Sant'Anna
School of Advanced Studies – Pisa

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

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

Candidate:
Daniel Bristot de Oliveira

Supervisors:
Rômulo Silva de Oliveira
Tommaso Cucinotta

Red Hat

Real-Time Linux

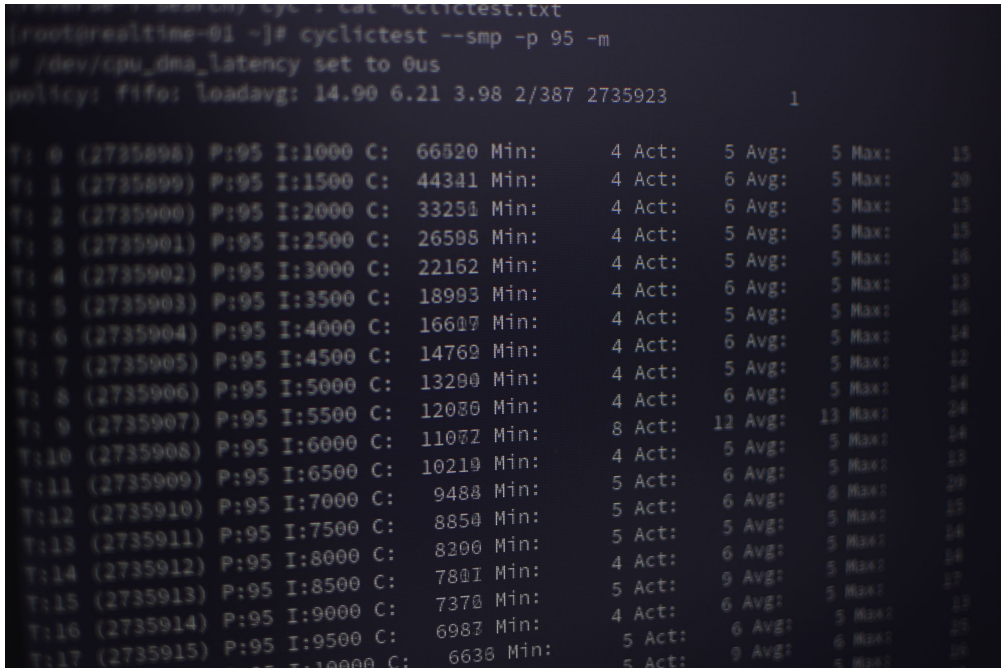
Red Hat

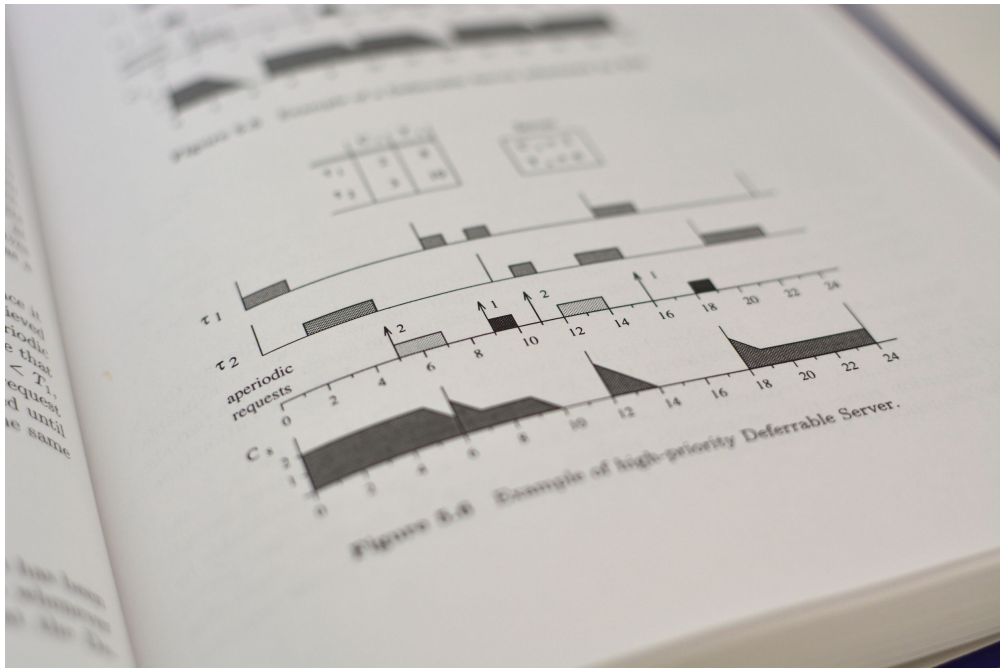
"Real-Time" Linux

Red Hat

Real-Time Linux vs Real-Time theory

Experimental vs Analytical

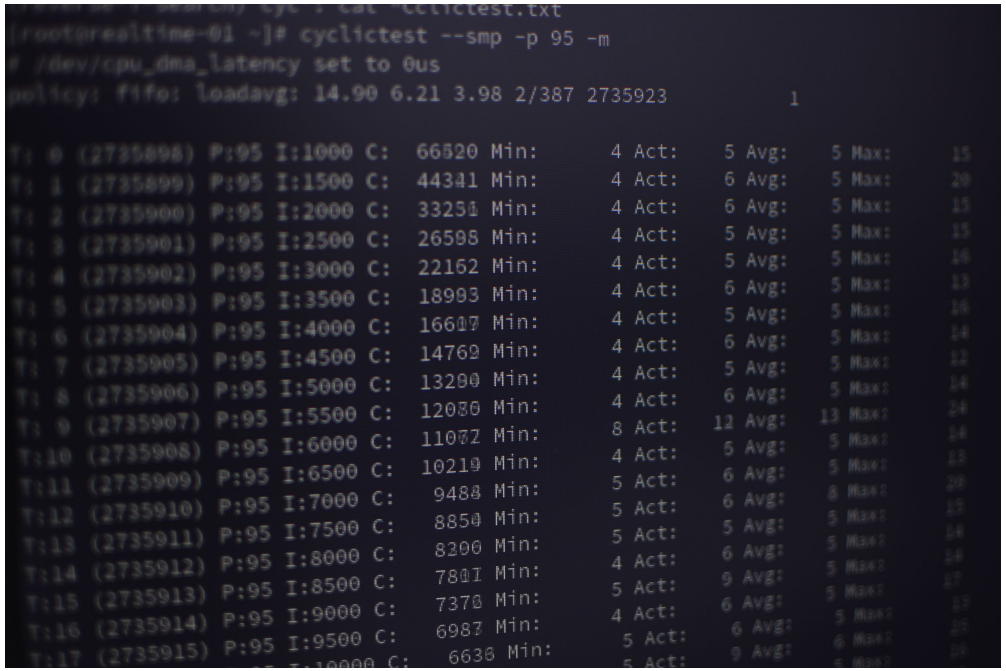


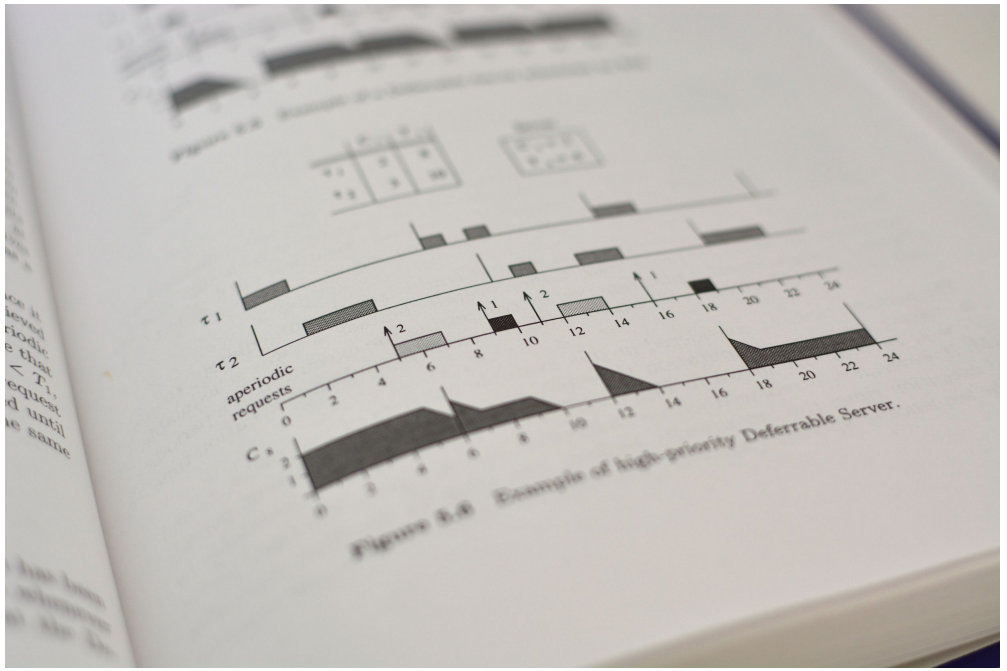


Red Hat

Real-Time Linux vs Real-Time theory

Complex synchronization vs simple models





Red Hat

How can we fill the gap between real-time Linux and real-time theory?

Describing real-time properties of Linux as in theory!

Easier said than done :-)

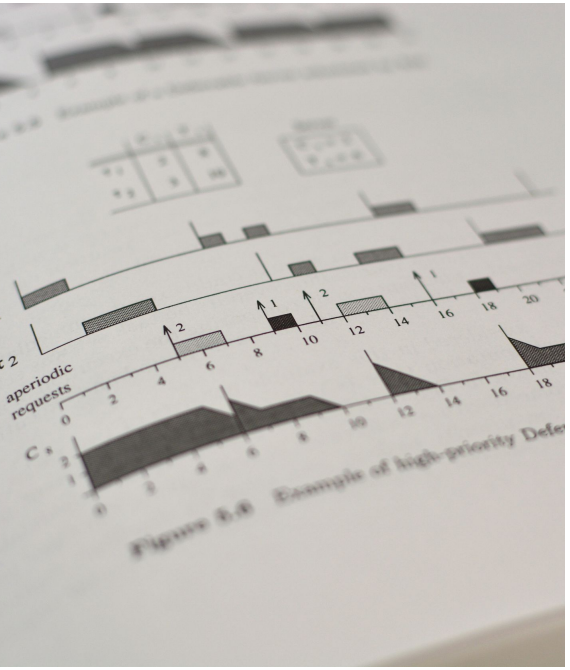
Linux is complex

- Lots of contexts
- Lots of hacks
- Lots of information
- Fast pacing
- ...

Red Hat

Panic!

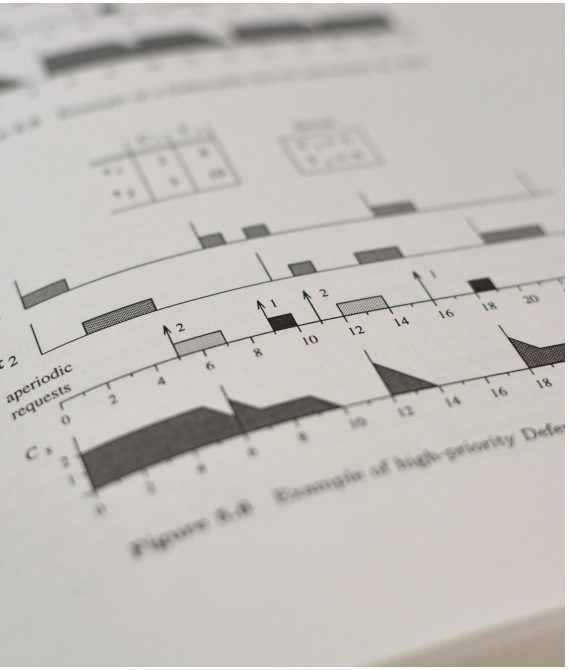
rebooting...



What does real-time mean?

Red Hat

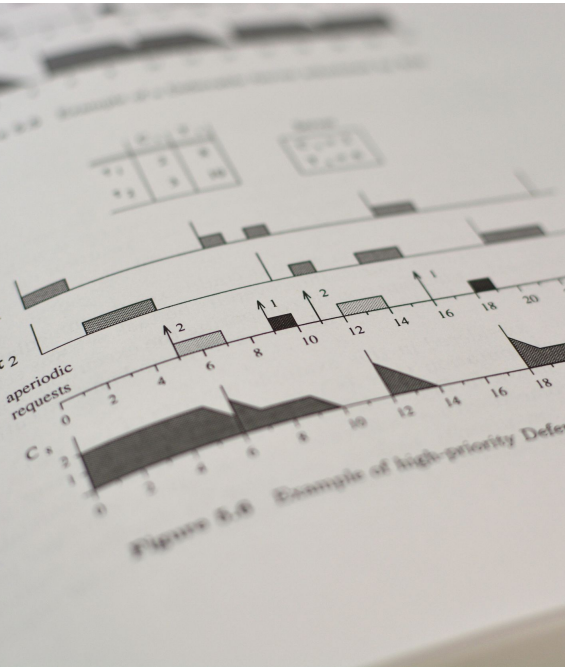
Real-time definition



Real-time systems are computing systems where the correct behavior **does not depend only on the logical behavior, but also on the timing behavior**.

Real-time Linux

To remove “” from real-time Linux...

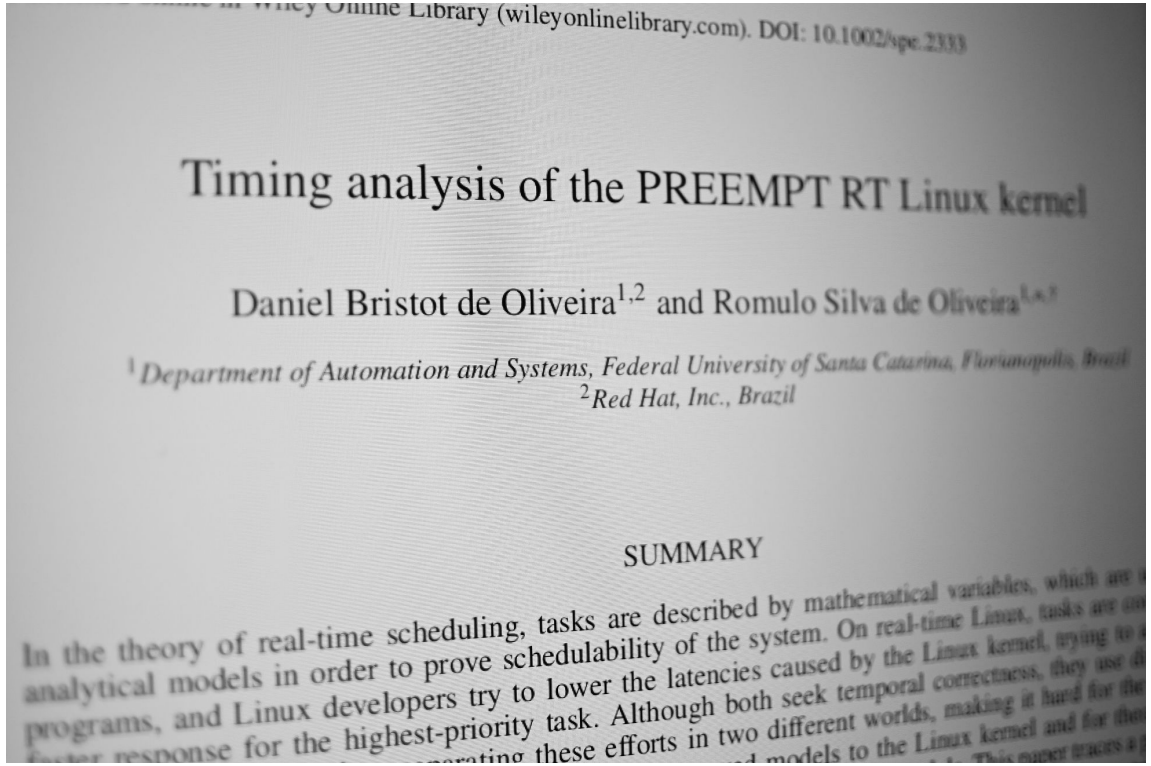


We need to show that Linux has a correct

logical and timing behavior.

Red Hat

An informal[2] try



Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2333

Timing analysis of the PREEMPT RT Linux kernel

Daniel Bristot de Oliveira[1,2] and Romulo Silva de Oliveira[1,*,†]

[1] *Department of Automation and Systems, Federal University of Santa Catarina, Florianopolis, Brazil*
[2] *Red Hat, Inc., Brazil*

SUMMARY

In the theory of real-time scheduling, tasks are described by mathematical variables, which are
analytical models in order to prove schedulability of the system. On real-time Linux, tasks are
programs, and Linux developers try to lower the latencies caused by the Linux kernel, trying to
response for the highest-priority task. Although both seek temporal correctness, they use
these efforts in two different worlds, making it hard for the
models to the Linux kernel and for the

Using an informal language, and Informally enrolled in the PhD.

Published at “Software: practice and experience.”

Building evidence that I could do the PhD as a partial-time student, and learning.

