}}$.

**Property 1.** *The scheduling latency of an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ cannot *be prolonged due to high-priority interference from other threads* $\tau_j^{THD} \in \Gamma_{HP_i}^{THD}$*.*

*Proof.* By contradiction, assume the property does not hold. Then, due to the priority ordering, it means that either: **(i)** $\tau_i^{\text{THD}}$ was not the highest-priority thread at the beginning of the interval [*A*, *F*] (as defined in Definition 1), or **(ii)** $\tau_i^{\text{THD}}$ has been preempted in [*A*, *F*]. Both cases contradict Definition 1, hence the property follows. □

Differently, Property 2 shows that the latency of a thread may be prolonged due to *priority-inversion blocking* caused by other threads $\tau_j^{\text{THD}} \in \Gamma_{\text{LP}_i}^{\text{THD}}$ with a lower priority.

**Property 2.** *The latency of an arbitrary thread* $\tau_i^{THD} \in \Gamma^{THD}$ can *be prolonged due to low-priority blocking from other threads* $\tau_j^{THD} \in \Gamma_{LP_i}^{THD}$*.*

*Proof.* The property follows by noting that, for example, a low-priority thread may disable the preemption to postpone the scheduler, potentially prolonging the latency of $\tau_i^{\text{THD}}$. □

[1] Note that $\texttt{RHP}_i$ is an event external to *the model*, for instance, it can be a hardware event that dispatches an IRQ, or the event that causes a thread to activate another thread.

With Property 1 and Property 2 in place, we bound the Linux latency as follows, referring to an arbitrary thread $\tau_i^{\mathsf{THD}}$ under analysis. First, as a consequence of Property 1, only the NMI and IRQs may prolong the latency due to high-priority interference, and such an interference is equal for all threads $\tau_i^{\mathsf{THD}} \in \Gamma^{\mathsf{THD}}$ since NMI and IRQs have higher priorities than threads. We model the interference due to the NMI and IRQs in a time window of length $t$ with the functions $I^{\mathsf{NMI}}(t)$ and $I^{\mathsf{IRQ}}(t)$, respectively. We then show next in Section 6.3 how to derive such functions. Besides interference, the latency is caused by constant kernel overheads (e.g., due to the execution of the kernel code for performing the context switch) and priority-inversion blocking (see Property 2), which we bound with a term $L^{\mathsf{IF}}$. In principle, the delays originating $L^{\mathsf{IF}}$ may be different for each thread $\tau_i^{\mathsf{THD}} \in \Gamma^{\mathsf{THD}}$. However, for simplicity, we conservatively bound $L^{\mathsf{IF}}$ in a thread-independent manner as discussed next in Section 6.2.2 and 6.3. The latency of $\tau_i^{\mathsf{THD}}$ is then a function of the above delays, and is bounded by leveraging standard techniques for response-time analysis in real-time systems (AUDSLEY et al., 1993; JOSEPH; PANDYA, 1986; LEHOCZKY; SHA; DING, 1989), i.e., by the least positive value fulfilling the following equation:

$$L = L^{\mathsf{IF}} + I^{\mathsf{NMI}}(L) + I^{\mathsf{IRQ}}(L). \tag{18}$$

Next, we show how to bound $L^{\mathsf{IF}}$.

### 6.2.2 Bounding $L^{\mathsf{IF}}$

**Analysis Approach.** As discussed in Section 6.1, after the $\texttt{RHP}_i$ event occurs (i.e., when $\tau_i^{\mathsf{THD}}$ becomes the ready thread with the highest priority), the kernel identifies the need to schedule a new thread when the `set_need_resched` event takes place. Then, an ordered sequence of events occurs. Such events are motivated by the operations and rules discussed in Section 6.1, graphically illustrated in the lower part of Figure 78, and discussed below.

**EV1** The necessary conditions to call the scheduler need to be fulfilled: IRQs are enabled, and preemption is disabled to call the scheduler. It follows from rule R5 and R6;

**EV2** The scheduler is called. It follows from R12;

**EV3** In the scheduler code, IRQs are disabled to perform a context switch. It follows from rule R8;

**EV4** The context switch occurs. It follows from rule R13 and R14;

**EV5** Interrupts are enabled by the scheduler. It follows from R5;

**EV6** The scheduler returns;

**EV7** The preemption is enabled, returning the thread its own execution flow.

Note that, depending on what the processor is executing when the $\texttt{RHP}_i$ event occurs, not all the events may be involved in (and hence prolong) the scheduling latency. Figure 77 illustrates all the allowed sequences of events from the occurrence of the `set_need_resched` event (caused by $\texttt{RHP}_i$) until the context switch (EV4), allowing the occurrence of the other events (EV5-EV7). According to the automaton model, there are five possible and mutually-exclusive cases, highlighted with different colors in Figure 77. Our strategy for bounding $L^{\mathsf{IF}}$ consists in deriving an individual bound for each of the five cases, taking the maximum as a safe bound. To derive the five cases, we first distinguish between: **(i)** if $\texttt{RHP}_i$ occurs when the current thread $\tau_j^{\mathsf{THD}} \in \Gamma_{\mathsf{LP}_i}^{\mathsf{THD}}$ is in the scheduler execution flow, both voluntarily, or involuntarily as a consequence of a previous `set_need_resched` occurrence, after disabling the preemption to call the scheduler and, **(ii)** otherwise.

We can distinguish three mutually-exclusive sub-cases of (i):

**i-a** if $\texttt{RHP}_i$ occurs between events EV1 and EV2, i.e., after that preemption has been disabled to call the scheduler and before the actual scheduler call (black in Figure 77);

**i-b** if $\texttt{RHP}_i$ occurs in the scheduler between EV2 and EV3, i.e., after that the scheduler has already been called and before interrupts have been disabled to cause the context switch (pink in Figure 77);

**i-c** if $\texttt{RHP}_i$ occurs in the scheduler between EV3 and EV7, i.e., after interrupts have already been masked in the scheduler code and when the scheduler returns (brown in Figure 77);

In case (ii), $\texttt{RHP}_i$ occurred when the current thread $\tau_j^{\mathsf{THD}} \in \Gamma_{\mathsf{LP}_i}^{\mathsf{THD}}$ is not in the scheduler execution flow. Based on the automaton of Figure 77, two sub-cases are further differentiated:

**ii-a** when $\texttt{RHP}_i$ is caused by an IRQ, and the currently executing thread may delay $\texttt{RHP}_i$ only by disabling interruptions (green in Figure 77).

**ii-b** otherwise (blue in Figure 77).

**Variables Selection.** One of the most important design choices for the analysis consists in determining the most suitable variables to be used for deriving the analytical bound. Since the very early stages of its development, the PREEMPT_RT Linux had as a target to minimize the code portions executed in interrupt context and the code sections in which the preemption is disabled. One of the advantages of this design choice consists indeed in the reduction of scheduling delays. Nevertheless, disabling

Table 9 – Parameters used to bound $L^{\mathsf{IF}}$.

| Param. | Length of the longest interval |
|---|---|
| $D_{\mathsf{PSD}}$ | in which preemptions are disabled to schedule. |
| $D_{\mathsf{PAIE}}$ | in which the system is in state `pe_ie` of Figure 77. |
| $D_{\mathsf{POID}}$ | in which the preemption is disabled to postpone the scheduler or IRQs are disabled. |
| $D_{\mathsf{ST}}$ | between two consecutive occurrences of EV3 and EV7. |

the preemption or IRQs is sometimes merely mandatory in the kernel code. As pointed out in Property 2, threads may also disable the preemption or IRQs, e.g., to enforce synchronization, thus impacting on the scheduling latency. Building upon the design principles of the fully-preemptive PREEMPT_RT kernel, Table 9 presents and discusses the set of variables selected to bound the latency, which are more extensively discussed next in Sections 6.3, and graphically illustrated in Figure 78. Such variables considers the longest intervals of time in which the preemption and/or IRQs are disabled, taking into consideration the different disabling modes discussed in Section 6.1.

**Deriving the bound.** Before discussing the details of the five cases, we present a bound on the interference-free duration of the scheduler code in Lemma 1.

**Lemma 1.** *The interference-free duration of the scheduler code is bounded by $D_{PSD}$.*

*Proof.* It follows by noting that by rule R6 the scheduler is called and returns with the preemption disabled to call the scheduler and, by rules R2d, R3c, and R7, the preemption is not enabled again until the scheduler returns. □

Next, we provide a bound to $L^{\mathsf{IF}}$ in each of the five possible chains of events.

**Case (i).** In case (i), the preemption is already disabled to call the scheduler, hence either `set_need_resched` has already been triggered by another thread $\tau_j^{\mathsf{THD}} \neq \tau_i^{\mathsf{THD}}$ or the current thread voluntarily called the scheduler. Then, due to rules R13 and R14, a context switch will occur. Consequently, the processor continues executing the scheduler code. Due to rule R5, the scheduler is called with interrupts enabled and preemption disabled, hence $\mathtt{RHP}_i$ (and consequently `set_need_resched`) must occur because of an event triggered by an interrupt. By rule R2, NMI cannot cause `set_need_resched`; consequently, it must be caused by an IRQ or the scheduler code itself. Due to EV3, IRQs are masked in the scheduler code before performing the context switch. We recall that case (i) divides into three possible sub-cases, depending on whether $\mathtt{RHP}_i$ occurs between EV1 and EV2 (case i-a), EV2 and EV3 (case i-b), or EV3 and EV7 (case i-c). Lemma 2 bounds $L^{\mathsf{IF}}$ for cases (i-a) and (i-b).