Red Hat

Timing analysis of the PREEMPT RT Linux kernel

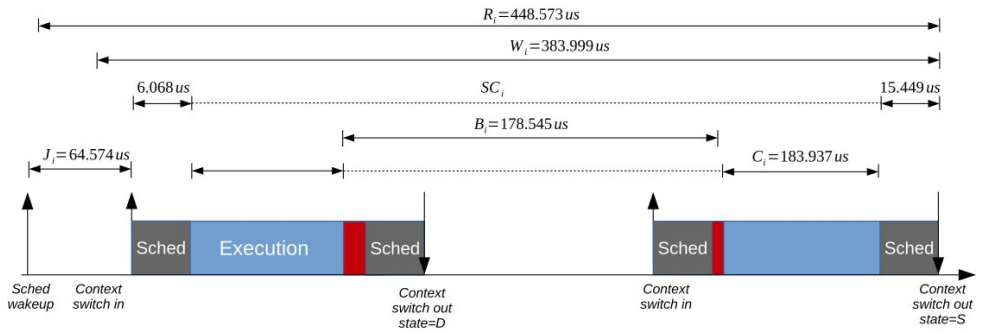


Figure 10. Timeline of task pi-839.

The execution time of the system call *read()* was 191.341 µs, which is much greater than 5.609 µs, which was its execution time in the absence of blocking, in the previous example.

```
 95 pi-839    [  9]  [01]   168.239972371      ---------->
 96 pi-839    [  9]  [01]   168.239972371                    do_notify_resume() {
 97 pi-839    [  9]  [01]   168.240005839                      softirq_raise: vec=8 [action=HRTIMER]
 98 pi-839    [  9]  [01]   168.240013714                      sched_wakeup: comm=ksoftirqd/1 pid=15  \
                                                              prio=98                                \
                                                              success=1 target_cpu=001
 99 pi-839    [  9]  [01]   168.240037487    + 64.696 us     } do_notify_resume ()
100 pi-839    [  9]  [01]   168.240037487      <----------
101 pi-839    [  9]  [01]   168.240038795      ---------->
102 pi-839    [  9]  [01]   168.240038795                    sys_rt_sigreturn() {
103 pi-839    [  9]  [01]   168.240045205      6.183 us      } sys_rt_sigreturn ()
104 pi-839    [  9]  [01]   168.240045205      <----------
105 pi-839    [  9]  [01]   168.240047616      ---------->
106 pi-839    [  9]  [01]   168.240047616                    sys_pause() {
107 pi-839    [  9]  [01]   168.240047925                      schedule() {
108 pi-839    [  9]  [01]   168.240063374                          sched_switch: prev_comm=pi         \
                                                                  prev_pid=839                       \
                                                                  prev_prio=9 prev_state=S ==>       \
                                                                  next_comm=ksoftirqd/1 next_pid=15  \
                                                                  next_prio=98
```

It is the background of the thesis.

It was done before the official PhD enrolment, but... it is part of the PhD, at least inside our ❤️.

Red Hat

Timing analysis of the PREEMPT RT Linux kernel

Lessons learned

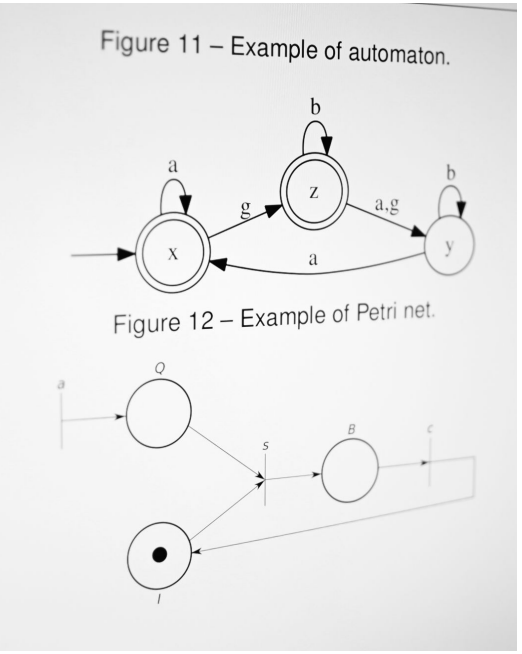
Pros

- Good level of abstraction
- Usage of trace and events
- Lightweight
- The timeline format is intuitive

Cons

- Manual interpretation of the data
- The translation from trace to timeline was informal, and so prone to ambiguosity
- Impossible to verify trace ⇔ interpretation

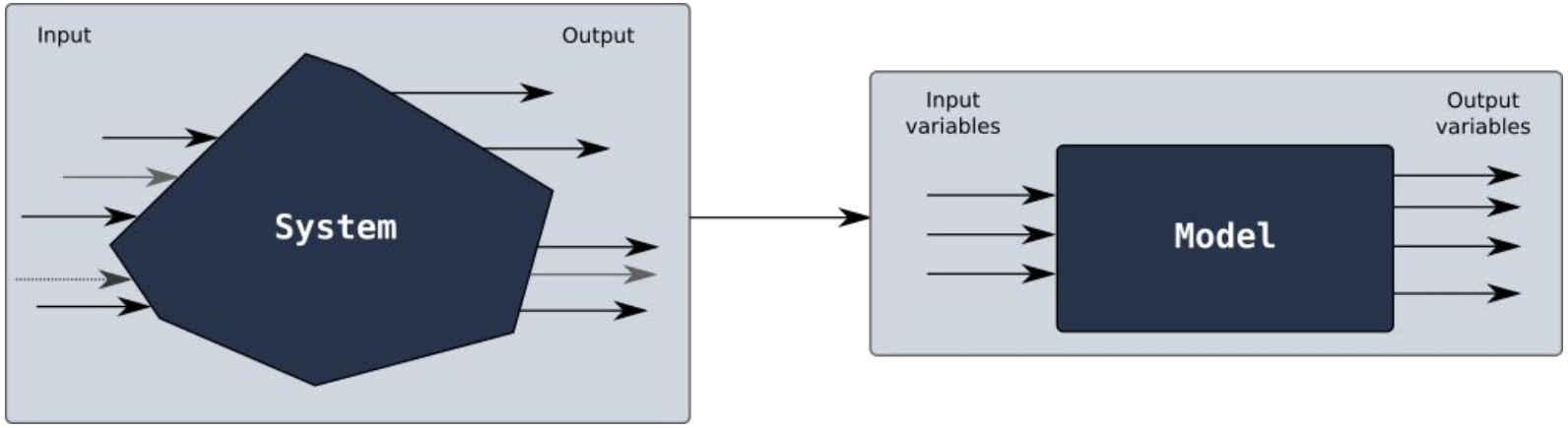
Getting formal: formal methods



- A collection of mathematical techniques to rigorously state the specification of a system
- Useful to demonstrate properties of a system
- Remove the ambiguous nature of natural language
- Enable automatic verification of the system

At this point, Daniel started attending to classes about Formal Methods, Discrete Math, Discrete Events Systems, and Formal Verification.

Getting formal: formal models



A model is an abstraction, removing the unnecessary details, focusing in a specific behavior.

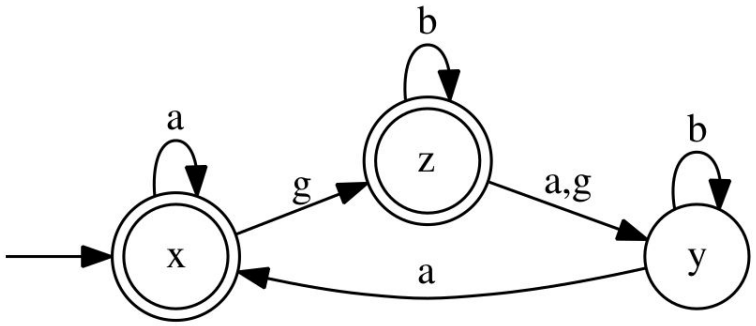
Once a **satisfactory** model is found, it can be used instead of the system.

Red Hat

Automata a formal language

That looked natural for a kernel developer

Graphical representation



Formal definition

$G = \{X, E, F, x_0, X_m\}$, where

X is set of states

E is the finite set of events

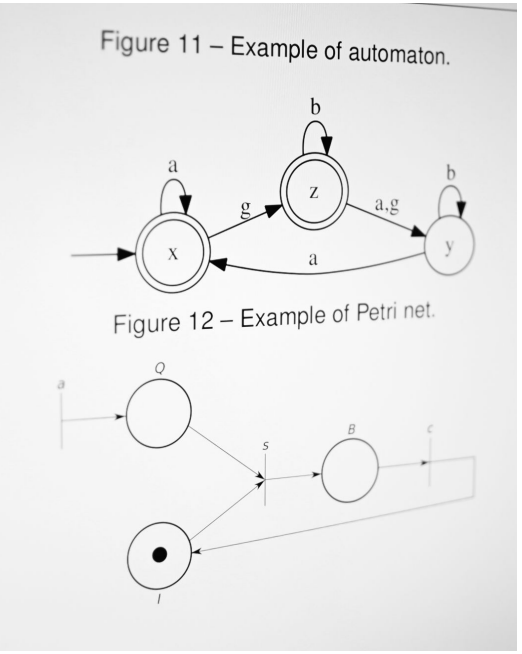
$F : X \times E \rightarrow X$ is the transion function

x_0 is the initial state

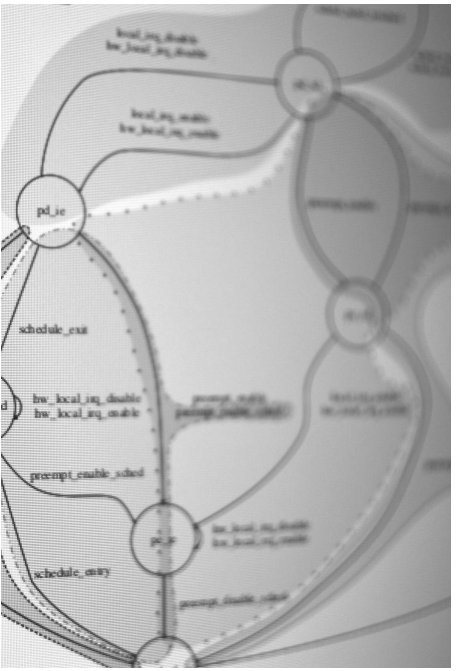
$X_m \subseteq X$ is the set of final states

Modeling strategy

The modular approach



- Instead of modeling the system as a single automaton, the modular approach uses **generators** and **specifications**
  - **Generators**:
    - Independent subsystems models
    - Generates all chain of events (without control)
  - **Specification**:
    - Control/synchronization rules of two or more subsystems
    - Blocks some events
- The parallel composition operation synchronizes the generators and specifications
  - **The result is an automaton with all possible chain of events**

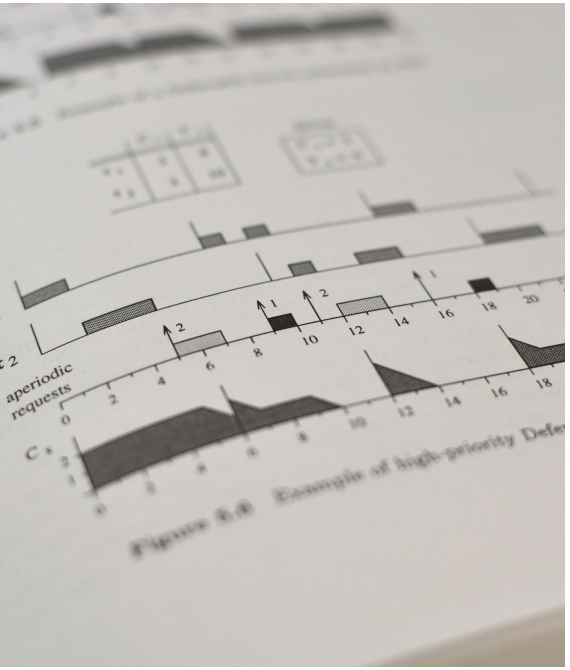


Goals and contributions

Red Hat

Goals of this thesis

Three sub-goals



This thesis proposes the creation of **a formal model of the Linux tasks**, including the synchronization primitives that influence their timing behavior.

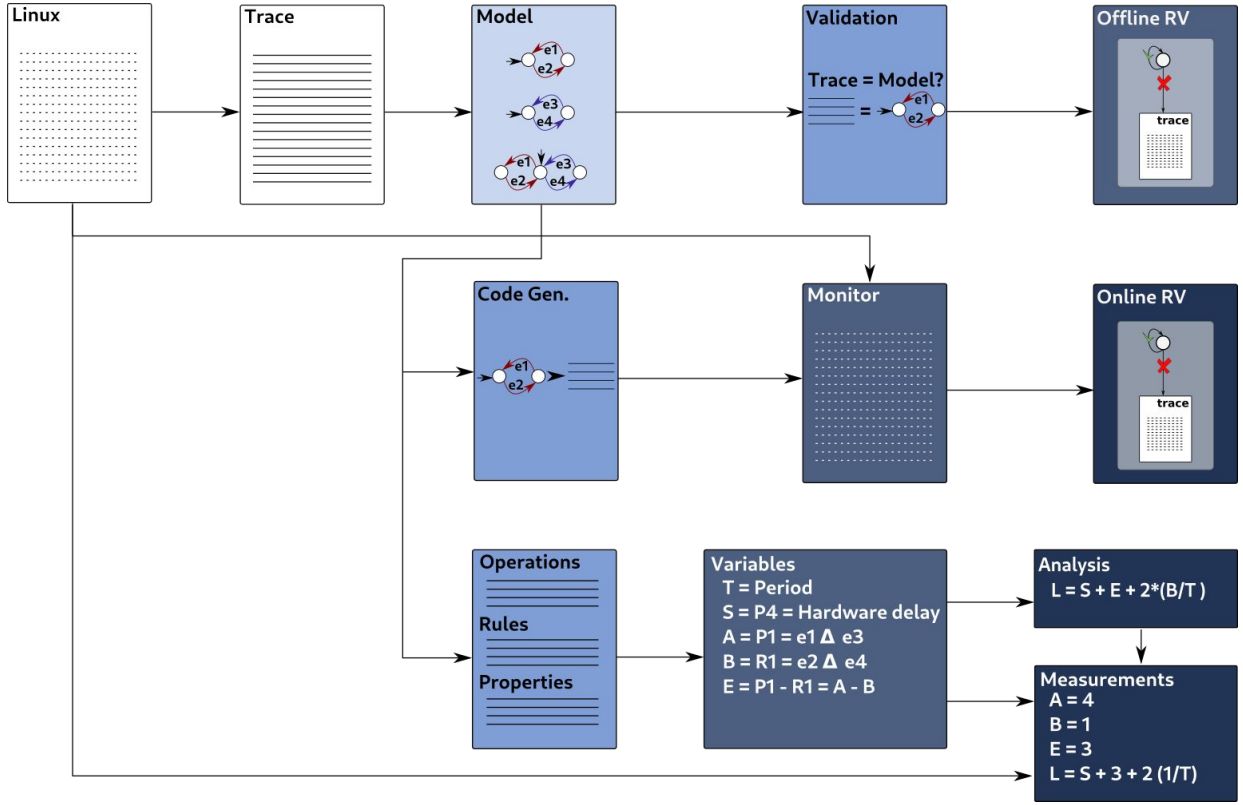
This model should
enable the formal verification of the logical behavior of the system, as well as the **formal analysis of its timing behavior**.

Contributions

Three stages

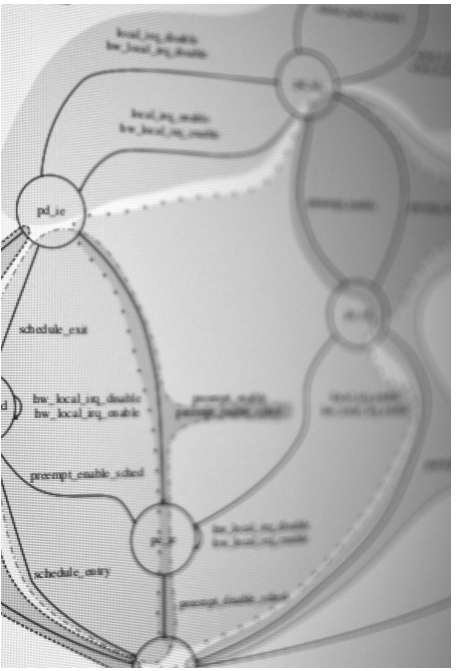
- First stage: the **formal model**
  - The methodology
  - The model
  - *Offline* verification
- Second stage: **efficient runtime verification** of the logical behavior
  - Online runtime verification
  - Auto code generation from models
  - Can be used in production
- Third stage: **analysis and measurements** of the timing behavior
  - Interpretation of the model using academic real-time viewpoint
  - Definition of a safe latency bound
  - Development of a tool to measure the components of the latency bound

Contributions



All the results are available online here:

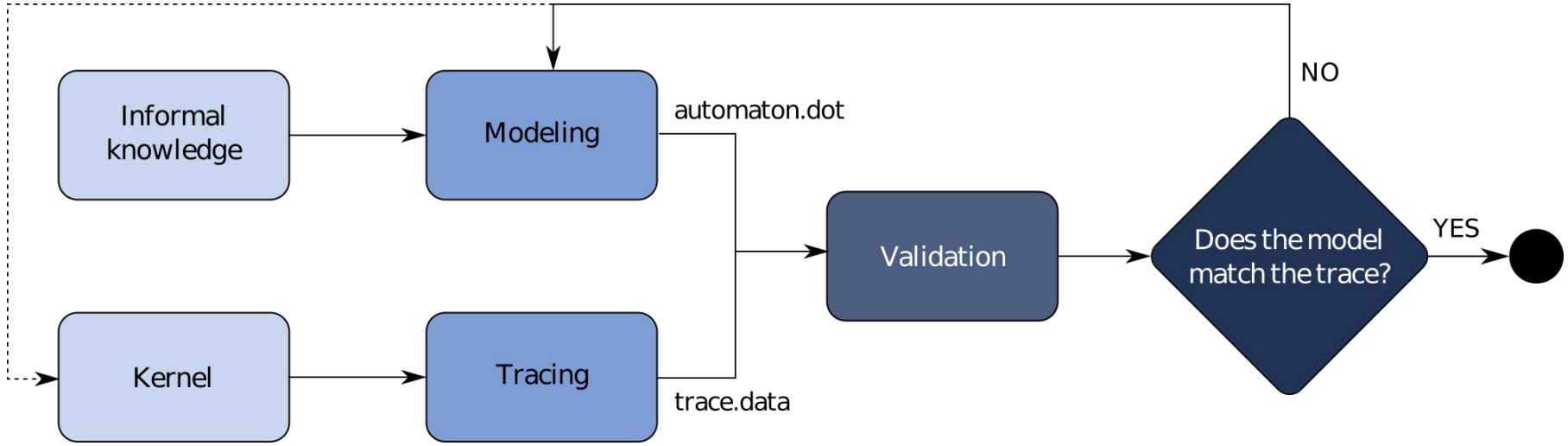
Red Hat



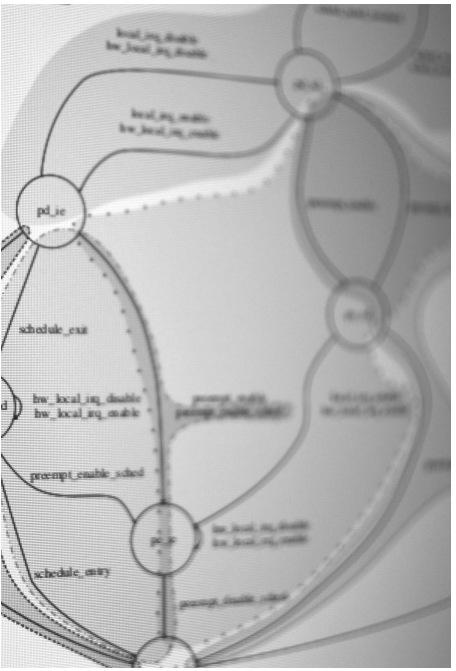
Part I:
The thread model

The PREEMPT_RT Thread model

Approach



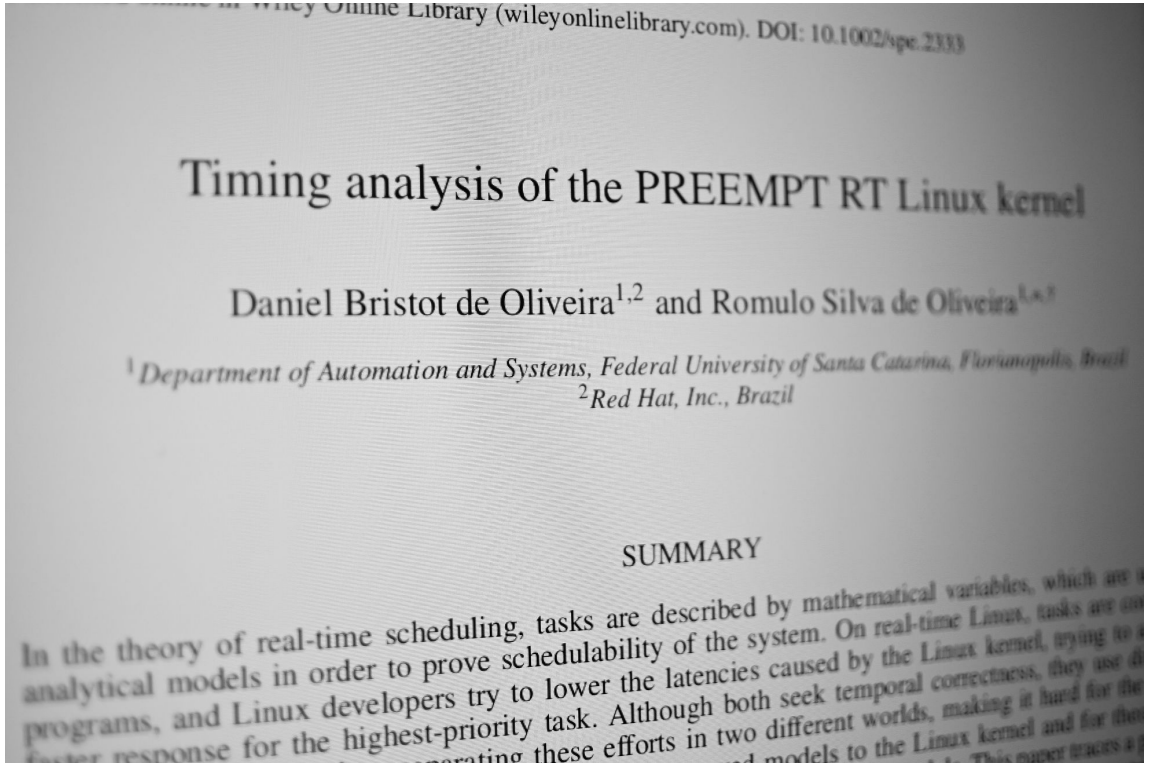
Red Hat



Modeling

Red Hat

Events



Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2333

Timing analysis of the PREEMPT RT Linux kernel

Daniel Bristot de Oliveira[1,2] and Romulo Silva de Oliveira[1,*,†]

[1] *Department of Automation and Systems, Federal University of Santa Catarina, Florianopolis, Brazil*
[2] *Red Hat, Inc., Brazil*

SUMMARY

In the theory of real-time scheduling, tasks are described by mathematical variables, which are
analytical models in order to prove schedulability of the system. On real-time Linux, tasks are
programs, and Linux developers try to lower the latencies caused by the Linux kernel, trying to
response for the highest-priority task. Although both seek temporal correctness, they use
these efforts in two different worlds, making it hard for the
models to the Linux kernel and for the

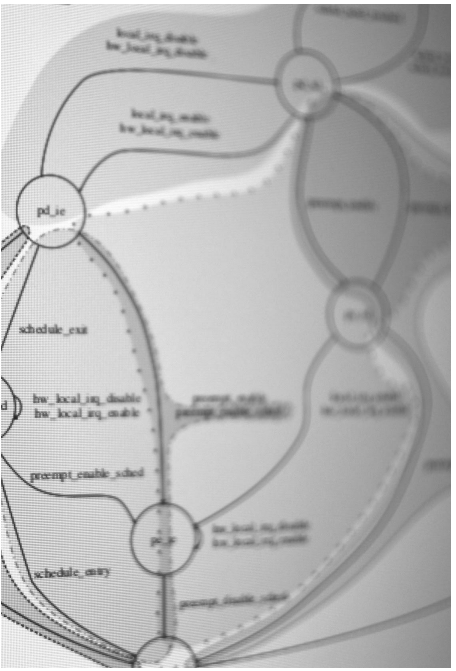
Based in the Timing analysis paper, but improved.

Based also on a daily base work as a kernel developer at red hat.

Red Hat

Interrupt related events

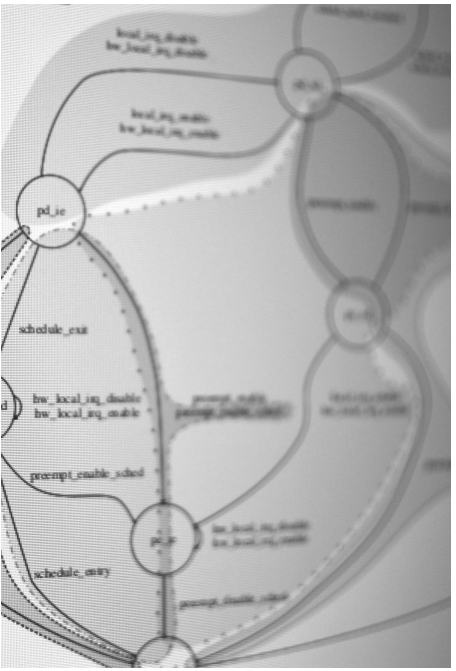
NMI, IRQ, IRQ control



Automaton event	Kernel 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:irq_enable	Unmask IRQs
nmi_entry	**irq_vectors:nmi**	Begin NMI handler
nmi_exit	**irq_vectors:nmi**	Return NMI Handler

Scheduling events

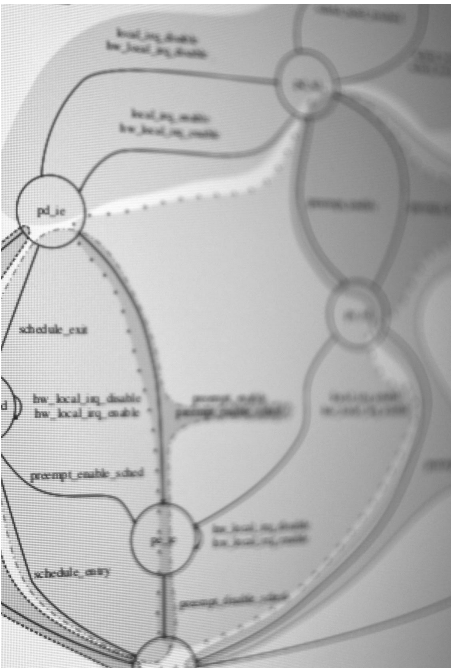
Preemption and scheduling



Automaton event	Kernel 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

Thread states

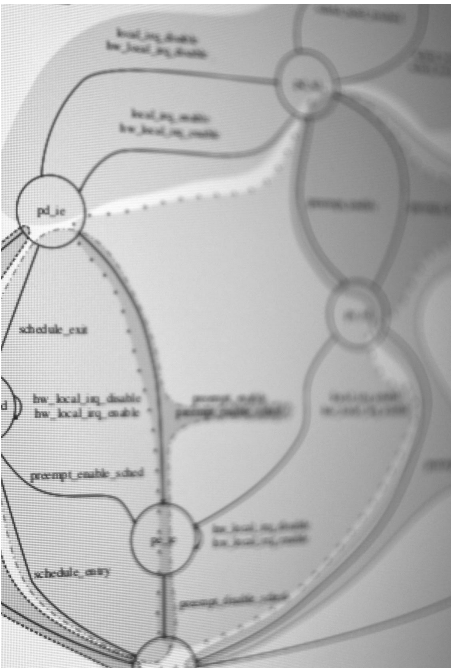
Runnable or not runnable? That is the question!



Automaton event	Kernel event	Description
sched_waking	sched:sched_waking	Activation of a thread
sched_set_state_runnable	**sched:sched_set_state**	Thread is runnable
sched_set_state_sleepable	**sched:sched_set_state**	Thread can go to sleepable

Context switch

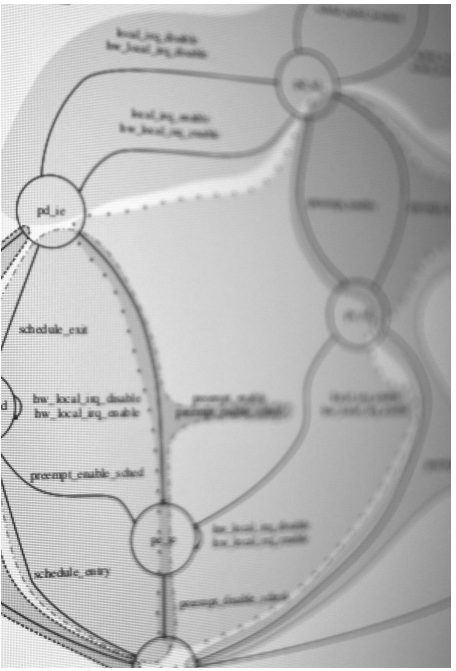
Two “meta” tasks: the one under analysis and all other



Automaton event	Kernel event	Description
sched_switch_in	sched:sched_switch	Switch in of the thread under analysis
sched_switch_suspend	sched:sched_switch	Switch out due to a suspension of the thread under analysis
sched_switch_preempt	sched:sched_switch	Switch out due to a preemption of the thread under analysis
sched_switch_blocking	sched:sched_switch	Switch out due to a blocking of the thread under analysis
sched_switch_in_o	sched:sched_switch	Switch in of another thread
sched_switch_out_o	sched:sched_switch	Switch out of another thread

Mutex

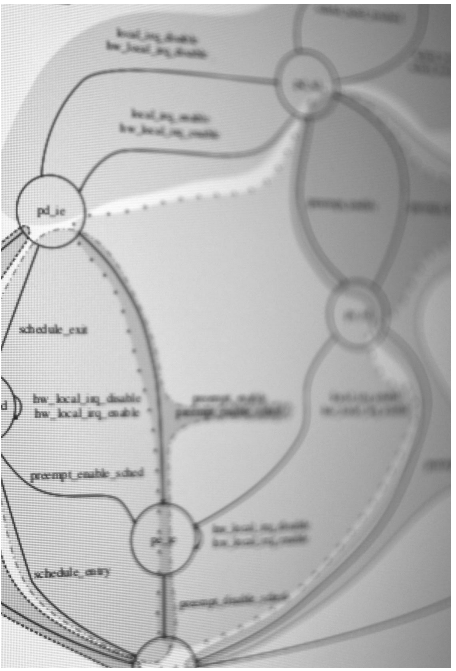
Mutual exclusion



Automaton event	Kernel 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

Read/write lock and semaphore

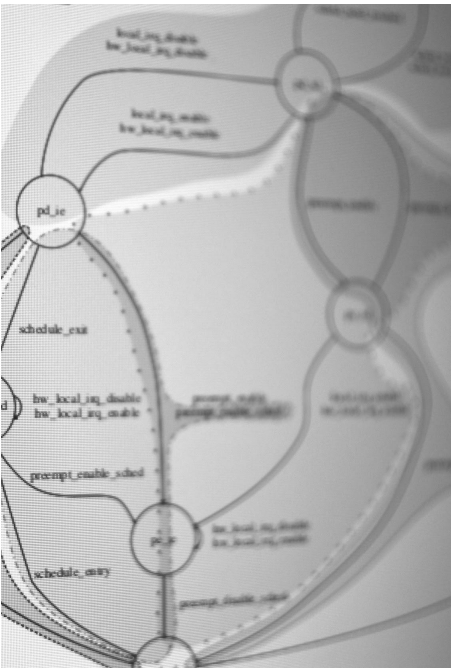
Read side



Automaton event	Kernel event	Description
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**	Abandoned a R/W Lock or Sem as reader

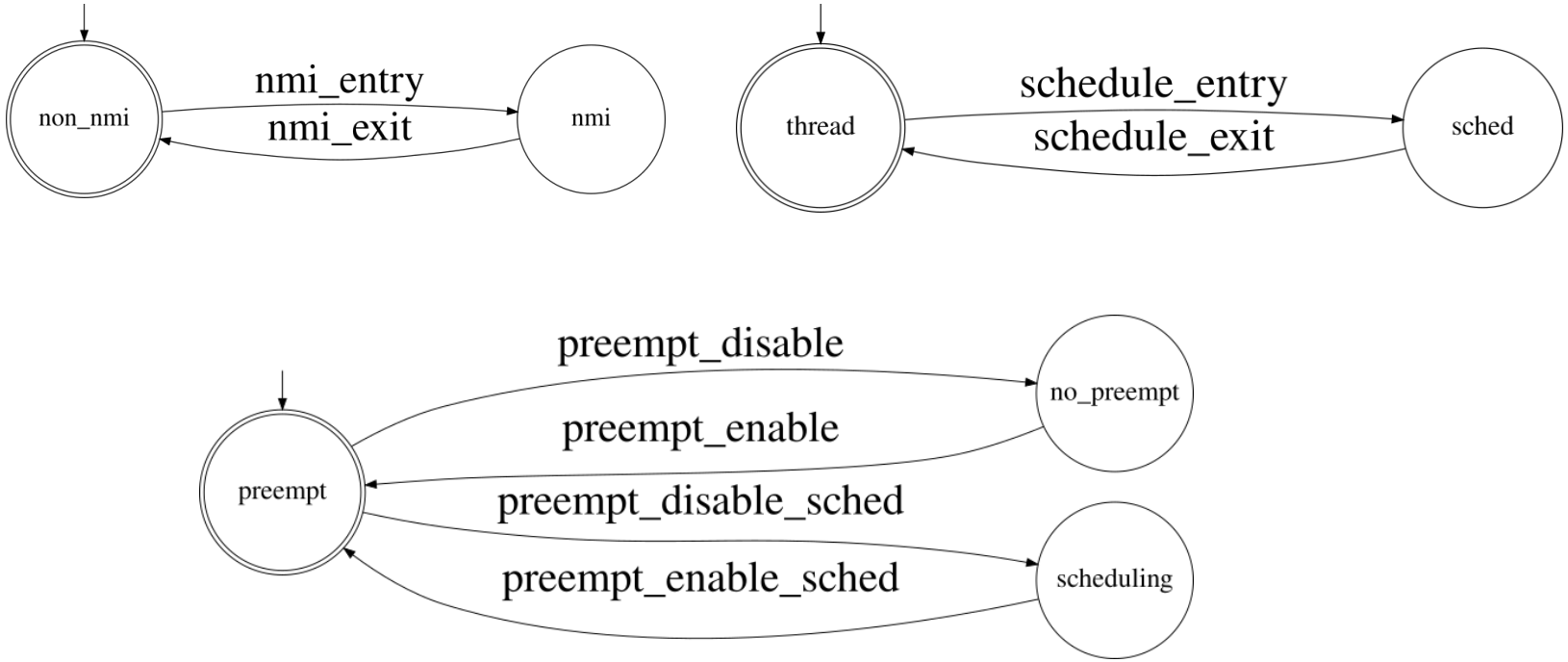
Read/write lock and semaphore

Write side



Automaton event	Kernel event	Description
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

Generators



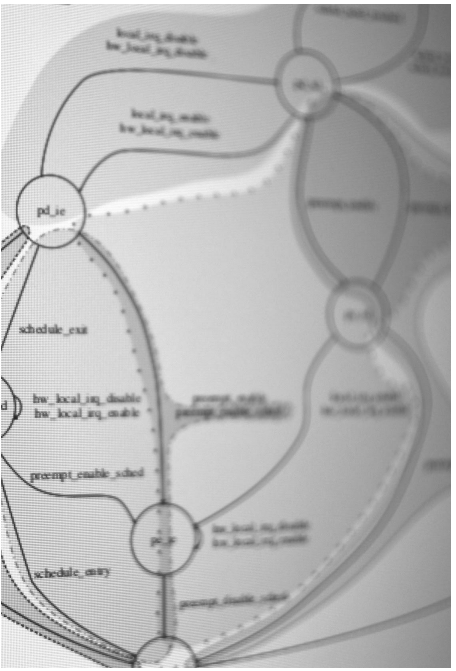
They are mostly basic kernel operations, in the way that developers think about them independently.