**Lemma 2.** *In cases (i-a) and (i-b), it holds*

$$L^{IF}_{(i-a)} \leq D_{PSD},\ L^{IF}_{(i-b)} \leq D_{PSD}. \tag{19}$$

*Proof.* In both cases it holds that preemption is disabled to call the scheduler and IRQs have not been disabled yet (to perform the context switch) when $\texttt{RHP}_i$ occurs. Due to rules R2 and R5, $\texttt{RHP}_i$ may only be triggered by an IRQ or the scheduler code itself. Hence, when $\texttt{RHP}_i$ occurs `set_need_resched` is triggered and the scheduler performs the context switch for $\tau_i^{\text{THD}}$. Furthermore, in case (i-b) the processor already started executing the scheduler code when $\texttt{RHP}_i$ occurs. It follows that $L^{\text{IF}}$ is bounded by the interference-free duration of the scheduler code. By Lemma 1, such a duration is bounded by $D_{\text{PSD}}$. In case (i-a), the scheduler has not been called yet, but preemptions have already been disabled to schedule. By rule R12, it will immediately cause a call to the scheduler, and the preemption is not enabled again between EV1 and EV2 (rules R2d, R3c, and R7). Therefore, also for case (i-a) $L^{\text{IF}}$ is bounded by $D_{\text{PSD}}$, thus proving the lemma. □

Differently, case (i-c), in which $\texttt{RHP}_i$ occurs between EV3 and EV7, i.e., after interrupts are disabled to perform the context switch, is discussed in Lemma 3.

**Lemma 3.** *In case (i-c), it holds*

$$L^{IF}_{(i-c)} \leq D_{ST} + D_{PAIE} + D_{PSD}. \tag{20}$$

*Proof.* In case (i), the scheduler is already executing to perform the context switch of a thread $\tau_j^{\text{THD}} \neq \tau_i^{\text{THD}}$. Due to rules R2 and R5, $\texttt{RHP}_i$ may only be triggered by an IRQ or the scheduler code itself. If the scheduler code itself caused $\texttt{RHP}_i$ before the context switch (i.e., between EV3 and EV4), the same scenario discussed for case (i-b) occurs, and the bound of Equation 19 holds. Then, case (i-c) occurs for $\texttt{RHP}_i$ arriving between EV4 and EV7 for the scheduler code, or EV3 and EV7 for IRQs. IRQs may be either disabled to perform the context switch (if $\texttt{RHP}_i$ occurs between EV3 and EV5), or already re-enabled because the context switch already took place (if $\texttt{RHP}_i$ occurs between EV5 and EV7). In both cases, thread $\tau_i^{\text{THD}}$ needs to wait for the scheduler code to complete the context switch for $\tau_j^{\text{THD}}$. If $\texttt{RHP}_i$ occurred while IRQs were disabled (i.e., between EV3 and EV5), the IRQ causing $\texttt{RHP}_i$ is executed, triggering `set_need_resched`, when IRQs are enabled again just before the scheduler returns (see rule R5).

Hence, due to rule R14, the scheduler needs to execute again to perform a second context switch to let $\tau_i^{\text{THD}}$ execute. As shown in the automaton of Figure 77, there may exist a possible system state in case (i-c) (the brown one in Figure 77) in which, after $\texttt{RHP}_i$ occurred and before the scheduler code is called again, both the preemption and IRQs are enabled before calling the scheduler (state `pe_ie` in Figure 77). This system state is visited when the kernel is executing the non-atomic function to enable preemption, because the previous scheduler call (i.e., the one that caused the context switch for $\tau_j^{\text{THD}}$) enabled IRQs before returning (EV5). Consequently, we can bound $L^{\text{IF}}$ in case (i-c) by bounding the interference-free durations of the three intervals: $I_{\text{ST}}$,

which lasts from EV3 to EV7, $I_{\mathsf{PAIE}}$, which accounts for the kernel being in the state `pe_ie` of Figure 77 while executing EV7, and $I_{\mathsf{S}}$, where preemption is disabled to call the scheduler and the scheduler is called again to schedule $\tau_i^{\mathsf{THD}}$ (from EV1 to EV7). By definition and due to Lemma 1 and rules R2d, R3c, R7, and R12, $I_{\mathsf{ST}}$, $I_{\mathsf{PAIE}}$, and $I_{\mathsf{S}}$ cannot be longer than $D_{\mathsf{ST}}$, $D_{\mathsf{PAIE}}$, and $D_{\mathsf{PSD}}$, respectively. The lemma follows by noting that the overall duration of $L^{\mathsf{IF}}$ is bounded by the sum of the individual bounds on $I_{\mathsf{ST}}$, $I_{\mathsf{PAIE}}$, and $I_{\mathsf{S}}$. □

**Case (ii).** In case (ii), $\mathtt{RHP}_i$ occurs when the current thread $\tau_j^{\mathsf{THD}} \in \Gamma_{\mathsf{LP}_i}^{\mathsf{THD}}$ is not in the scheduler execution flow. As a consequence of the $\mathtt{RHP}_i$ events, `set_need_resched` is triggered. By rule R14, triggering `set_need_resched` always result in a context switch and, since $\mathtt{RHP}_i$ occurred outside the scheduler code, the scheduler needs to be called to perform the context switch (rule R9). Hence, we can bound $L^{\mathsf{IF}}$ in case (ii) by individually bounding two time intervals $I_{\mathsf{S}}$ and $I_{\mathsf{SO}}$ in which the processor is *executing* or *not executing* the scheduler execution flow (from EV1 to EV7), respectively. As already discussed, the duration of $I_{\mathsf{S}}$ is bounded by $D_{\mathsf{PSD}}$ (Lemma 1). To bound $I_{\mathsf{SO}}$, we need to consider individually cases (ii-a) and (ii-b). Lemma 4 and Lemma 5 bound $L^{\mathsf{IF}}$ for cases (ii-a) and (ii-b), respectively.

**Lemma 4.** *In case (ii-a), it holds*

$$L_{(ii-a)}^{IF} \leq D_{POID} + D_{PSD}. \tag{21}$$

*Proof.* In case (ii-a) $\mathtt{RHP}_i$ occurs due to an IRQ. Recall from Operation O3 that when an IRQ is executing, it masks interruptions. Hence, the IRQ causing $\mathtt{RHP}_i$ can be delayed by the current thread or a lower-priority IRQ that disabled IRQs. When $\mathtt{RHP}_i$ occurs, the IRQ triggering the event disables the preemption (IRQs are already masked) to fulfill R10 and R11, and triggers `set_need_resched`. If preemption was enabled before executing the IRQ handler and if `set_need_resched` was triggered, when the IRQ returns, it first disables preemptions (to call the scheduler, i.e., `preempt_disable_sched`). It then unmasks interrupts (this is a safety measure to avoid stack overflows due to multiple scheduler calls in the IRQ stack). This is done to fulfill the necessary conditions to call the scheduler discussed in rules R5 and R6. Due to rules R3a and R12, the scheduler is called once the IRQ returns. Hence, it follows that in the whole interval $I_{\mathsf{SO}}$, either the preemption or interrupts are disabled. Then it follows that $I_{\mathsf{SO}}$ is bounded by $D_{\mathsf{POID}}$, i.e., by the length of the longest interval in which either the preemption or IRQs are disabled. The lemma follows recalling that the duration of $I_{\mathsf{S}}$ is bounded by $D_{\mathsf{PSD}}$. □

**Lemma 5.** *In case (ii-b), it holds*

$$L_{(ii-b)}^{IF} \leq D_{POID} + D_{PAIE} + D_{PSD}, \tag{22}$$

*Proof.* In case (ii-b) the currently executing thread delayed the scheduler call by disabling the preemption or IRQs. The two cases in which the $\texttt{RHP}_i$ event is triggered either by a thread or an IRQ are discussed below.

*(1) $\texttt{RHP}_i$ is triggered by an IRQ.* Consider first that $\texttt{RHP}_i$ is triggered by an IRQ. Then, the IRQ may be postponed by a thread or a low-priority IRQ that disabled interrupts. When the IRQ is executed, it triggers `set_need_resched`. When returning, the IRQ returns to the previous preemption state[2], i.e, if it was disabled before the execution of the IRQ handler, preemption is disabled, otherwise it is enabled. If the preemption was enabled before executing the IRQ, the same scenario discussed for case (ii-a) occurs, and the bound of Equation 21 holds. Otherwise, if the preemption was disabled to postpone the scheduler execution, the scheduler is delayed due to priority-inversion blocking. Then it follows that when delaying the scheduler execution, either the preemption or IRQs are disabled. When preemption is re-enabled by threads and interrupts are enabled, the preemption needs to be disabled again (this time not to postpone the scheduler execution, but to call the scheduler) to fulfill the necessary conditions listed in rules R5 and R6, hence necessarily traversing the `pe_ie` state (shown in Figure 77), where both preemptions and interrupts are enabled. Hence, it follows that $I_{\text{SO}}$ is bounded by $D_{\text{POID}} + D_{\text{PAIE}}$ if $\texttt{RHP}_i$ is triggered by an IRQ.

*(2) $\texttt{RHP}_i$ is triggered by a thread.* In this case, a thread causes `set_need_resched`. Since the `set_need_resched` event requires IRQs and preemption disabled, the scheduler execution is postponed until IRQs and preemption are enabled (`pe_ie` state). Once both are enabled, the preemption is disabled to call the scheduler. Then it follows that $I_{\text{SO}}$ is bounded by $D_{\text{POID}} + D_{\text{PAIE}}$ if $\texttt{RHP}_i$ is triggered by a thread. Then it follows that $I_{\text{SO}}$ is bounded by $D_{\text{POID}} + D_{\text{PAIE}}$ in case (ii-b). The lemma follows recalling that $I_S$ is bounded by $D_{\text{PSD}}$. □

By leveraging the individual bounds on $L^{\text{IF}}$ in the five cases discussed above, Lemma 6 provides an overall bound that is valid for all the possible events sequences.

**Lemma 6.**

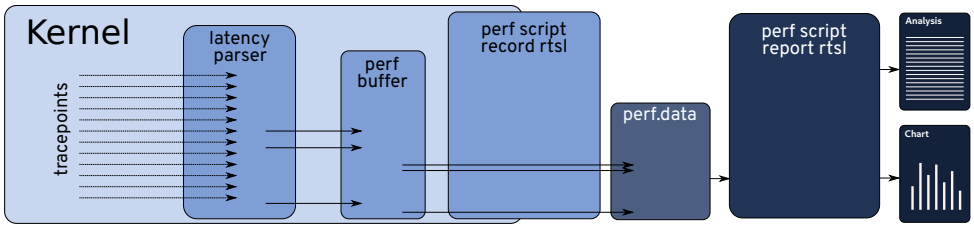
$$L^{IF} \leq max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD}, \tag{23}$$

*Proof.* The lemma follows by noting that cases (i-a), (i-b), (i-c), (ii-a), (ii-b) are mutually-exclusive and cover all the possible sequences of events from the occurrence of $\texttt{RHP}_i$ and `set_need_resched`, to the time instant in which $\tau_i^{\text{THD}}$ is allowed to execute (as required by Definition 1), and the right-hand side of Equation 23 simultaneously upper bounds the right-hand sides of Equations 19, 20, 21, and 22. □

Theorem 1 summarizes the results derived in this section.