They can be specialized, but better not generalize them.

Red Hat

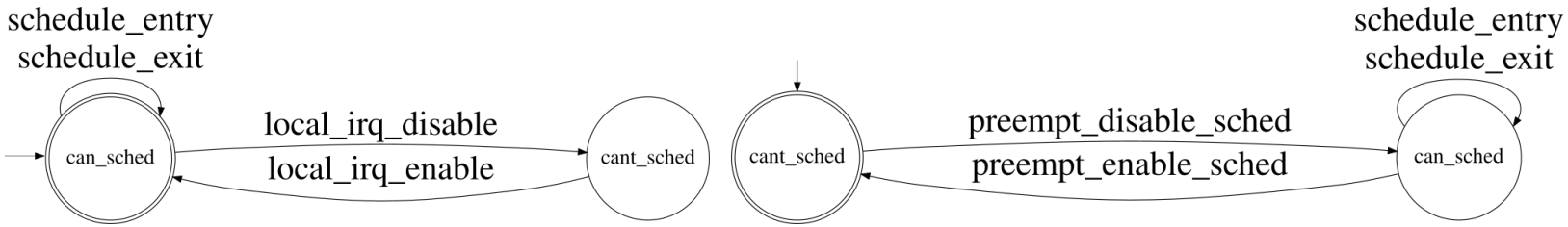
The generators!

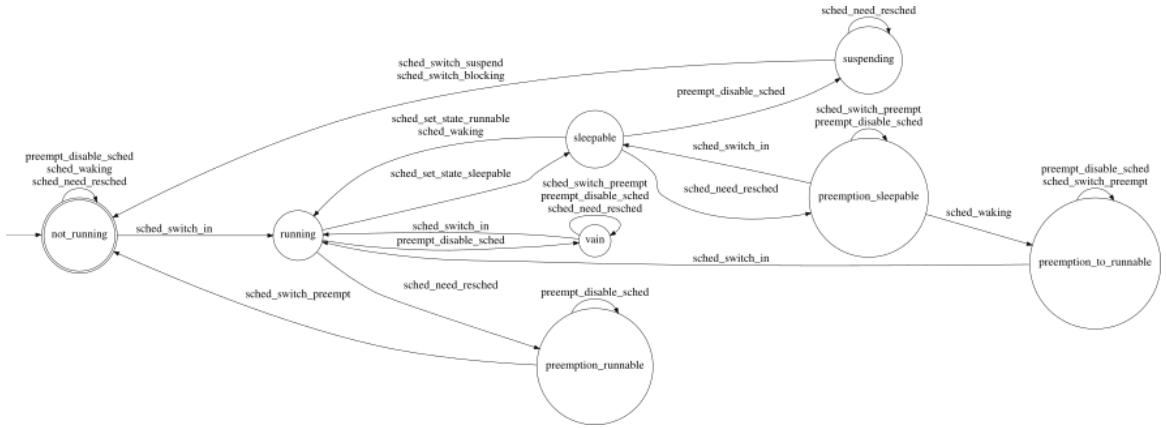
Independent operations



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

Specifications





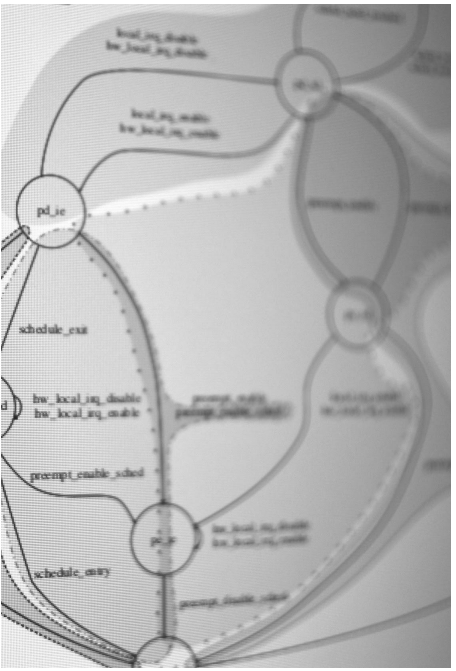
We tried to keep the specifications as simple as possible, trying to model a single behavior per specification.

We also tried to keep a logical interpretation for each specification, like “necessary” and “sufficient” conditions.

Red Hat

Specifications

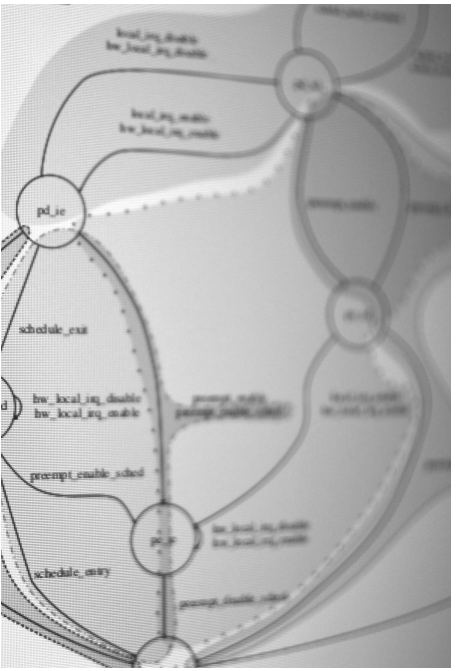
Relation among operations (continue..)



Name	*States*	*Events*	*Transitions*
S01 Sched in after wakeup	**2**	**5**	**6**
S02 Resched and wakeup sufficiency	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

Specifications

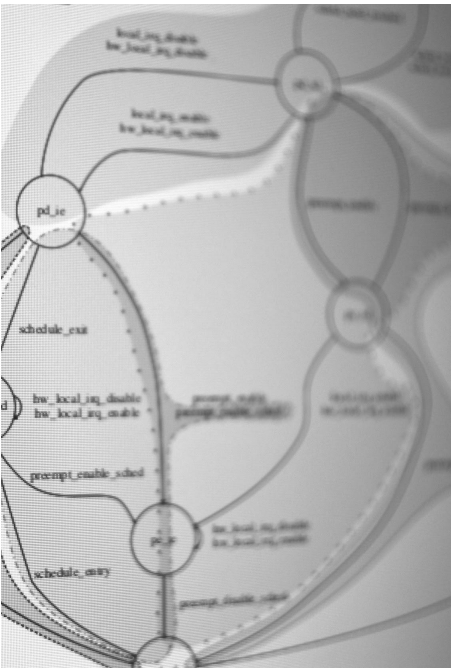
Relation among operations



Name	*States*	*Events*	*Transitions*
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**

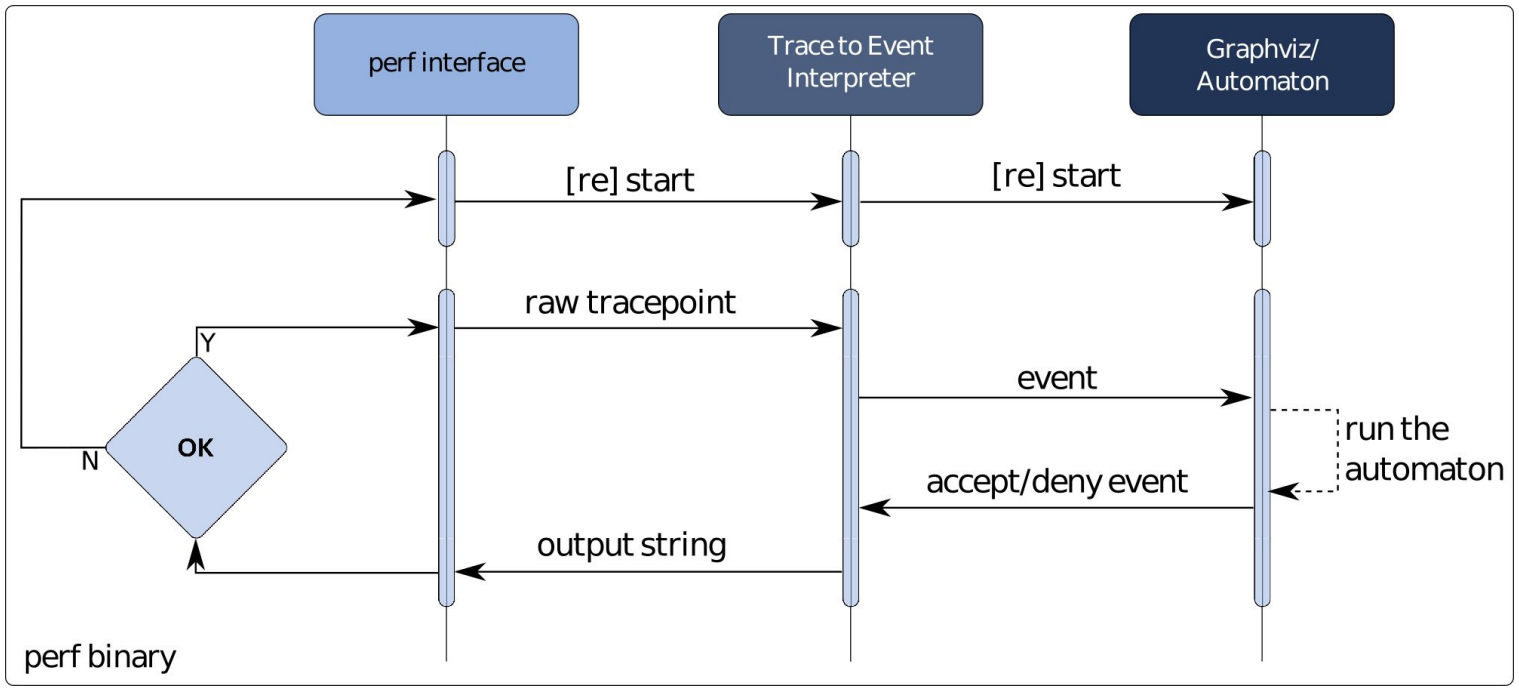
The model!

Composition of generators and specifications



- The final model has:
  - **9017** states
  - **20103** transitions
- It would be impossible to model it directly
- Using the modular approach, the final model is composed of:
  - 34 events
  - 12 generators
  - 33 specifications
    - The most complex module (a specification) has **eight** states!

Verification: perf task_model



The perf task_model extension was developed to do the automatic verification

Two phases: **record** and **report**

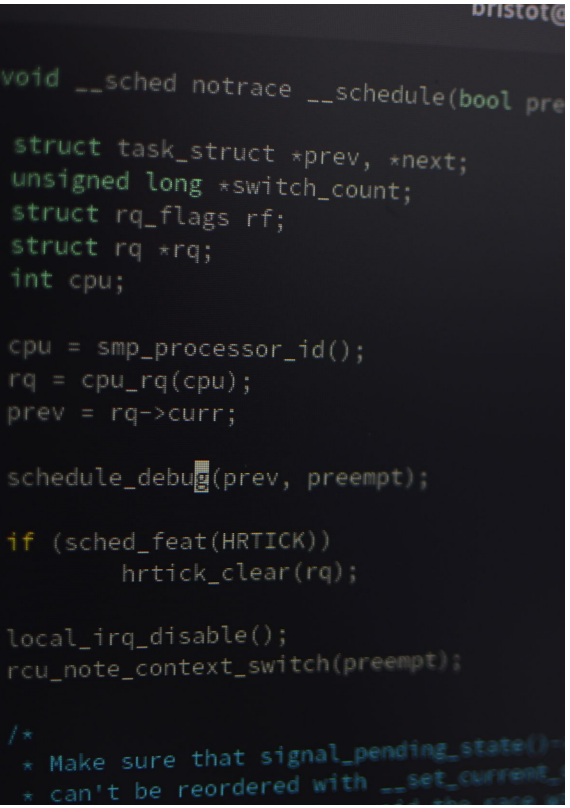
All in user-space

That was a big problem of the "timing analysis" of the previous paper: there was no way to compare the kernel against our reasoning

Red Hat

perf task_model output

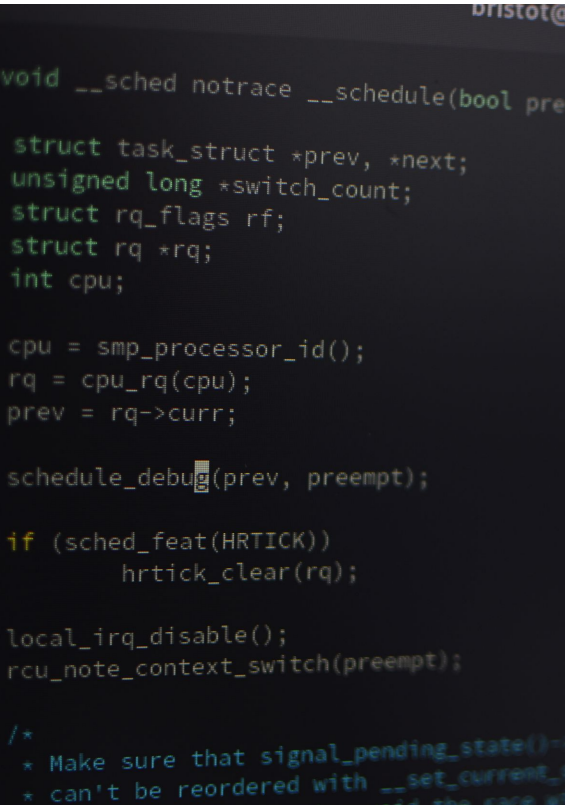
Automatically runs the automaton, based on the kernel trace




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

Runtime verification of the kernel

Even better than we expected



- By modeling the **expected** behavior, we can catch cases in which the **kernel does not behave as expected**
  - We **found three problems** on kernel
    - One **unexpected call to schedule**()
      - Schedule called in vain
        - Resulted in a kernel patch
    - **Locking correctness**
      - A scheduling while in atomic in the single-core case
    - **Perf & Ftrace losing events**
      - A problem in the trace recursion control

A Thread Synchronization Model for the PREEMPT_RT Linux Kernel

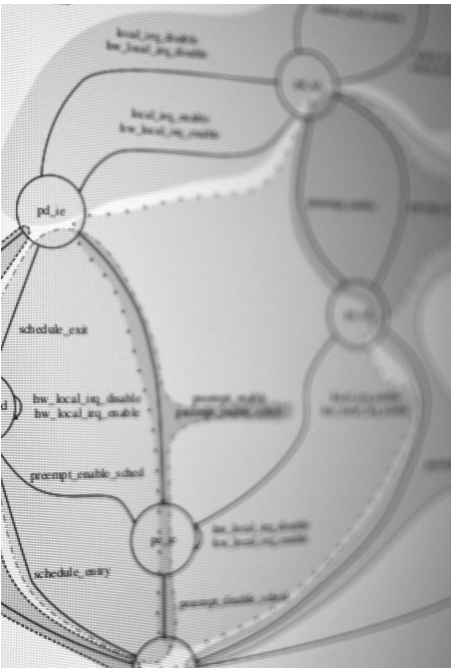
Lessons learned

Pros

- **Formal model**
- Automatic cross-verification
- Automata is simple enough to avoid modeling problems
  - We often faced state explosion but made it
- The format was well received by Linux kernel community

Cons

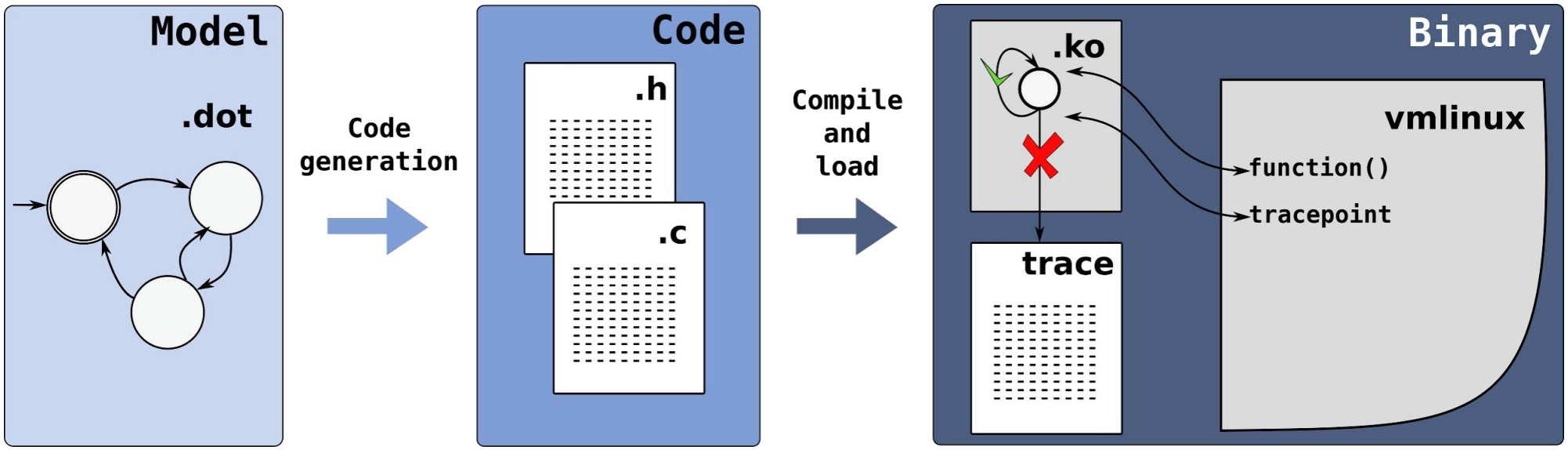
- The verification uses too much resources
  - GBs of data per sec
- Offline
  - No actions can be taken during a problem
- It shows the bound of the scheduling latency, but it is only logical and **too formal**!



Part II:
Verifying the logical behavior

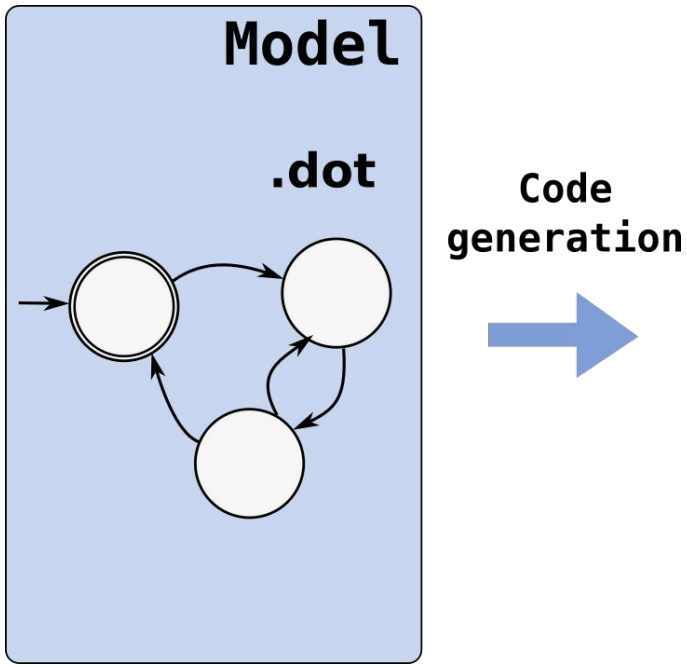
Efficient runtime verification for the Linux Kernel

Approach



Code generation

dot2c

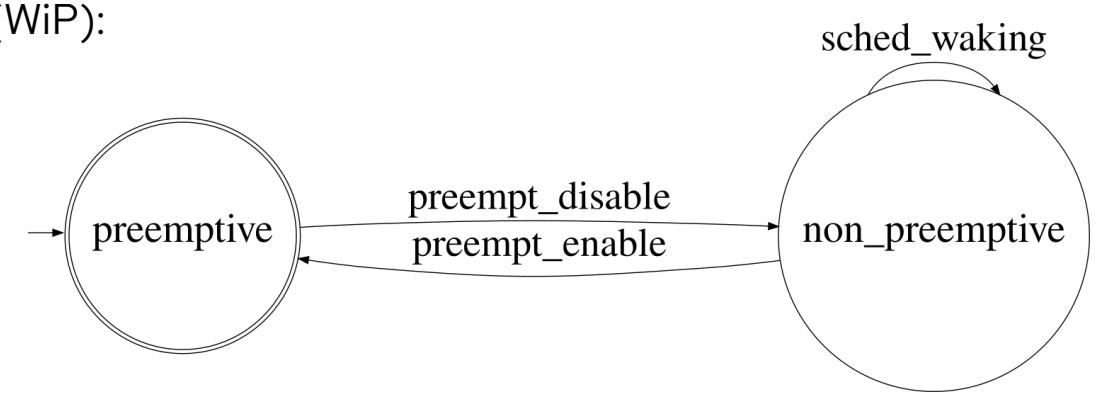


- We develop the **dot2c** tool to translate the model into code
  - It is unpractical to thing about coding a model with 20k+ states
- It is a python program that has one input:
  - An automaton model in the .dot format
    - It is an open format (graphviz)
- **Supremica tool** exports models with this format

Code generation

WiP Example

Wakeup in preemptive model (WiP):

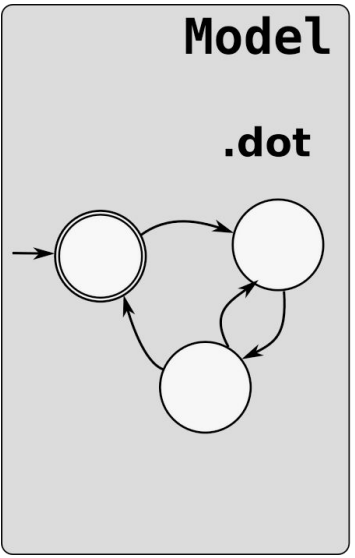


Code generation:

```
[bristot@t460s dot2c]$ ./dot2c wakeup_in_preemptive.dot
…
```

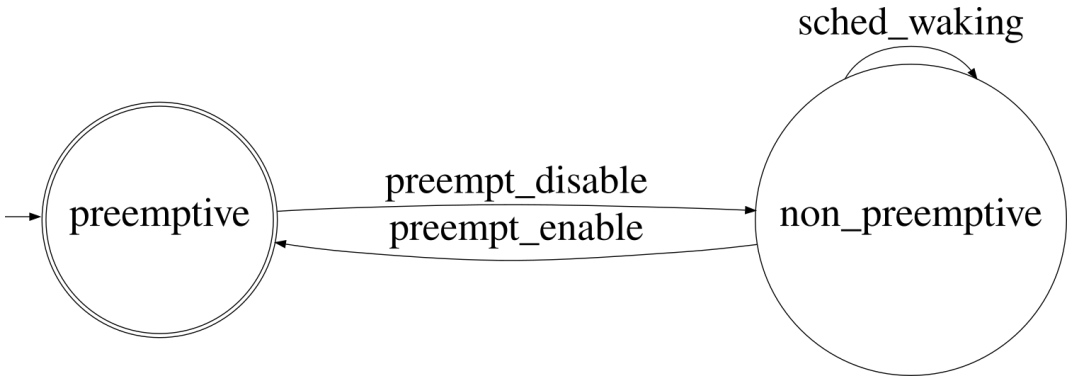
Code generation

Automaton in C



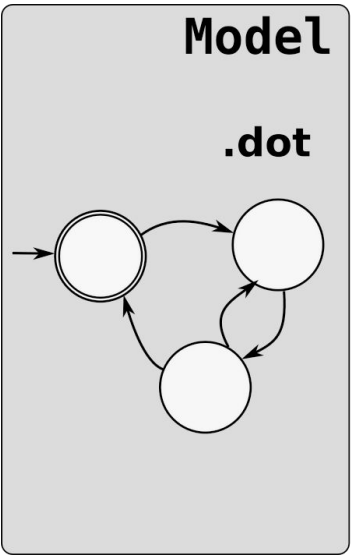
Code generation

```
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];
};
```



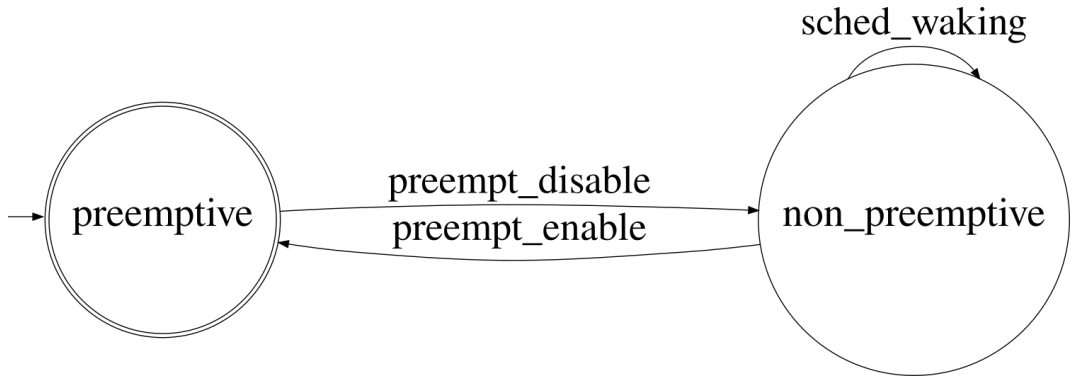
Code generation

Automaton in C



Code generation

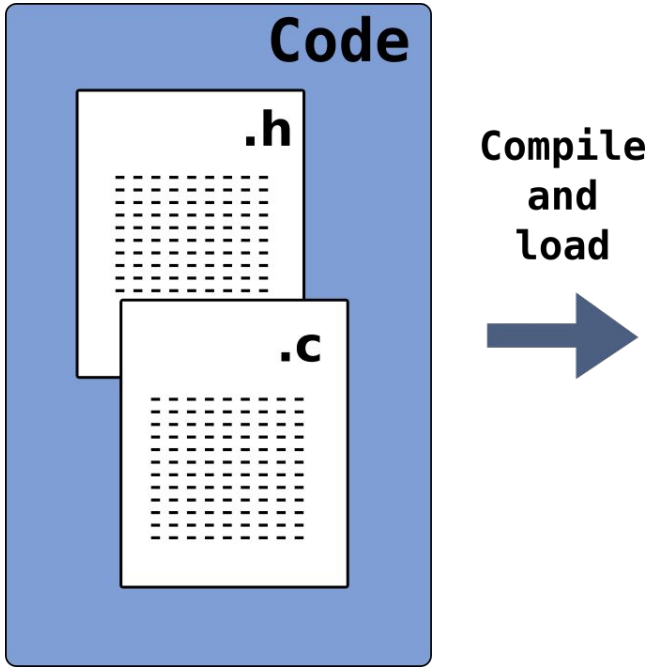
```
enum states {
        preemptive = 0,
        non_preemptive,
        state_max
};

enum events {
        preempt_disable = 0,
        preempt_enable,
        sched_waking,
        event_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 }
};
```



Red Hat

Monitor

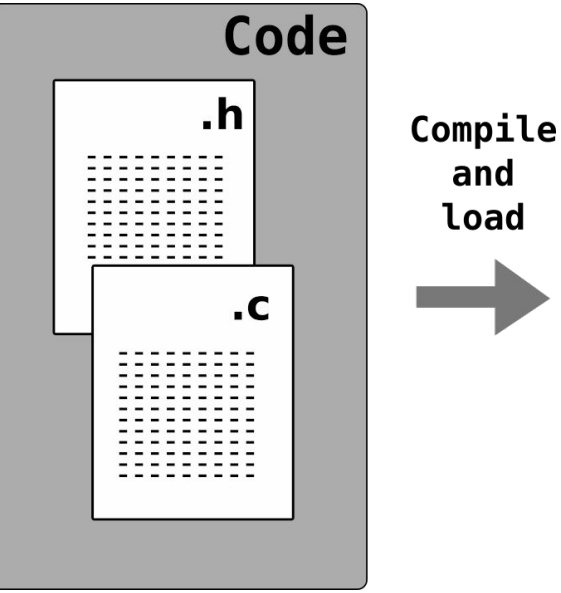
dot2c



- Interprets the kernel events, using the model
- Built as a kernel module
  - Processing the events synchronously with the kernel execution
- Set up instrumentation:
  - Hooks to kernel events, e.g., tracepoints, functions,...
  - Waits for the initial condition
- Verifies if a given kernel event is accepted by the model
  - If an error occurs, actions can be taken in the current state of the system
    - Stacktraces
    - Print variables
    - Save a memory dump...

Monitor

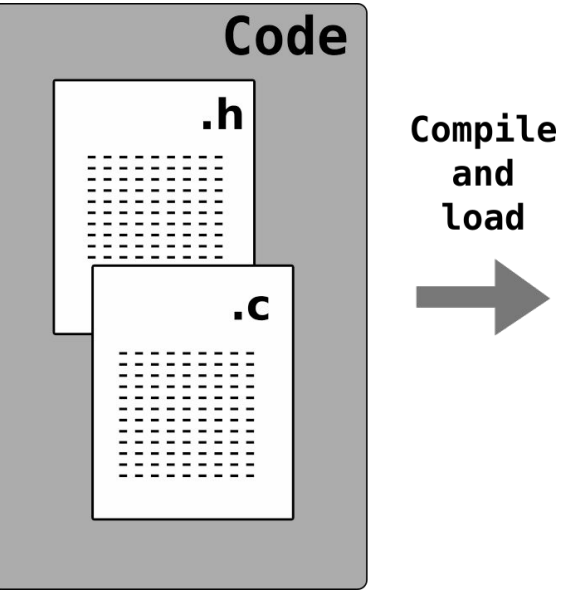
Main function: process event




```
char process_event(struct verification *ver, enum events event)
{
        int curr_state = get_curr_state(ver);
        int next_state = get_next_state(ver, curr_state, event);

        if (next_state != NULL) {
                set_curr_state(ver, next_state);

                debug("%s -> %s = %s %s\n",
                        get_state_name(ver, curr_state),
                        get_event_name(ver, event),
                        get_state_name(ver, next_state),
                        next_state ? "" : "safe!");

                return true;
        }

        error("event %s not expected in the state %s\n",
                get_event_name(ver, event),
                get_state_name(ver, curr_state));

        stack(0);

        return false;
}
```

Monitor

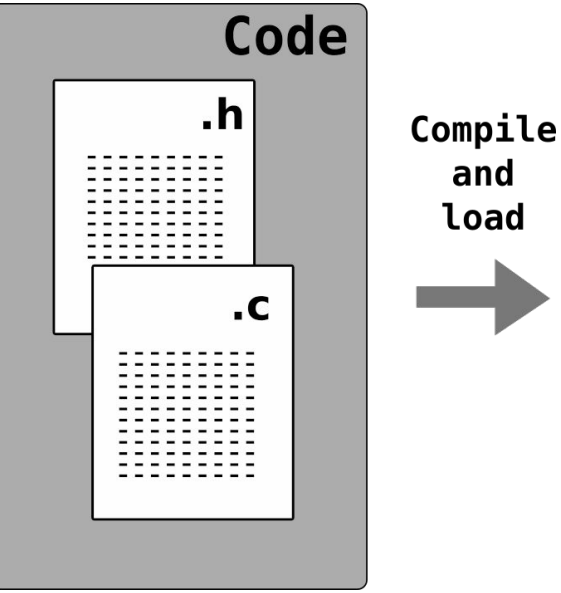
Main function: in details




```
char *get_state_name(struct verification *ver, enum states state) {
        return ver->aut->state_names[state];
}

char *get_event_name(struct verification *ver, enum events event) {
        return ver->aut->event_names[event];
}

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

char get_curr_state(struct verification *ver) {
        return ver->curr_state;

}

void set_curr_state(struct verification *ver, enum states state) {
        ver->curr_state = state;
}
```

Monitor

Main function: in details



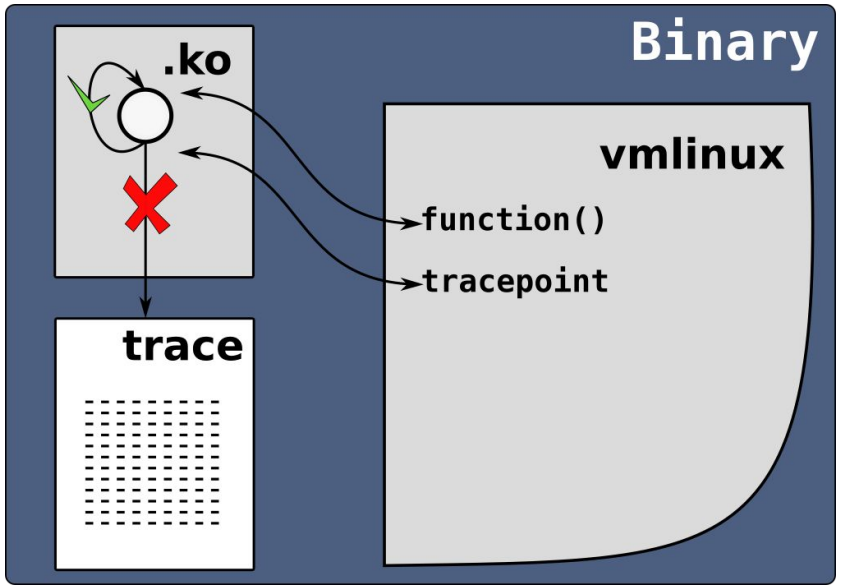

```
char *get_state_name(struct verification *ver, enum states state) {
        return ver->aut->state_names[state];
}

char *get_event_name(struct verification *ver, enum events event) {
        return ver->aut->event_names[event];
}

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

char get_curr_state(struct verification *ver) {
        return ver->curr_state;

}

void set_curr_state(struct verification *ver, enum states state) {
        ver->curr_state = state;
}
```

All operations are O(1)!