[2] Note that, internally to the IRQ handler, the preemption state may be changed, e.g., to trigger `set_need_resched`.

Figure 79 – `rt_sched_latency`: tool kit components.




**Theorem 1.** *The scheduling latency experienced by an arbitrary thread $\tau_i^{THD}$ is bounded by the least positive value that fulfills the following recursive equation:*

$$L = max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{NMI}(L) + I^{IRQ}(L) \quad (24)$$

*Proof.* The theorem follows directly from Lemmas 6 and Equation 18. □

## 6.3 RT_SCHED_LATENCY: EFFICIENT SCHEDULING LATENCY ESTIMATION TOOL KIT

The validation tool used in the development of the *Thread Synchronization Model* (OLIVEIRA; OLIVEIRA; CUCINOTTA, 2020) exports all the kernel events to the user-space using `perf`, for later analysis. Although useful for the model validation purpose, the low granularity nature of the synchronization primitives generates a prohibitive amount of information for a performance measurement tool. For instance, one second of trace could generate more than 800 MB of data per CPU. Doing the whole trace analysis in-kernel has shown to be very efficient, as presented in Chapter 5. The problem for such an approach lies in the amount of information that can be stored in kernel memory. While only the worst observed value for some variables, such as $D_{\text{POID}}$, are used in the analysis, the IRQ and NMI analysis required the recording of all interrupts occurrence during the measurements. So the experimental tool kit developed in this work, called `rt_sched_latency`, has a hybrid approach: it uses an in-kernel event parsing and an extension to the `perf script` tool for a post-processing phase. Figure 79 describes the interaction of the tools in the tool kit. The tool kit comprises the `latency parser` and the `perf script` extension, named `rtsl`.

The `latency parser` uses the kernel `tracepoints` from the *Thread Synchronization Model* to observe their occurrence from inside the kernel. The `latency parser` registers a *callback function* to the kernel `tracepoints`. When a `tracepoint` from the model is hit, rather than writing the trace to the trace buffer (a buffer maintained by the `perf` tool to store trace data) the respective *function* is called. The *callback functions* are used to pre-process the events, transforming them into relevant information. For example, *nmi_entry* event records the arrival time (all the values are *observed values*, but the *observed* qualifiers are omitted for simplicity) without printing the occurrence of the event. When the *nmi_exit* occurs, it computes the execution time of the NMI,

and prints the arrival time and the execution time of the NMI. A similar behavior is implemented for other metrics, for instance for the IRQ occurence. The difference is that the interference must be removed from other metrics. For example, if an NMI and an IRQ occur while measuring a candidate $D_{\text{POID}}$, the IRQ and the NMI execution time are discounted from the measured value.

The `latency parser` communicates with `perf` using a new set of `tracepoints`, and these are printed to the trace buffer. The following events are generated by the `latency parser`:

- **irq_execution**: prints the IRQ identifier, starting time, and execution time;
- **nmi_execution**: prints the starting time, and execution time;
- **max_poid**: prints the new maximum observed $D_{\text{POID}}$ duration;
- **max_psd**: prints the new maximum observed $D_{\text{PSD}}$ duration;
- **max_dst**: prints the new maximum observed $D_{\text{ST}}$ duration;
- **max_paie**: prints the new maximum observed $D_{\text{PAIE}}$ duration;

By only tracing the return of interrupts and the new maximum values for the thread metrics, the amount of data generated is reduced to the order of 200KB of data per second per CPU. Hence, reducing the overhead of saving data to the trace buffer, while enabling the measurements to run for hours by saving the results to the disk. The data collection is done by the `perf rtsl script`. It initiates the `latency parser` and start recording its events, saving the results to the `perf.data` file. The command also accepts a workload as an argument. For example, the following command line will start the data collection while running `cyclictest` concurrently:

```
perf script record rtsl cyclictest -smp -p95 -m -q
```

Indeed, this is how the data collection is made for Section 6.4. The trace analysis is done with the following command line: `perf script report rtsl`. The perf script will read the `perf.data` and perform the analysis. A `cyclictest.txt` file with `cyclictest` output is also read by the script, adding its results to the analysis as well. The script to run the analysis is implemented in `python`, which facilitates the handling of data, needed mainly for the IRQ and NMI analysis.

**IRQ and NMI analysis** While the variables used in the analysis are clearly defined (Table 9), the characterization of IRQs and NMI interference is delegated to functions (i.e., $I^{\text{NMI}}(L)$ and $I^{\text{IRQ}}(L)$), for which different characterizations are proposed next. The reason being is that there is no consensus on what could be the single best characterization of interrupt interference. For example, in a discussion among the Linux kernel

developers, it is a common opinion that the classical sporadic model would be too pessimistic (OLIVEIRA, 2019d). Therefore, this work assumes that there is no single way to characterize IRQs and NMIs, opting to explore different IRQs and NMI characterizations in the analysis. Also, the choice to analyze the data in user-space using `python` scripts were made to facilitate the extension of the analysis by other users or researchers. The tool presents the latency analysis assuming the following interrupts characterization:

- **No Interrupts**: the interference-free latency ($L^{\text{IF}}$);
- **Worst single interrupt**: a single IRQ (the worst over all) and a single NMI occurrence;
- **Single (worst) of each interrupt**: a single (the worst) occurrence of each interrupt;
- **Sporadic**: sporadic model, using the observed minimum inter-arrival time and WCET;
- **Sliding window**: using the worst-observed arrival pattern of each interrupt and the observed execution time of individual instances;
- **Sliding window with oWCET**: using the worst-observed arrival pattern of each interrupt and the observed worst-case execution time among all the instances (oWCET).

These different characterization lead to different implementations of $I^{\text{NMI}}(L)$ and $I^{\text{IRQ}}(L)$.

**perf rtsl output.** The `perf rtsl` tool has two outputs: the textual and the graphical one. The textual output prints a detailed description of the latency analysis, including the values for the variables defined in Section 6.2. By doing so, it becomes clear what are the contributions of each variable to the resulting scheduling latency. An excerpt from the output is shown in Figure 80. The tool also creates charts displaying the latency results for each interrupt characterization, as shown in the experiments in Section 6.4.

When the dominant factor of latency is an IRQ or NMI, the textual output already serves to isolate the context in which the problem happens. However, when the dominant factor arises from a thread, the textual output points only to the variable that dominates the latency. Then, to assist in the search for the code section, the `tracepoints` that prints each occurrence of the variables from `latency parser` can be used. These events are not used during the measurements because they occur too frequently, but they can be used in the debug stage. For example, Figure 81 shows the example of the `poid tracepoint` traced using `perf`, capturing the stack trace of

Figure 80 – `perf rtsl` output: excerpt from the textual output (time in nanoseconds).

```
Interference Free Latency:
    paie is lower than 1 us -> neglectable
    latency = max(poid,    dst) + paie +   psd
      42212 = max(22510, 19312) +    0 + 19702
Cyclictest:
    Latency =     27000 with Cyclictest
No Interrupts:
    Latency =     42212 with No Interrupts
Sporadic:
    INT:     oWCET     oMIAT
    NMI:         0         0
     33:     16914    257130
     35:     12913      1843 <- oWCET > oMIAT
    236:     20728      1558 <- oWCET > oMIAT
    246:      3299   1910321
    Did not converge.
```

```
continuing....
Sliding window:
    Window: 42212
            NMI:         0
             33:     16914
             35:     14588
            236:     20728
            246:      3299
    Window: 97741
            236:     21029 <- new!
    Window: 98042
    Converged!
    Latency =     98042 with Sliding Window
```

Figure 81 – Using `perf` and the *latency parser* to find the cause of a large $D_{\text{POID}}$ value.

```
# perf record -a -g -e rtsl:poid --filter "value > 60000"
# perf script
     php 25708 [001] 754905.013632: rtsl:poid: 68391
         ffffffff921cbb6d trace_preempt_on+0x13d ([kernel.kallsyms])
         ffffffff921039ca preempt_count_sub+0x9a ([kernel.kallsyms])
         ffffffff929a507a _raw_spin_unlock_irqrestore+0x2a ([kernel.kallsyms])
         ffffffff92109a55 wake_up_new_task+0x1c5 ([kernel.kallsyms])
         ffffffff920d4c5e _do_fork+0x14e ([kernel.kallsyms])
         ffffffff92004552 do_syscall_64+0x72 ([kernel.kallsyms])
         ffffffff92a00091 entry_SYSCALL_64_after_hwframe+0x49 ([kernel.kallsyms])
             7f2d61d7a685 __libc_fork+0xc5 (/usr/lib64/libc-2.26.so)
             55d87cba3b15 [unknown] (/usr/bin/php)
```

the occurrence of a $D_{\text{POID}}$ value higher than 60 microseconds[3]. In this example, it is possible to see that the spike occurs in the `php` thread while waking up a process during a `fork` operation. This trace is precious evidence, mainly because it is already isolated from other variables, such as the IRQs, that could point to the wrong direction.

## 6.4 EXPERIMENTAL ANALYSIS

This section presents latency measurements, comparing the results found by `cyclictest` and `perf rtsl` while running concurrently in the same system. The main objective of this experimental study is to corroborate the practical applicability of the analysis tool. To this end, we show that the proposed approach provides latency bounds respecting the under millisecond requirement in scheduling precision (which is typical of applications using PREEMPT_RT) for most of the proposed interrupt characterizations. The proposed `perf rtsl` tool individually characterizes the various sources of latency and composes them leveraging a theory-based approach allowing to find highly latency-intensive schedules in a much shorter time than `cyclictest`. The experiment was made in a workstation with one Intel *i7-6700K CPU @ 4.00GHz* processor, with eight cores,

[3] The *latency parser* `tracepoints` are also available via `ftrace`.

and in a server with two Non-Uniform Memory Access (NUMA) Intel *Xeon L5640 CPU @ 2.27GHz* processors with six cores each. Both systems run the Fedora 31 Linux distribution, using the kernel-rt *5.2.21-rt14*. The systems were tuned according to the best practices of real-time Linux systems (KLECH et al., 2020).