Only one variable to keep the state!

Instrumentation

Running the verification

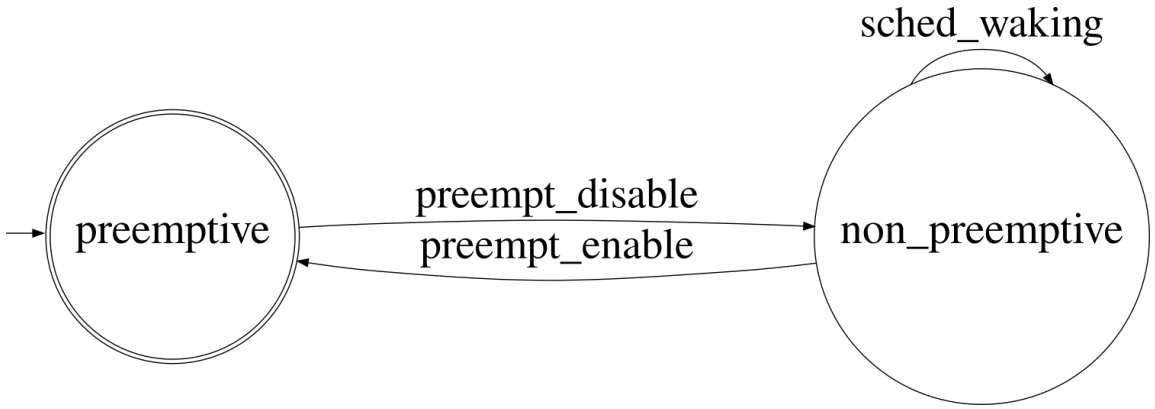


- Kernel module is loaded to a running kernel
  - While no problem is found:
    - Either print the execution of all events in the trace buffer
    - Or run silently
- If an unexpected transitions is found:
  - Print the error on trace buffer
  - Take any action

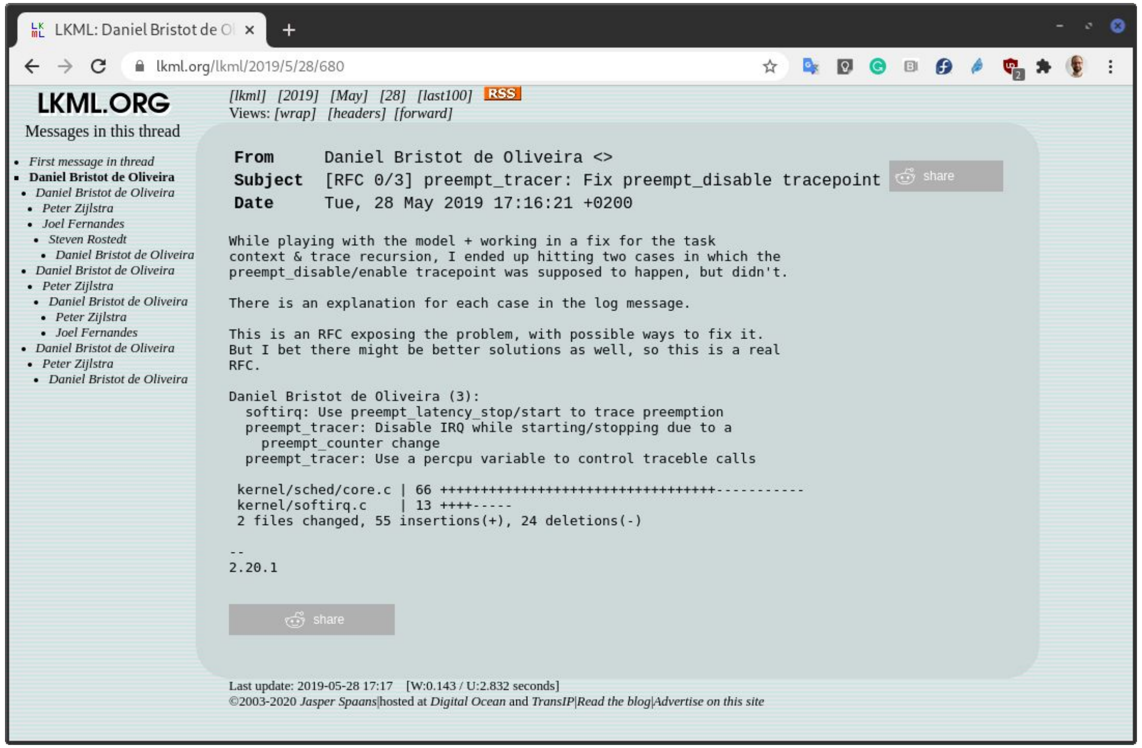
Error output

A real one

```
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
```



Kernel bug report

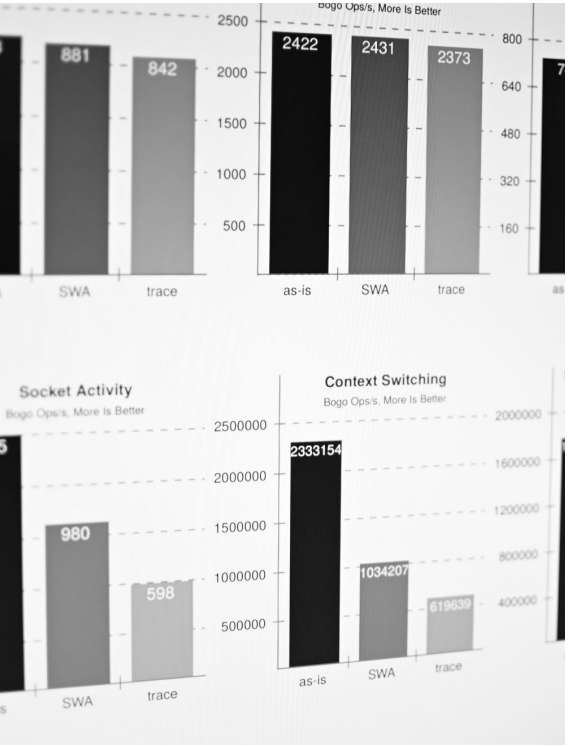


A problem with tracing subsystem was reported using this model's module.

Some preempt_disable/enable events missing.

Problem was reported and discussed.

Red Hat

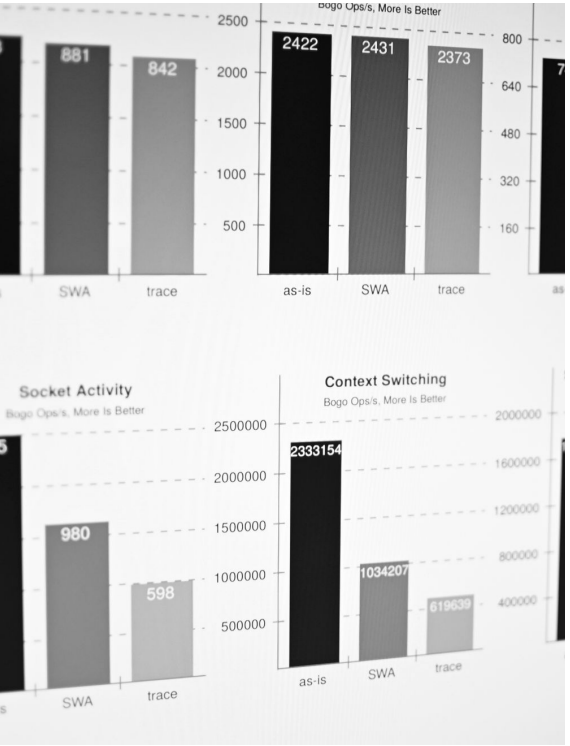


Performance evaluation

Red Hat

Performance evaluation

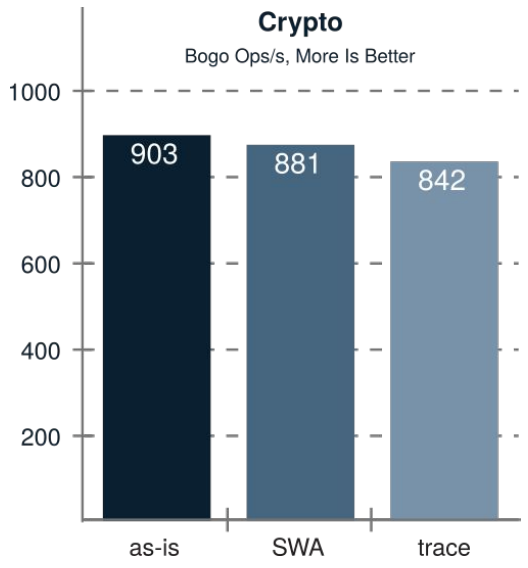
Setup

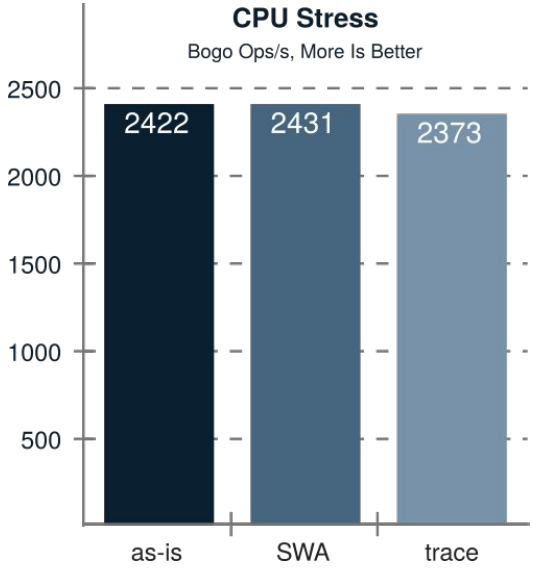


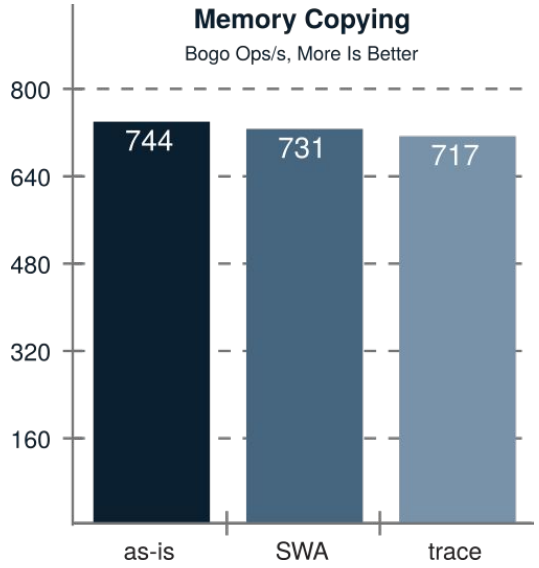
- Two benchmarks
  - Throughput using the Phoronix Test Suite
    - Low kernel activity
    - High kernel activity
  - Scheduling latency
    - Cyclictest
- Base of comparison
  - **as-is**: the system without any verification or trace
  - ***model***: a sample model
  - **trace**: tracing (ftrace) the same events used in the verification
    - **Only trace**! No collection or interpretation

Performance evaluation

High kernel activation (SWA monitor)



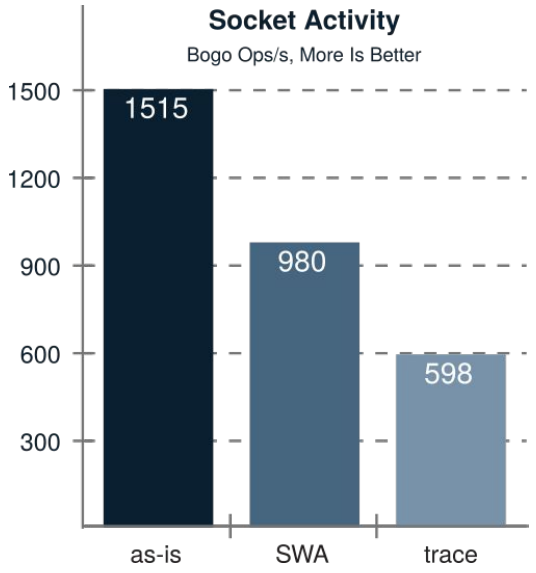


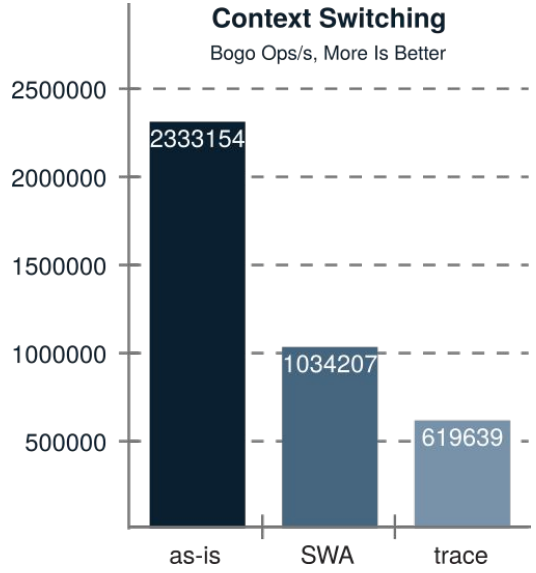


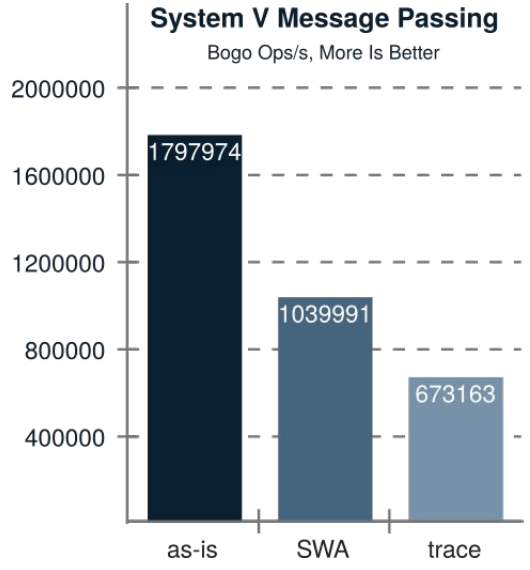
Red Hat

Performance evaluation

Low kernel activation (SWA monitor)



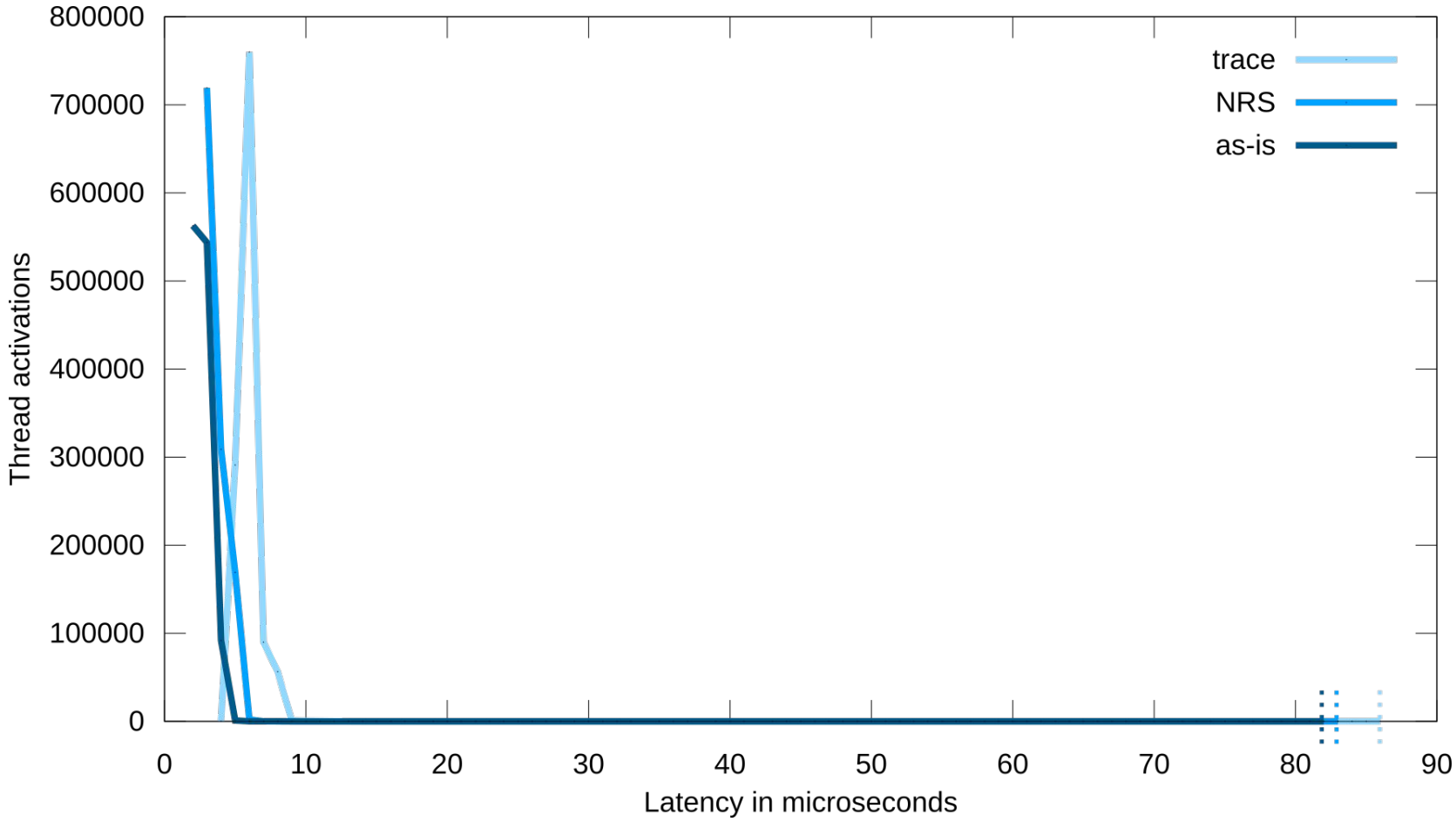




Red Hat

Performance evaluation

Scheduling latency experiment (NRS monitor)



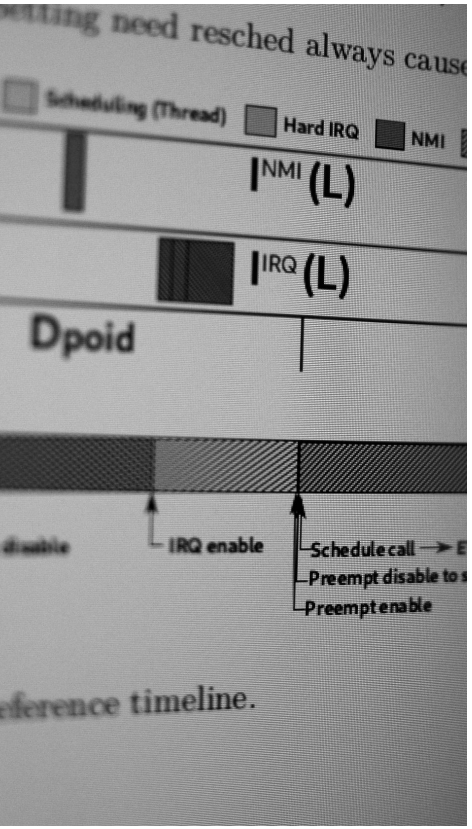
Red Hat

Remarks

- Trace is enable in production systems
  - So this method can be used on production as well
- This is useful mainly for debugging problems:
  - Model the expected behavior
  - Wait for an unexpected event to happen
- We already have content for a journal extension
- There is the interest of other working groups on it
  - Mainly for safety-critical systems and CI
  - We are trying to model other subsystem
  - I am also working with other formalism

The experiments and proof of concept code are available here:

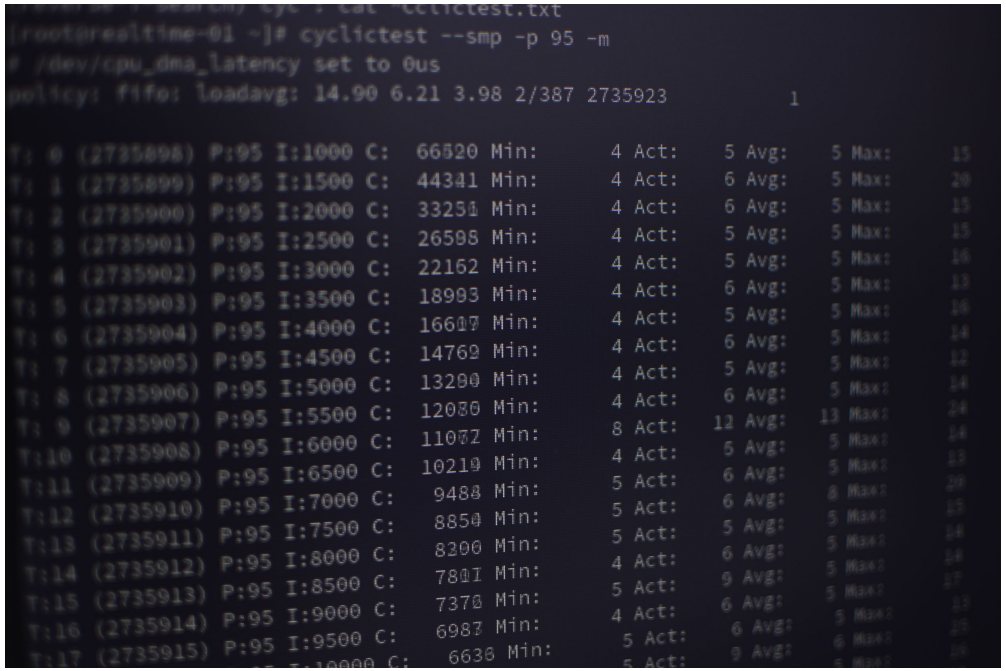
Red Hat



Part III:
Timing behavior analysis

Real-Time Linux vs Real-Time theory

Linux approach

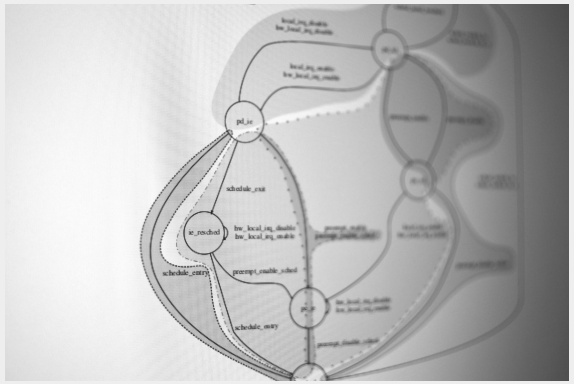


- Linux was adapted to become a RTOS
- PREEMPT_RT: *De facto* standard
- Evaluated (mainly) with cyclictest
- Cyclictest:
  - Practical: lightweight and out-of-the-box
  - It is a “black-box” test
  - No demonstration
  - Does not provide evidence of “root-cause”

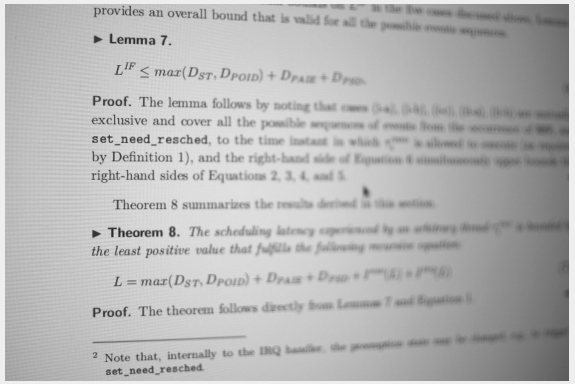
Demystifying the Real-Time Linux Scheduling Latency

Approach

Formal specification



Scheduling latency bound



provides an overall bound that is valid for all the possible events sequences.

► **Lemma 7.**

$$L^{IF} \leq max(D_{ST}, D_{POID}) + D_{PAIR} + D_{PSID}$$

Proof. The lemma follows by noting that cases (1-a), (1-b), (1c), (1-ca), (1-cb) are mutually exclusive and cover all the possible sequences of events from the occurrence of IRQ to `set_need_resched`, to the time instant in which γ_c^{next} is allowed to execute (as required by Definition 1), and the right-hand side of Equation 6 simultaneously upper bounds the right-hand sides of Equations 2, 3, 4, and 5.

Theorem 8 summarizes the results derived in this section.

► **Theorem 8.** *The scheduling latency experienced by an arbitrary thread γ_c^{next} is bounded by the least positive value that fulfills the following recursive equation:*

$$L = max(D_{ST}, D_{POID}) + D_{PAIR} + D_{PSID} + I^{pre}(L) + I^{pre}(L)$$

Proof. The theorem follows directly from Lemmas 7 and Equation 5.

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

Measurement and analysis

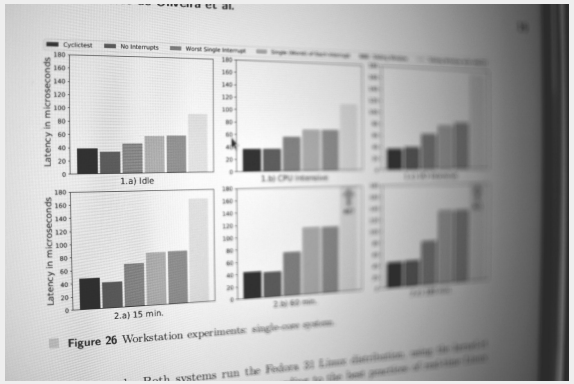


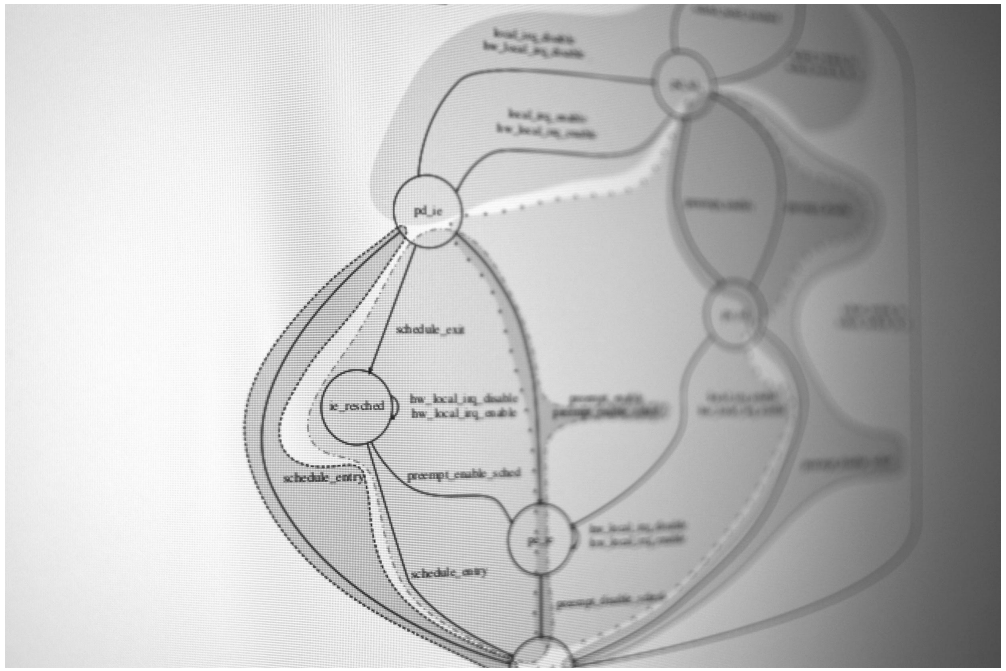
Figure 26 Workstation experiments: single-core systems

Both systems run the Fedora 31 Linux distribution, setup to favor the best practices of system tuning

Red Hat

From formal specification to synchronization rules

Formally backed natural language arguments



- Generators
  - Translated into a **set of operations**
- Specifications
  - Translated into a **set of** synchronization **rules**

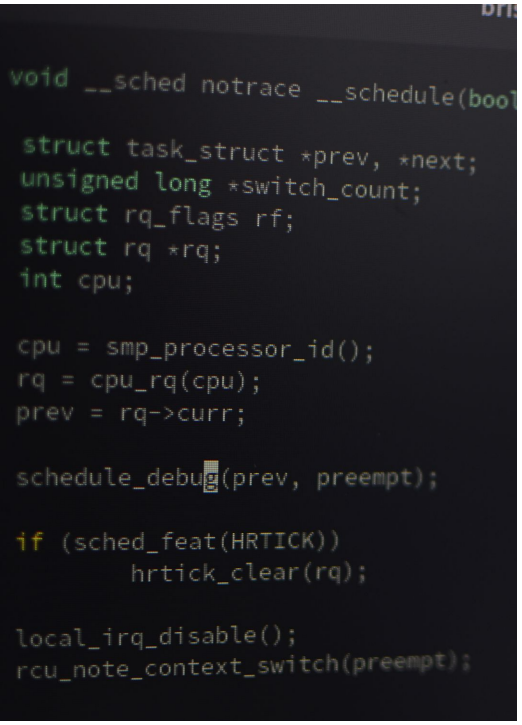
Scheduling latency definition

The **scheduling latency** experienced by an arbitrary thread T is

- the **longest time** elapsed **between** the *time A* in which any job of T becomes **ready and with the highest priority**
- and the *time F* in which the scheduler returns and allows T to **execute its code**

From the first necessary condition to *set need resched*, to the the last action after the scheduling, which is *enabling preemption* after the return from __schedule().

Interference and blocking



```
void __sched notrace __schedule(bool

 struct task_struct *prev, *next;
 unsigned long *switch_count;
 struct rq_flags rf;
 struct rq *rq;
 int cpu;

 cpu = smp_processor_id();
 rq = cpu_rq(cpu);
 prev = rq->curr;

 schedule_debug(prev, preempt);

 if (sched_feat(HRTICK))
         hrtick_clear(rq);

 local_irq_disable();
 rcu_note_context_switch(preempt);
```

The **scheduling latency** is caused by

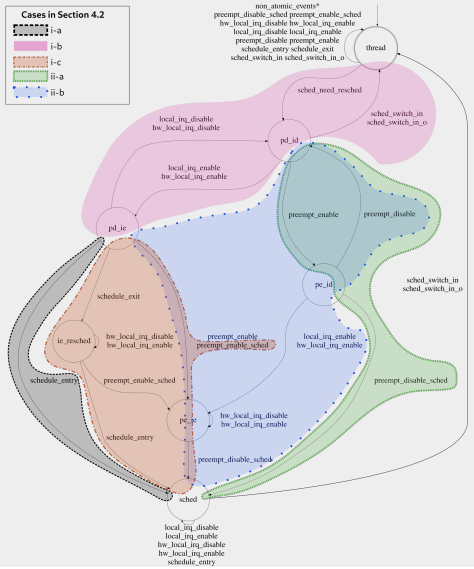
- **Blocking** from the current (and so lower) priority thread
  - Including scheduling
- **Interference** from IRQs and NMI

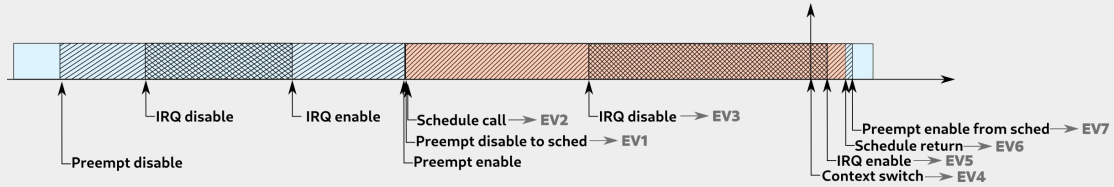
The scheduling latency in this paper refers to the delay between the notification of a new highest priority thread, to point in which this thread starts running its own code.

The highest priority thread can belong to any scheduler: the analysis is scheduler independent.