The first experiment runs on the workstation three different workloads for 30 minutes. In the first case, the system is mostly *idle*. Then workloads were generated using two `phoronix-test-suite` (`pts`) tests: the `openssl` stress test, which is a *CPU intensive* workload, and the `fio`, `stress-ng` and `build-linux-kernel` tests together, causing a mixed range of *I/O intensive* workload (PHORONIX, 2020). Different columns are reported in each graph, corresponding to the different characterization of interrupts discussed in Section 6.3. The result of this experiment is shown in Figure 82: 1.a, 1.b and 1.c, respectively. In the second experiment, the *I/O intensive* workload was executed again, with different test durations, as described in 2.a, 2.b, and 2.c. The results from `cyclictest` did not change substantially as the time and workload changes. On the other hand, the proposed approach results change, increasing the hypothetical bounds as the kernel load and experiment duration increase. Consistently with `cyclictest` results, the *No Interrupts* column also do not vary substantially. The difference comes from the interrupt workload: the more overloaded the system is, and the longer the tests run, the more interrupts are generated and observed, influencing the results. In all the cases, the *sporadic task model* appears to be overly pessimistic for IRQs: regularly, the *oWCET* of IRQs were longer than the minimal observed inter-arrival time of them. The *Sliding Window with oWCET* also stand out the other results. The results are truncated in the charts 2.b and 2.c: their values are 467 and 801 microseconds, respectively.

Although the reference automata model was developed considering single-core systems, the same synchronization rules are replicated in the multiple-core (*mc*) configuration, considering the local scheduling latency of each CPU. The difference between single and multiple-core cases resides in the inter-core synchronization using, for example, `spinlocks`. However, such synchronization requires preemption and IRQs to be disabled, hence, taking place inside the already defined variables. Moreover, when `cyclictest` runs in the `-smp` mode, it creates a thread per-core, aiming to measure the local scheduling latency. In a *mc* setup, the workload experiment was replicated in the workstation. Furthermore, the *I/O intensive* experiment was replicated in the server. The results of these experiments are shown in Figure 83. In these cases, the effects of the high kernel activation on I/O operations becomes evident in the workstation experiment (3.c) and in the server experiment(4.a). Again the *Sliding Window with oWCET* also stand out the other results, crossing the milliseconds barrier. The source of the higher values in the thread variables (Table 9) is due to cross-core synchronization using `spinlocks`. Indeed, the trace in Figure 81 was observed in the server running the *I/O* workload. The `php` process in that case was part of the `phoronix-test-suit` used

Figure 82 – Workstation experiments: single-core system.

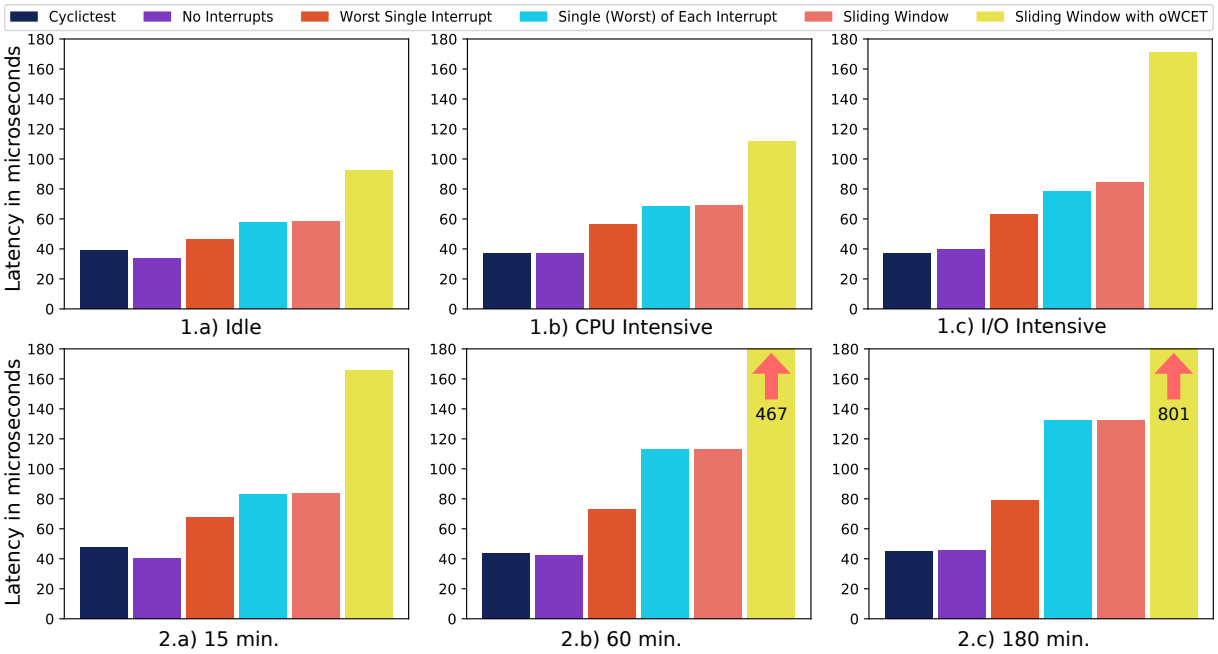
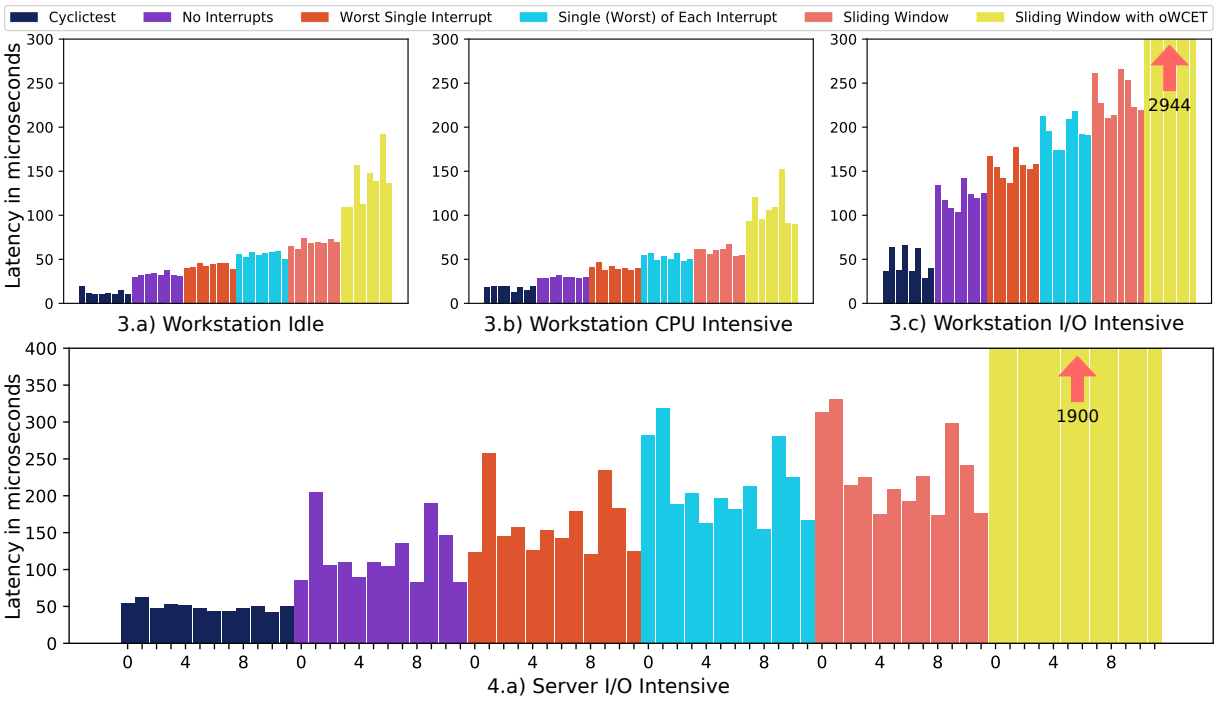



Figure 83 – Workstation and Server experiments: multicore systems.




to generate the workload.

Finally, by running `cyclictest` with and without using the `perf rtsl` tool, it was possible to observe that the trace impact in the minimum, average and maximum values are in the range from one to four microseconds, which is an acceptable range, given the frequency in which events occurs, and the advantages of the approach.

## 6.5 FINAL REMARKS

The work done by developers, reworking the Linux kernel to reduce the code sections that could delay the scheduling of the highest-priority thread, was done empirically, aided by `cyclictest`. However, the outcomes of such effort were known, and developers implicitly knew the model. The problem was that it is not trivial for the kernel developers to express their knowledge using the formalism used in the real-time theory. The challenge was then to connect these two non-trivial areas of knowledge, one expressed in `C` code, and another expressed in mathematical formalism, explicitly (EPSTEIN, 2008).

The usage of the model was an essential intermediary step between the real-time theory and Linux, facilitating the information exchange among the related, but intricate domains. The analysis, built upon a set of practically-relevant variables, ends up concluding what is informally know: the preemption and IRQ disabled sections, along with interrupts, are the evil for the scheduling latency. The real benefits come from the decomposition of the variables, and the efficient method for observing the values. Now users and developers have precise information regarding the sources of the latency on their systems, facilitating the tune their systems, and the definition of where to improve Linux's code, respectively. The improvement of the tool and its integration with the Linux kernel and perf codebase is the practical continuation of this work. But also researchers can take benefits from the approach in the creation of novel real-time related algorithms for Linux, taking into consideration realistic considerations regarding the non-negligible synchronizations of Linux and their associated delays.

# 7 FINAL REMARKS

Linux is complex, and it is fair to say that nobody knows all the details of its implementation. In practice, the analysis and development of the PREEMPT_RT do not require the knowledge of all the details of Linux, but the sole understanding of some execution context and synchronization mechanism, and this fact inspired the development of this thesis. The work presented in (OLIVEIRA; OLIVEIRA, 2016) is the first step in this regard, and it shows that, empirically, the kernel developers can make parallels with the terms used in the academy, and vice-versa. The problem was that empirical evidence is not enough when dealing with safety-critical systems, and the real-time systems theory is highly motivated for such a class of systems.

The development of an abstract model using formal methods was a natural answer to unveil the complexity of Linux in a deterministic way. However, given the known limitations of the application for formal methods on complex systems, the selection of the formalism and methodology was not straightforward.

The real-time characteristic of the system would naturally point to timed formalism, such as timed automata, and the runtime nature of the system led to simulation prone formats such as Petri nets. Still, they did not seem to match with the daily routine of a kernel developer. The option for the simple automata format for the model was taken because, for a kernel developer, the occurrence of the events, as read in the trace, was enough to the understanding of the system. Limiting the model for the goal of explaining the system dynamics, instead of simulating the system execution with more complex formalism, certainly contributed to avoiding hitting the limits imposed by the current state-of-art of formal methods.

The modular approach was another crucial factor in the choice. The guarantee of not forgetting any transition by synchronizing the generators and creating small and specialized specifications enabled an easy and secure way to specify the system. The feedback from the developers that the specification in the formal format was of straightforward understanding added a level of confidence that the modeling process was taking the right direction[1]. Finally, it is worth mentioning that the Supremica IDE played a central role in the modeling phase, mainly by automatizing the automata operations and the non-functional verification of the model. It also allows the simulation of the model, which was fundamental during the debugging of problems.

The simplicity of the automata format and the flexibility of the modular approach was the perfect match for connecting these three complex areas: the Linux kernel, real-time systems theory, and formal methods. This conclusion is confirmed with the usage of automata models (both the complete model, or the specifications) in the verification of the *logical* behavior of the system, and in the analysis of the *timing* behavior of the

[1] See Section 7.4.

systems.

Regarding the *logical* behavior, the automata formalism turned possible the verification at a low overhead, enabling the usage of a very frequent event set with a little impact in the system, as shown in Chapter 5. The practical outcomes of this research received significant attention from the Linux developers, mainly for those that prospect using Linux on safety-critical systems.

Regarding the *timing* behavior, the approach taken in Chapter 6 shows that it was possible to describe the properties and the dynamics of part of the Linux kernel behavior without actually touching the kernel code. These findings closed an important problem that was described and discussed with the Linux kernel community by researchers back in 2009 (BRANDENBUG; ANDERSON, 2009). It is important to note that this problem stayed open not because of a *rivalry* but because of the complexity of translating the kernel behavior to the real-time scheduling formalism. The proposed model successfully filled this gap.

## 7.1 THE FUTURE OF THE MODEL

At this point, it is necessary to discuss the future of the model. The event set used in the thread synchronization model roots the addition of the symmetric multiprocessing (SMP) support and the early development of the PREEMPT_RT, being part of the kernel for at least one decade. For instance, the research presented in (OLIVEIRA; OLIVEIRA, 2016) uses almost the same event set, and it is the result of a study that started back in 2010. Given the maturity of the current model and the satisfactory results, now formally unveiled and confirmed in Chapter 6, it is reasonable to assume that no changes are expected in the foreseeable future.

Nevertheless, it is the nature of the kernel to change, but that is not the end of the model. Instead, it will be a chance to show another value of the model. The model assumes that the current design is correct, based on the fact that the kernel has been working in this way for a long time. So, during the development, the non-functional verifications enabled by the automata formalism were used to ensure that the model was following this assumption, for instance, by not presenting deadlocks or livelocks. The future changes of Linux might be tried first in the model, enabling the powerful arsenal of formal methods that requires a model, especially model checking (CLARKE; EMERSON; SIFAKIS, 2009).

The modular approach will facilitate the changes in the model, but that is not the sole benefit. With the modular approach, it will be possible to identify which specifications changed. As a consequence, the technologies that were built upon these specifications will have to be updated. Knowing this is an essential factor for the sustainability of the Linux ecosystem as a whole, mainly for the safety-critical community. But, it is fair to say that the model will receive attention only if it shows practical value

over time, being used daily by developers. This topic opens the future work section.

## 7.2 FUTURE WORK

The proof of concept runtime verification method presented this thesis has the potential to be used in practice. Still, it needs to be enhanced with the addition of an intuitive interface. The interface could be similar to the `ftrace`, including some existing models already compiled with the kernel, but still enabling the dynamic load of new models. The runtime verification options, such as the actions to be taken in the occurrence of a violation, could have a configuration interface as well. Extending the `dot2c` tool to support more complex automata methods, such as the timed automata, is also another possible work. Finally, the method will be proposed for upstream integration, enabling the verification at the development time. Indeed, the feedback from the kernel developers is that this tool will be very useful for the continuous integration (CI) effort[2].

The latency measurement tool already has a practical interface. It will also be extended, including the usage of a database that could record multiple trace sections, enabling the collection of many days of tracing of the same system or the trace of the same system with different kernels. The latency parser tool that runs in-kernel will be proposed upstream as well, being ready to use without involving compiling an external module by the users.

The model presented in this thesis, like any model, can be extended to include other aspects. The first aspect would be the inclusion of the synchronization mechanism used in multicore systems. This effort would include the spin-based locking, described in Section 2.2, and the migration control. Another essential aspect would be the addition of other elements in the single-core (or per-cpu) model. For example, the more fine-grained operations involving the cache memory or concurrent access require memory ordering. The model could also be extended as is, with the inclusion of parameters or the timing aspects already existing in the automata formalism. Although this future work sounds contradictory after the argumentation in favor of the basic automata format, it is not: the simplest format was also selected because of the unknown number of states and transitions that the final model would have, but this is not a problem after the development of this thesis. Moreover, the usage of the simple format was also made with the awareness of the fact that it could be extended later.

Another possible research branch from this work would be the definition of a set of *non-negligible* or *non-atomic* operations involving the scheduling of tasks on Linux. These operations are useful for the definition of a more appropriate task model to be used in the development of novel scheduling algorithms that would fit better in the context of Linux. The approach for the definition of these operations, their behavior,

[2] See: https://kernelci.org/.

and the way they influence the timing aspects of the scheduler could be similar to the one presented in Chapter 6.

## 7.3 LIST OF PUBLICATIONS

The following papers were published based on the research work that has been presented in this thesis:

- D. B. De Oliveira, D. Casini, R. S. De Oliveira. T. Cucinotta. *Demystifying the Real-Time Linux Scheduling Latency*, (to appear) in the Proceedings of the **32th Euromicro Conference on Real-time Systems (ECRTS)**, July 7-10th, 2020, Modena, Italy.
- D. B. De Oliveira, R. S. De Oliveira, T. Cucinotta. *A thread synchronization model for the PREEMPT_RT Linux kernel*, **Elsevier Journal of Systems Architecture (JSA)**, Vol. 107, August 2020.
- D. B. De Oliveira, T. Cucinotta, R. S. De Oliveira. *Efficient formal verification for the Linux kernel*, **17th International Conference on Software Engineering and Formal Methods (SEFM 2019)**, September 16-20th, 2019, Oslo, Norway.
- D. B. de Oliveira, R. S. de Oliveira, T. Cucinotta. *Untangling the Intricacies of Thread Synchronization in the PREEMPT RT Linux Kernel,* in Proceedings of the **22nd IEEE International Symposium on Real-Time Distributed Computing (IEEE ISORC 2019)**, May 7-9, 2019, Valencia, Spain.
- 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 **International Workshop on Embedded Operating Systems (EWILI 2018)**, October 10th, 2018, Torino, Italy.
- D. B. de Oliveira, R. S. de Oliveira, T. Cucinotta, L. Abeni. *Automata-Based Modeling of Interrupts in the Linux PREEMPT RT Kernel*, in Proceedings of the **22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA 2017)**, September 12-15, 2017, Limassol, Cyprus.

### 7.3.1 Other publications

The following papers were also published, but they are not directly related to the work presented in this thesis:

- D. B. de Oliveira, D. Casini, R. S. de Oliveira, T. Cucinotta, A. Biondi and G. Buttazzo. *Nested Locks in the Lock Implementation: The Real-Time Read-Write*

*Semaphores on Linux*, in Proceedings of the **International Real-Time Scheduling Open Problems Seminar (RTSOPS 2018)**, co-located with the 30th Euromicro Conference on Real-Time Systems (ECRTS 2018). July 3, 2018, Barcelona, Spain.

- K. P. Silva, L. F. Arcaro, D. B. de Oliveira. *An Empirical Study on the Adequacy of MBPTA for Tasks Executed on a Complex Computer Architecture with Linux*, in Proceedings of the **23rd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA 2018)**, September 4th - 7th, 2018, Torino, Italy.

## 7.4 INTERACTIONS WITH THE LINUX KERNEL DEVELOPMENT COMMUNITY

One key aspect of this thesis is the interaction with the Linux kernel development community. The interaction aimed to collect feedback about the level of abstraction and the practical aspects of the formalism, for example, to understand if the automata were a viable specification language. But also, to present the proof of concept tools derived from this work, and potential ways to extend them, so to make them widely available to the kernel developers.

Indeed, the usage of automata-based models for runtime verification was a secondary goal of this project, mainly because we were not expecting to find bugs in the kernel during the development of the model. This fact is evidenced by the change in the modeling approach presented in Figure 26 and in the equivalent illustration used in the first publication (OLIVEIRA, D. B. de et al., 2017), which did not consider any change in the kernel, but only from the model. The feedback from the industry and the kernel development community is good evidence that the runtime verification method has the potential to be applied all over the kernel.

The topics covered in this thesis were presented to the Linux kernel community with the following talks:

- *Efficient Runtime Verification for the Linux Kernel*: **Red Hat Research Day Europe, 2020**. Brno, CZ (invited talk).
- *Real-time Linux: what is, what is not and what is next*: **Real-time Linux Summit, 2019**. Lyon, FRA.
- *Formal verification made easy (and fast)!*: **Linux Plumbers Conference, 2019**. Lisbon, PT; **Kernel Recipe**, 2019. Paris, FR (invited talk); **Linux Foundation ELISA project meeting**, 2020. Online (invited talk).
- *Mathemazing the latency*: **Linux Plumbers Conference, 2019**. Lisbon, PT.

- *Mind the gap between real-time Linux and real-time theory*: **Real-time Linux Summit, 2018**. Edinburgh, UK; **Linux Plumbers Conference, 2018**. Vancouver, CA.
- *How can we catch problems that can break the PREEMPT_RT preemption model?*: **Linux Plumbers Conference, 2018**. Vancouver, CA.
- *Beyond the latency: New metrics for the real-time kernel*: **Linux Plumbers Conference, 2018**. Vancouver, CA.

## 7.5 ACKNOWLEDGMENT

This work was developed while the Ph.D. candidate was employed by Red Hat, Inc. Red Hat supported the development of this work.

## REFERENCES

ABDEDDAÏM, Yasmina; MALER, Oded. Job-Shop Scheduling Using Timed Automata? In: BERRY, Gérard; COMON, Hubert; FINKEL, Alain (Eds.). **Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 478–492.

ABENI, L. et al. A measurement-based analysis of the real-time performance of linux. In: PROCEEDINGS. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium. San Jose, California: IEEE, 2002. p. 133–142. DOI: 10.1109/RTTAS.2002.1137388.

AKESSON, K. et al. Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. In: 2006 8th International Workshop on Discrete Event Systems. Ann Arbor, MI, USA: IEEE, July 2006. p. 384–385. DOI: 10.1109/WODES.2006.382401.

ALBARGHOUTHI, Aws et al. Ufo: A framework for abstraction-and interpolation-based software verification. In: SPRINGER. INTERNATIONAL Conference on Computer Aided Verification. Berkeley, CA, USA: Springer, 2012. p. 672–678.

ALGLAVE, Jade et al. 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. Williamsburg, VA, USA: ACM, 2018. (ASPLOS '18), p. 405–418. DOI: 10.1145/3173162.3177156. Available from: http://doi.acm.org/10.1145/3173162.3177156.

AMNELL TOBIAS AND FERSMAN, Elena et al. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In: LARSEN, Kim Guldstrand; NIEBERT, Peter (Eds.). **Formal Modeling and Analysis of Timed Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 60–72.

AMNELL, Tobias et al. TIMES b— A Tool for Modelling and Implementation of Embedded Systems. In: KATOEN, Joost-Pieter; STEVENS, Perdita (Eds.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 460–464.

ANDRÉ, Étienne et al. IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems. In: GIANNAKOPOULOU, Dimitra; MÉRY, Dominique (Eds.). **Proceedings of the 18th International Symposium on Formal Methods (FM'12)**. Paris, France: Springer, Aug. 2012. (Lecture Notes in Computer Science), p. 33–36. Available from: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/AFKS-fm12.pdf.

Parametric Schedulability Analysis of Fixed Priority Real-Time Distributed Systems. In: ARTHO, Cyrille; ÖLVECZKY, Peter Csaba (Eds.). **Formal Techniques for Safety-Critical Systems**. Cham: Springer International Publishing, 2014. p. 212–228.

AUDSLEY, Neil et al. Applying new scheduling theory to static priority pre-emptive scheduling. **Software engineering journal**, v. 8, n. 5, p. 284–292, 1993.

B. B. BRANDENBURG. **Scheduling and Locking in Multiprocessor Real-Time Operating Systems**. 2011. PhD thesis – University of North Carolina at Chapel Hill. Available from: `https://cs.unc.edu/~anderson/diss/bbbdiss.pdf`.

BALL, Thomas; LEVIN, Vladimir; RAJAMANI, Sriram K. A Decade of Software Model Checking with SLAM. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 7, p. 68–76, July 2011. ISSN 0001-0782. DOI: `10.1145/1965724.1965743`. Available from: `https://doi.org/10.1145/1965724.1965743`.

BALL, Thomas; RAJAMANI, Sriram K. **SLIC: A Specification Language for Interface Checking (of C)**. Online, Jan. 2002. p. 12. Available from: `https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/`. Visited on: 26 Mar. 2020.

BALL, Thomas; RAJAMANI, Sriram K. The SLAM Project: Debugging System Software via Static Analysis. In: PROCEEDINGS of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, Oregon: ACM, 2002. (POPL '02), p. 1–3. DOI: `10.1145/503272.503274`.

BASTONI, A.; BRANDENBURG, B.B.; ANDERSON, J.H. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In: REAL-TIME Systems Symposium (RTSS), 2010 IEEE 31st. San Diego, CA, USA: IEEE, Nov. 2010. p. 14–24. DOI: `10.1109/RTSS.2010.23`.

BEYER, Dirk; KEREMOGLU, M Erkan. CPAchecker: A tool for configurable software verification. In: SPRINGER. INTERNATIONAL Conference on Computer Aided Verification. Snowbird, UT, USA: Springer, 2011. p. 184–190.

BLACKHAM, B. et al. Timing Analysis of a Protected Operating System Kernel. In: PROCEEDINGS of the 32nd IEEE Real-Time Systems Symposium (RTSS11). Vienna, Austria: IEEE, Nov. 2011. p. 339–348.

BLOCK, A.; ANDERSON, J. H. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. In: 12TH International Conference on Parallel and Distributed Systems - (ICPADS'06). Minneapolis, MN: IEEE, July 2006. 10 pp.-. DOI: `10.1109/ICPADS.2006.21`.

BOUAJJANI, A.; TRIPAKIS, S.; YOVINE, S. On-the-fly symbolic model checking for real-time systems. In: PROCEEDINGS Real-Time Systems Symposium. San Francisco, CA: IEEE, Dec. 1997. p. 25–34. DOI: `10.1109/REAL.1997.641266`.

BRANDENBUG, Bjorn; ANDERSON, James. Joint Opportunities for Real-Time Linux and Real-Time System Research. In: PROCEEDINGS OF THE 11TH REAL-TIME LINUX WORKSHOP (RTLWS 2009). Dresden, Germany: [s.n.], Sept. 2009. p. 19–30.

BRANDENBURG, B. B.; 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). Pisa, Italy: IEEE, July 2007. p. 61–70. DOI: 10.1109/ECRTS.2007.17.

BRANDENBURG, B. B.; GÜL, M. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In: 2016 IEEE Real-Time Systems Symposium (RTSS). Porto, Portugal: IEEE, Nov. 2016. p. 99–110. DOI: 10.1109/RTSS.2016.019.

BRANDENBURG, Bjorn B.; ANDERSON, James H. 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). Pisa, italy: [s.n.], 2007. p. 61–70.

BROWN, Jeremy H; MARTIN, Brad. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. In: PROC. of the 12th Real-Time Linux Workshop (RTLWS'12). Nairobi, Kenya: RTLWS, 2010. p. 1–17.

BUTTAZZO, Giorgio C. **Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications**. 3rd. New York, USA: Springer Publishing Company, Incorporated, 2011. ISBN 1461406757, 9781461406754.

CALANDRINO, John M. et al. 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: IEEE Computer Society, 2006. (RTSS '06), p. 111–126. DOI: 10.1109/RTSS.2006.27.

CASSANDRAS, Christos G.; LAFORTUNE, Stephane. **Introduction to Discrete Event Systems**. 2nd. Boston, MA: Springer Publishing Company, Incorporated, 2010. ISBN 1441941193, 9781441941190.

CASSAR, Ian; FRANCALANZA, Adrian. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based systems. In: CÁMARA, Javier; PROENÇA, José (Eds.). Proceedings 13th International Workshop on **Foundations of Coordination Languages and Self-Adaptive Systems**. Rome, Italy: Open Publishing Association, 2015. (Electronic Proceedings in Theoretical Computer Science), p. 54–68. DOI: 10.4204/EPTCS.175.4.

CENTER, Thomas J. Watson IBM Research et al. **An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types**.

Cambridge, USA: IBM Thomas J. Watson Research Division, 1976. (Research reports // IBM). Available from: https://books.google.it/books?id=Hi1yPgAACAAJ.

CERQUEIRA, Felipe; BRANDENBURG, Björn. A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT. In: SYSGO AG. 9TH Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications. Paris, France: OSPERT, 2013. p. 19–29.

CHAKI, S. et al. Modular verification of software components in C. **IEEE Transactions on Software Engineering**, v. 30, n. 6, p. 388–402, June 2004. ISSN 0098-5589. DOI: 10.1109/TSE.2004.22.

CHAKI, Sagar et al. Concurrent software verification with states, events, and deadlocks. **Formal Aspects of Computing**, v. 17, n. 4, p. 461–483, Dec. 2005. ISSN 1433-299X. DOI: 10.1007/s00165-005-0071-z. Available from: https://doi.org/10.1007/s00165-005-0071-z.

CHISHIRO, H. RT-Seed: Real-Time Middleware for Semi-Fixed-Priority Scheduling. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). York, UK: IEEE, May 2016. p. 124–133. DOI: 10.1109/ISORC.2016.26.

CIMATTI, A.; PALOPOLI, L.; RAMADIAN, Y. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In: 2008 Real-Time Systems Symposium. Barcelona, Spain: IEEE, Nov. 2008. p. 80–89. DOI: 10.1109/RTSS.2008.36.

CLARKE, Edmund M.; EMERSON, E. Allen; SIFAKIS, Joseph. Model Checking: Algorithmic Verification and Debugging. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 11, p. 74–84, Nov. 2009. ISSN 0001-0782. DOI: 10.1145/1592761.1592781. Available from: https://doi.org/10.1145/1592761.1592781.

CLARKE, Edmund; KROENING, Daniel; LERDA, Flavio. A Tool for Checking ANSI-C Programs. In: JENSEN, Kurt; PODELSKI, Andreas (Eds.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 168–176.

CLARKE, Edmund; KROENING, Daniel; SHARYGINA, Natasha, et al. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: HALBWACHS, Nicolas; ZUCK, Lenore D. (Eds.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 570–574.

CONDLIFFE, Jamie. **U.S. Military Drones Are Going to Start Running on Linux**. Online: GIZMODO, July 2014. Available from: https://gizmodo.com/u-s-military-drones-are-going-to-start-running-on-linu-1572853572. Visited on: 26 Mar. 2020.

CORBET, J. **Linux at NASDAQ OMX**. Boulder, Colorado: LWN.net, Oct. 2010. Available from: https://lwn.net/Articles/411064/. Visited on: 26 Mar. 2020.

CORBET, J. **Statistics from the 4.17 kernel development cycle**. Boulder, Colorado: LWN.net, May 2018. Available from: https://lwn.net/Articles/756031/. Visited on: 26 Mar. 2020.

CORBET, J. **The kernel lock validator**. Boulder, Colorado: LWN.net, May 2006. Available from: https://lwn.net/Articles/185666/. Visited on: 26 Mar. 2020.

CORBET, Jonathan. **Jump label**. Boulder, Colorado: LWN.net, Oct. 2010. Available from: https://lwn.net/Articles/412072/. Visited on: 26 Mar. 2020.

CORBET, Jonathan; RUBINI, Alessandro; KROAH-HARTMAN, Greg. **Linux Device Driver**. 3. ed. Sebastopol, CA, USA: O'Reilly Media, 2005.

COTRONEO, Domenico; DI LEO, Domenico, et al. A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain. In: FLAMMINI, Francesco; BOLOGNA, Sandro; VITTORINI, Valeria (Eds.). **Computer Safety, Reliability, and Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 213–227.

COTRONEO, Domenico; LEO, Domenico Di, et al. SABRINE: State-based Robustness Testing of Operating Systems. In: PROCEEDINGS of the 28th IEEE/ACM International Conference on Automated Software Engineering. Silicon Valley, CA, USA: IEEE Press, 2013. (ASE'13), p. 125–135. DOI: 10.1109/ASE.2013.6693073. Available from: https://doi.org/10.1109/ASE.2013.6693073.

CUCINOTTA, T. et al. A Real-Time Service-Oriented Architecture for Industrial Automation. **IEEE Transactions on Industrial Informatics**, v. 5, n. 3, p. 267–277, Aug. 2009. ISSN 1551-3203. DOI: 10.1109/TII.2009.2027013.

DAWS, C.; YOVINE, S. Two examples of verification of multirate timed automata with Kronos. In: PROCEEDINGS 16th IEEE Real-Time Systems Symposium. Pisa, Italy: IEEE, Dec. 1995. p. 66–75. DOI: 10.1109/REAL.1995.495197.

DRONAMRAJU, Srikar. **Uprobe-tracer: Uprobe-based Event Tracing**. Online: Linux Kernel Documentation, May 2019. Available from: https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt. Visited on: 26 Mar. 2020.

DUBEY, A.; KARSAI, G.; ABDELWAHED, S. 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. Tokyo, Japan: IEEE, Mar. 2009. p. 55–62. DOI: 10.1109/ISORC.2009.28.

EHRIG, Hartmut et al. Introduction to algebraic specification. Part 1: Formal methods for software development. **The Computer Journal**, The British Computer Society, Oxford, UK, v. 35, n. 5, p. 460–467, 1992.

ELLSON, John et al. Graphviz— Open Source Graph Drawing Tools. In: MUTZEL, Petra; JÜNGER, Michael; LEIPERT, Sebastian (Eds.). **Graph Drawing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 483–484.

EPSTEIN, Joshua M. Why Model? **Journal of Artificial Societies and Social Simulation**, v. 11, n. 4, p. 12, 2008. ISSN 1460-7425. Available from: http://jasss.soc.surrey.ac.uk/11/4/12.html.

FALCONE, Yliès et al. A taxonomy for classifying runtime verification tools. In: SPRINGER. INTERNATIONAL Conference on Runtime Verification. Limassol, Cyprus: Springer, 2018. p. 241–262.

FAYYAD-KAZAN, Hasan; PERNEEL, Luc; TIMMERMAN, Martin. Linux PREEMPT-RT vs Commercial RTOSs: How Big is the Performance Gap? **GSTF Journal on Computing**, v. 3, n. 1, 2013.

FERSMAN, Elena; MOKRUSHIN, Leonid, et al. Schedulability Analysis of Fixed-Priority Systems Using Timed Automata. **Theor. Comput. Sci.**, Elsevier Science Publishers Ltd., GBR, v. 354, n. 2, p. 301–317, Mar. 2006. ISSN 0304-3975. DOI: 10.1016/j.tcs.2005.11.019. Available from: https://doi.org/10.1016/j.tcs.2005.11.019.

FERSMAN, Elena; PETTERSSON, Paul; YI, Wang. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In: KATOEN, Joost-Pieter; STEVENS, Perdita (Eds.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 67–82.

GERUM, Philippe. Xenomai-Implementing a RTOS emulation framework on GNU/Linux. **White Paper, Xenomai**, p. 1–12, 2004.

GLEIXNER, Thomas. Realtime Linux: academia v. reality. **Linux Weekly News**, LWN.net, Boulder, Colorado, July 2010. Available from: https://lwn.net/Articles/397422/. Visited on: 26 Mar. 2020.

GUTIÉRREZ, Carlos San Vicente et al. Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications. **CoRR**, abs/1808.10821, 2018. arXiv: 1808.10821. Available from: http://arxiv.org/abs/1808.10821.

HENZINGER, Thomas A. et al. Lazy Abstraction. In: PROCEEDINGS of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, Oregon: ACM, 2002. (POPL '02), p. 58–70. DOI: 10.1145/503272.503279.

HERZOG, B. et al. INTspect: Interrupt Latencies in the Linux Kernel. In: 2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC). Salvador, Brazil, Brazil: IEEE, Nov. 2018. p. 83–90.

JOSEPH, M.; PANDYA, P. Finding Response Times in a Real-Time System. **The Computer Journal**, v. 29, n. 5, p. 390–395, Jan. 1986.

KAYNAR, D. K. et al. Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems. In: RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003. Cancun, Mexico: IEEE, Dec. 2003. p. 166–177. DOI: 10.1109/REAL.2003.1253264.

KENNA, C.J. et al. Soft Real-Time on Multiprocessors: Are Analysis-Based Schedulers Really Worth It? In: REAL-TIME Systems Symposium (RTSS), 2011 IEEE 32nd. Vienna, Austria: IEEE, Nov. 2011. p. 93–103. DOI: 10.1109/RTSS.2011.16.

KLECH, Jaroslav et al. **Advanced tuning procedures to optimize latency in RHEL for Real Time**. Online: Red Hat Customer Portal, Feb. 2020. Available from: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/tuning_guide/index. Visited on: 26 Mar. 2020.

KLEIN, Gerwin et al. seL4: Formal Verification of an OS Kernel. In: PROCEEDINGS of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. Big Sky, Montana, USA: ACM, 2009. (SOSP '09), p. 207–220. DOI: 10.1145/1629575.1629596.

LAMPKA, Kai; PERATHONER, Simon; THIELE, Lothar. Component-based system design: analytic real-time interfaces for state-based component implementations. **International Journal on Software Tools for Technology Transfer**, v. 15, n. 3, p. 155–170, June 2013. ISSN 1433-2787. DOI: 10.1007/s10009-012-0257-7. Available from: https://doi.org/10.1007/s10009-012-0257-7.

LAMPORT, Leslie. The Temporal Logic of Actions. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 16, n. 3, p. 872–923, May 1994. ISSN 0164-0925. DOI: 10.1145/177492.177726.

LARSEN, Kim G. et al. Robust synthesis for real-time systems. **Theoretical Computer Science**, v. 515, p. 96–122, 2014. ISSN 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2013.08.015. Available from: http://www.sciencedirect.com/science/article/pii/S0304397513006397.

LE, Thi Thieu Hoa et al. Timed-automata based schedulability analysis for distributed firm real-time systems: a case study. **International Journal on Software Tools for Technology Transfer**, v. 15, n. 3, p. 211–228, June 2013. ISSN 1433-2787. DOI: 10.1007/s10009-012-0245-y. Available from: https://doi.org/10.1007/s10009-012-0245-y.

LEHOCZKY, J.; SHA, L.; DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In: [1989] Proceedings. Real-Time Systems Symposium. Santa Monica, CA, USA: IEEE, Dec. 1989. p. 166–171.

LEI, Bin et al. State Based Robustness Testing for Components. **Electron. Notes Theor. Comput. Sci.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 260, p. 173–188, Jan. 2010. ISSN 1571-0661. DOI: 10.1016/j.entcs.2009.12.037. Available from: http://dx.doi.org/10.1016/j.entcs.2009.12.037.

LELLI, Juri et al. Deadline scheduling in the Linux kernel. **Software: Practice and Experience**, v. 46, n. 6, p. 821–839, 2016. ISSN 1097-024X. DOI: 10.1002/spe.2335.

LI, P. et al. A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems. **IEEE Transactions on Software Engineering**, v. 30, n. 9, p. 613–629, Sept. 2004. ISSN 0098-5589. DOI: 10.1109/TSE.2004.45.

LINUX FOUNDATION. **Automotive Grade Linux**. Online: The Linux Foundation Projects, 2016. Available from: https://www.automotivelinux.org/. Visited on: 26 Mar. 2020.

LIU, Jane W. S. W. **Real-Time Systems**. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN 0130996513.

LOVE, Robert. **Linux Kernel Development**. 3. ed. Crawfordsville, Indiana: Addison-Wesley, 2010.

LYNCH, Nancy; SEGALA, Roberto; VAANDRAGER, Frits. Hybrid I/O automata. **Information and Computation**, v. 185, n. 1, p. 105–157, 2003. ISSN 0890-5401. DOI: https://doi.org/10.1016/S0890-5401(03)00067-1. Available from: http://www.sciencedirect.com/science/article/pii/S0890540103000671.

MANTEGAZZA, Paolo et al. RTAI: Real-time application interface. -Specialized Systems Consultants Incorporated: PO Box 55549: Seattle, WA . . ., 2000.

MARINAS, Catalin. **Formal Methods for Kernel Hackers**. Vancouver, CA: Linux Plumbers Conference, 2018. Available from: https://linuxplumbersconf.org/event/2/contributions/60/attachments/18/42/FormalMethodsPlumbers2018.pdf. Visited on: 26 Mar. 2020.

MATNI, G.; DAGENAIS, M. Automata-based approach for kernel trace analysis. In: 2009 Canadian Conference on Electrical and Computer Engineering. St. John's, NL, Canada: IEEE, May 2009. p. 970–973. DOI: 10.1109/CCECE.2009.5090273.

MCKENNEY, Paul. **A realtime preemption overview**. Boulder, Colorado: LWN.net, Aug. 2005. Available from: https://lwn.net/Articles/146861/. Visited on: 26 Mar. 2020.

NORSTROM, C.; WALL, A.; WANG YI. Timed automata as task models for event-driven systems. In: PROCEEDINGS Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306). Hong Kong, China, China: IEEE, Dec. 1999. p. 182–189. DOI: 10.1109/RTCSA.1999.811218.

O'REGAN, Gerard. **Concise guide to formal methods**. Berlin, Heidelberg: Springer, 2017.

OLIVEIRA, D. B. de et al. Automata-based modeling of interrupts in the Linux PREEMPT RT kernel. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). Limassol, Cyprus: IEEE, Sept. 2017. p. 1–8. DOI: 10.1109/ETFA.2017.8247611.

OLIVEIRA, Daniel B. de; OLIVEIRA, Rômulo S. de; CUCINOTTA, Tommaso. A Thread Synchronization Model for the PREEMPT_RT Linux Kernel. **Journal of Systems Architecture**, p. 101729, 2020. ISSN 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101729. Available from: http://www.sciencedirect.com/science/article/pii/S1383762120300230.

OLIVEIRA, Daniel Bristot de. **__schedule() being called twice, the second in vain**. Pisa, Italy: Daniel's page, July 2018. Available from: http://bristot.me/__schedule-being-called-twice-the-second-in-vain/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Beyond the latency: New metrics for the real-time kernel**. Vancouver, CA: Linux Plumbers Conference, 2018. Available from: https://linuxplumbersconf.org/event/2/contributions/241/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **BUG-RT: scheduling while in atomic in the watchdog's hrtimer**. Online: Linux Kernel Mainling List, 2019. Available from: https://www.spinics.net/lists/linux-rt-users/msg20376.html. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **BUG: ftrace/perf dropping events at the begin of interrupt handlers**. Online: Linux Kernel Mainling List, 2018. Available from: https://www.spinics.net/lists/linux-rt-users/msg19781.html. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Early context tracking patch set: fixing perf and ftrace losing events**. Pisa, Italy: Daniel's page, 2019. Available from:

http://bristot.me/early-context-tracking-patch-set-fixing-perf-ftrace-losing-events/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **How can we catch problems that can break the PREEMPT_RT preemption model?** Vancouver, CA: Linux Plumbers Conference, 2018. Available from: https://linuxplumbersconf.org/event/2/contributions/190/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Linux Task Model**. Pisa, Italy: Daniel's page, 2019. Available from: http://bristot.me/linux_task_model/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Mathemizing the Latency**. Lisbon, PT: Linux Plumbers Conference, 2019. Available from: https://linuxplumbersconf.org/event/4/contributions/413/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Mind the gap between real-time Linux and real-time theory, Part I**. Edinburgh, UK: Real-time Linux Summit, Edinburg, 2018. Available from: https://wiki.linuxfoundation.org/realtime/events/rt-summit2018/schedule%5C#abstracts. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de. **Mind the gap between real-time Linux and real-time theory, Part II**. Vancouver, CA: Linux Plumbers Conference, 2018. Available from: https://www.linuxplumbersconf.org/event/2/contributions/75/. Visited on: 26 Mar. 2020.

OLIVEIRA, Daniel Bristot de; CUCINOTTA, Tommaso; OLIVEIRA, Rômulo Silva de. Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel. In: PROC. of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). Valencia, Spain: IEEE, May 2019.

OLIVEIRA, Daniel Bristot de; OLIVEIRA, Rômulo Silva de. Timing analysis of the PREEMPT_RT Linux kernel. **Softw., Pract. Exper.**, v. 46, n. 6, p. 789–819, 2016. DOI: 10.1002/spe.2333.

OLIVEIRA, Daniel Bristot de et al. 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). Barcelona, Spain: RTSOPS, July 2018.

PALOPOLI, L. et al. AQuoSA – Adaptive Quality of Service Architecture. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 39, n. 1, p. 1–31, Jan. 2009. ISSN 0038-0644. DOI: 10.1002/spe.v39:1. Available from: http://dx.doi.org/10.1002/spe.v39:1.

PHORONIX. **Phoronix Test Suite: Open-Source, Automated Benchmarking**. Online: Phoronix Test Suite, Feb. 2020. Available from: https://www.phoronix-test-suite.com. Visited on: 26 Mar. 2020.

POIMBOEUF, Josh. **Introducing kpatch: Dynamic Kernel Patching**. Online: Red Hat Blog, Feb. 2014. Available from: https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching. Visited on: 26 Mar. 2020.

POSADAS, H. et al. 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. Carmona, Seville, Spain: IEEE, May 2010. p. 238–244. DOI: 10.1109/ISORC.2010.18.

POST, Hendrik; KÜCHLIN, Wolfgang. Integrated static analysis for Linux device driver verification. In: SPRINGER. INTERNATIONAL Conference on Integrated Formal Methods. Oxford, UK: Springer, 2007. p. 518–537.

PULLUM, Laura L. **Software Fault Tolerance Techniques and Implementation**. Norwood, MA, USA: Artech House, Inc., 2001. ISBN 1-58053-137-7.

RAMADGE, P. J.; WONHAM, W. M. Supervisory Control of a Class of Discrete Event Processes. **SIAM J. Control Optim.**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 25, n. 1, p. 206–230, Jan. 1987. ISSN 0363-0129. DOI: 10.1137/0325013.

RED HAT. INC. **Test Suite User Guide**. Online: Red Hat Customer Portal, Feb. 2020. Available from: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_hardware_certification/1.0/html/test_suite_user_guide/sect-layered-product-certs%5C#cert-for-rhel-for-real-time. Visited on: 26 Mar. 2020.

REGHENZANI, F.; MASSARI, G.; FORNACIARI, W. Mixed Time-Criticality Process Interferences Characterization on a Multicore Linux System. In: 2017 Euromicro Conference on Digital System Design (DSD). Vienna, Austria: IEEE, Aug. 2017.

REGNIER, Paul; LIMA, George; BARRETO, Luciano. Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems. v. 42, n. 6, p. 52–63, 2008.

ROSTEDT, S. **Using KernelShark to analyze the real-time scheduler**. Boulder, Colorado: LWN.net, Feb. 2011. Available from: https://lwn.net/Articles/425583/. Visited on: 26 Mar. 2020.

ROSTEDT, Steven. Finding origins of latencies using ftrace, 2009.

ROSTEDT, Steven. Secrets of the Ftrace function tracer. **Linux Weekly News**, LWN.net, Boulder, Colorado, Jan. 2010. Available from: http://lwn.net/Articles/370423/. Visited on: 26 Mar. 2020.

SHAHPASAND, R.; SEDAGHAT, Y.; PAYDAR, S. Improving the stateful robustness testing of embedded real-time operating systems. In: 2016 6th International Conference on Computer and Knowledge Engineering (ICCKE). Mashhad, Iran: IEEE, Oct. 2016. p. 159–164. DOI: 10.1109/ICCKE.2016.7802133.

SOUTO, Pedro et al. Overhead-Aware Schedulability Evaluation of Semi-Partitioned Real-Time Schedulers. In: PROCEEDINGS of the 2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications. USA: IEEE Computer Society, 2015. (RTCSA '15), p. 110–121. DOI: 10.1109/RTCSA.2015.13. Available from: https://doi.org/10.1109/RTCSA.2015.13.

SPEAR, A.; LEVY, M.; DESNOYERS, M. Using Tracing to Solve the Multicore System Debug Problem. **Computer**, v. 45, n. 12, p. 60–64, Dec. 2012. ISSN 0018-9162. DOI: 10.1109/MC.2012.191.

SPIVEY, J. M. **The Z Notation: A Reference Manual**. USA: Prentice-Hall, Inc., 1989. ISBN 013983768X.

SUN, Youcheng; LIPARI, Giuseppe. A Weak Simulation Relation for Real-Time Schedulability Analysis of Global Fixed Priority Scheduling Using Linear Hybrid Automata. In: PROCEEDINGS of the 22nd International Conference on Real-Time Networks and Systems. Versaille, France: Association for Computing Machinery, 2014. (RTNS '14), p. 35–44. DOI: 10.1145/2659787.2659814. Available from: https://doi.org/10.1145/2659787.2659814.

TOUPIN, D. Using Tracing to Diagnose or Monitor Systems. **IEEE Software**, v. 28, n. 1, p. 87–91, Jan. 2011. ISSN 0740-7459. DOI: 10.1109/MS.2011.20.

VARDHAN, Vibhore et al. GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy. **IJES**, v. 4, n. 2, p. 152–169, 2009. DOI: 10.1504/IJES.2009.027939. Available from: https://doi.org/10.1504/IJES.2009.027939.

WANG, X.; LI, Z.; WONHAM, W. M. Dynamic Multiple-Period Reconfiguration of Real-Time Scheduling Based on Timed DES Supervisory Control. **IEEE Transactions on Industrial Informatics**, v. 12, n. 1, p. 101–111, Feb. 2016. ISSN 1551-3203. DOI: 10.1109/TII.2015.2500161.

WITKOWSKI, Thomas et al. Model Checking Concurrent Linux Device Drivers. In: PROCEEDINGS of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. Atlanta, Georgia, USA: ACM, 2007. (ASE '07), p. 501–504. DOI: 10.1145/1321631.1321719.

YOVINE, Sergio. Kronos: A verification tool for real-time systems. **International Journal on Software Tools for Technology Transfer**, Springer, v. 1, n. 1-2, p. 123–133, 1997.

ZAKHAROV, Ilja S. et al. Configurable toolset for static verification of operating systems kernel modules. **Programming and Computer Software**, v. 41, n. 1, p. 49–64, 2015. DOI: 10.1134/S0361768815010065. Available from: https://doi.org/10.1134/S0361768815010065.

ZAKHAROV, Ilja et al. Generating Environment Model for Linux Device Drivers. In: DOI: 10.15514/SYRCOSE-2013-7-13.