Blocking bound

From the specification that bounds the block to a timeline

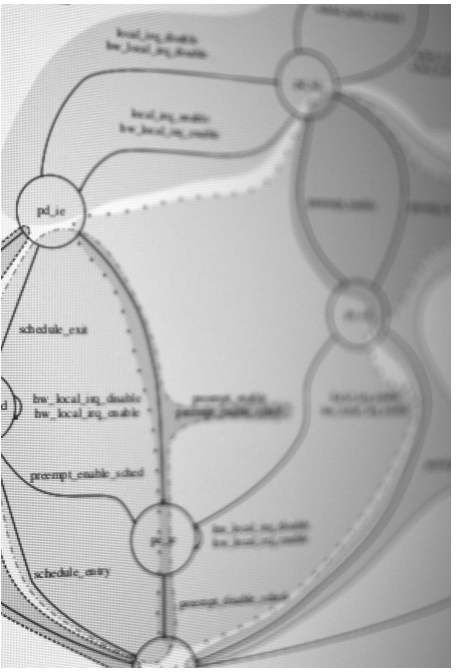


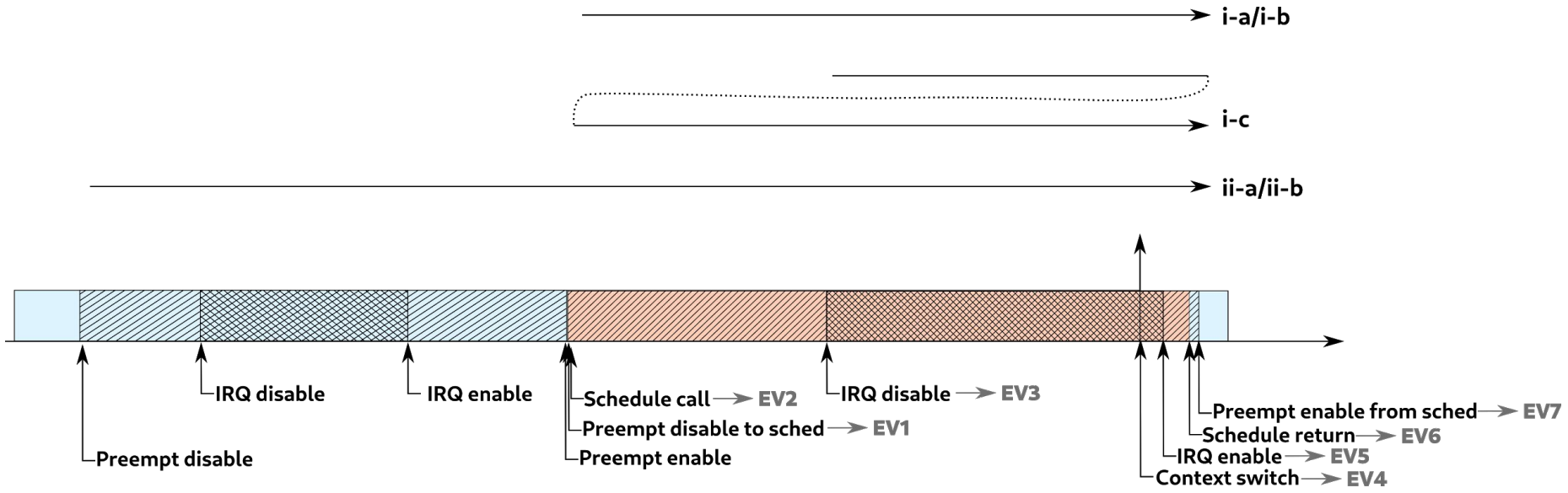


Red Hat

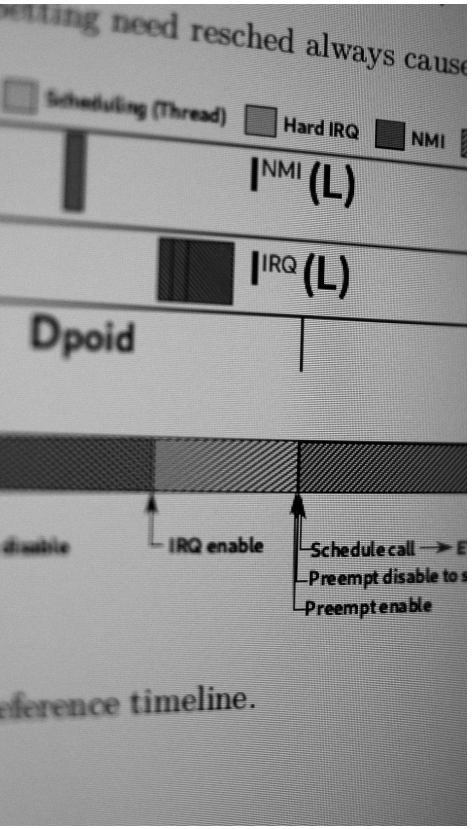
Timeline and cases

All possible cases





Red Hat



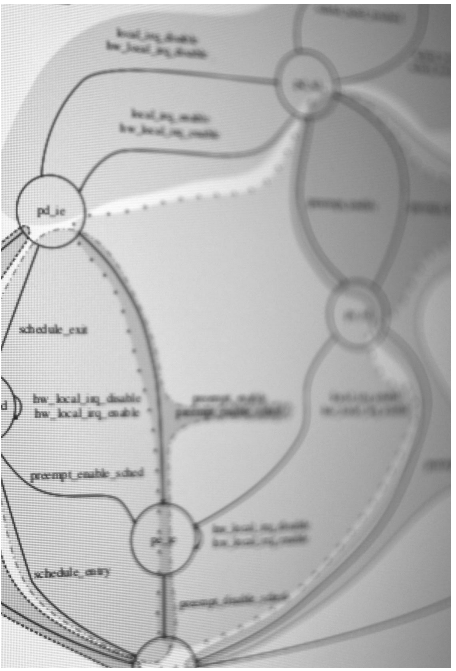
Blocking variables

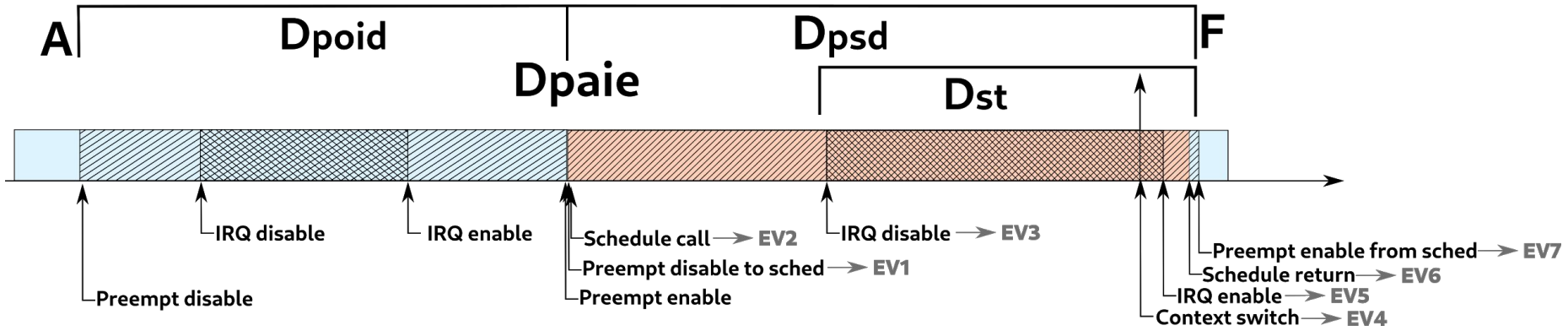
- **DPOID**: preemption or interrupts disabled to postpone the scheduler
- **DPAIE**: preemption and interrupts enabled, as a transient state from **poid** to **psd**; when scheduling a new highest priority thread
- **DPSD**: preemption disable to schedule
- **DST**: delay caused by the scheduling tail; the "non return" point in which a new arrived task will have to wait for the current scheduling operation to finish before scheduling

In the model, the preemption control is specialized into two different operations: to *postpone the scheduler* (the most known behavior) or to *protect the execution of the* __*schedule()* function from recursion.

Timeline and cases

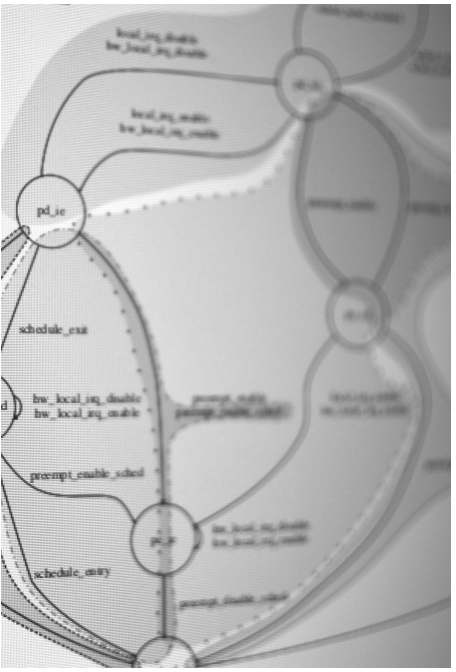
Variables in the the timeline

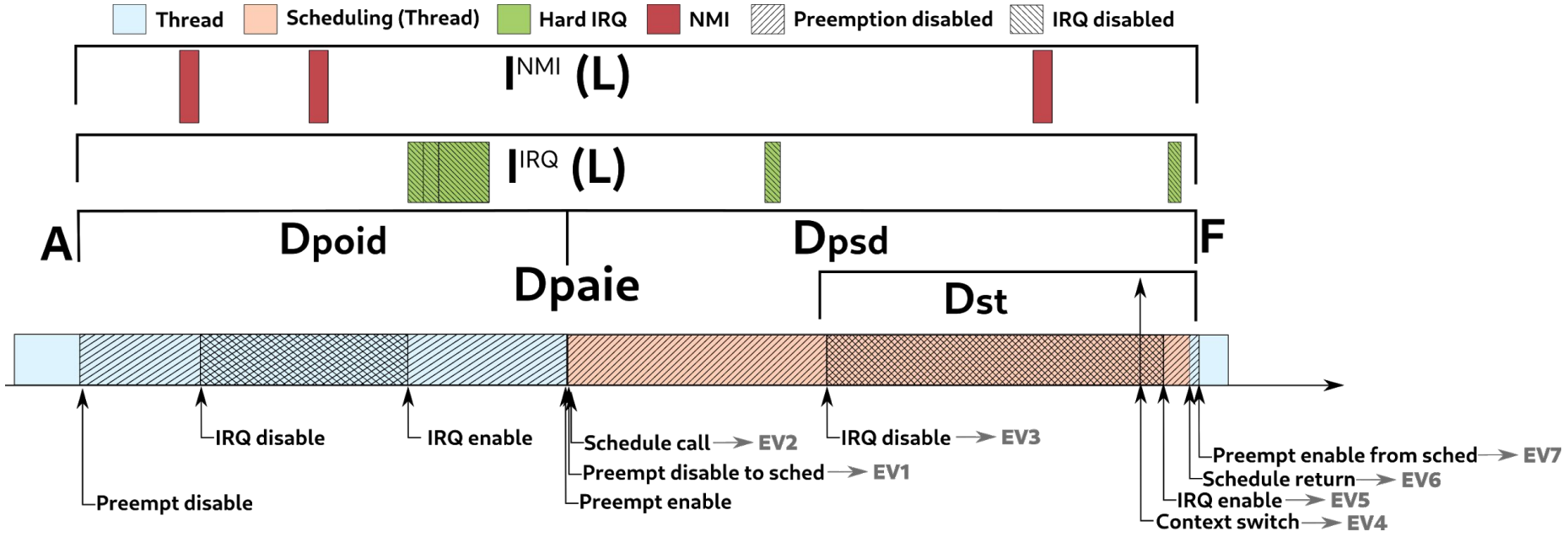




Timeline and cases

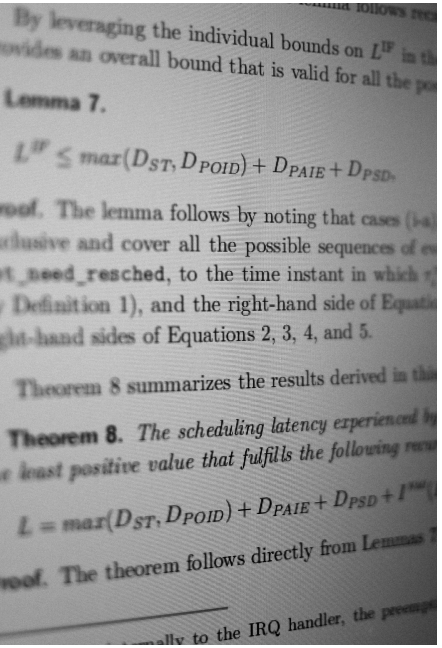
IRQ and NMI interference





Red Hat

And the *scheduling latency* bounds to:



By leveraging the individual bounds on L^{IF} in th
ovides an overall bound that is valid for all the po

Lemma 7.

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

roof. The lemma follows by noting that cases (1-a)
clusive and cover all the possible sequences of ev
t_need_resched, to the time instant in which r
Definition 1), and the right-hand side of Equati
ght-hand sides of Equations 2, 3, 4, and 5.

Theorem 8 summarizes the results derived in thi

Theorem 8. *The scheduling latency experienced by*
e least positive value that fulfills the following recu

$L = max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{***}($

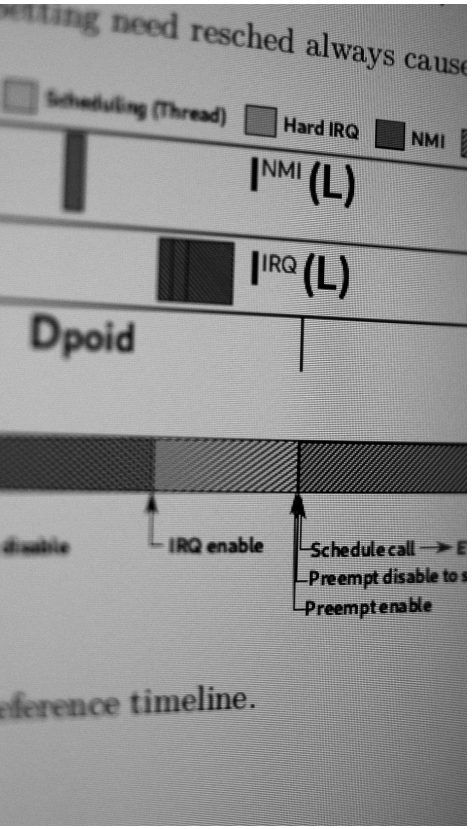
roof. The theorem follows directly from Lemmas

ally to the IRQ handler, the preempt

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

The bound considers all possible cases. Note that the Latency *L* is present in both sides of the equation.

So, L is bounded by the least positive value fulfilling the equation (like on RTA).



Interrupts are workload dependent

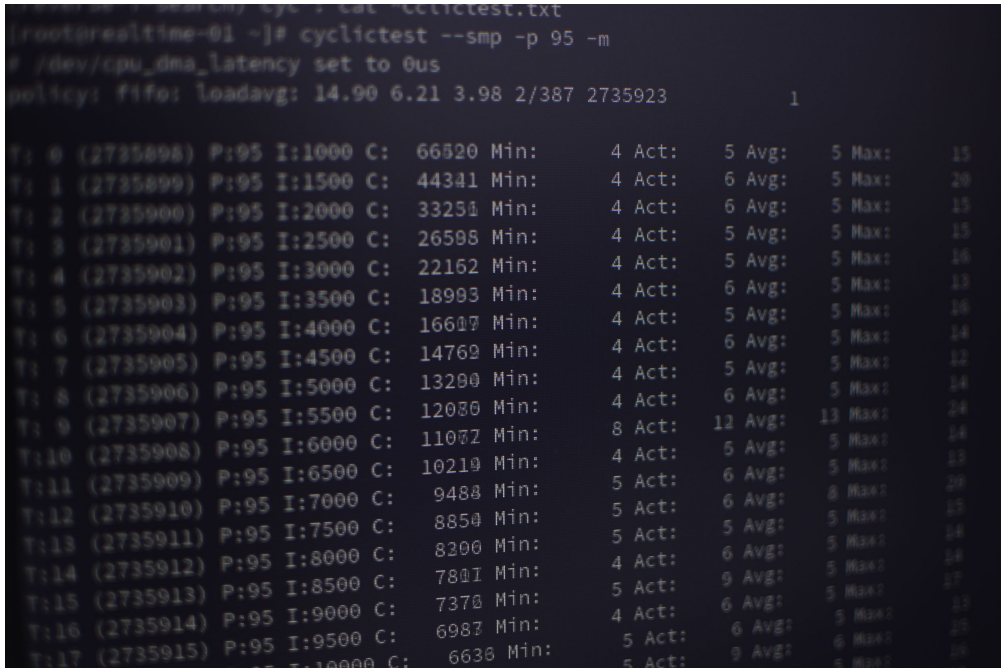
- Instead of proposing “the best” interrupt characterization, the rtsl reports the scheduling latency based on some well-known characterizations:
  - No interrupt
  - Worst single interrupt
  - Single occurence of all interrupts
  - Sporadic
  - Sliding window (Author’s preferred)
  - Sliding window with oWCET

This topic was heavily discussed at the Real-time Micro Conference (inside Linux Plumbers) in 2019, more info here:

Red Hat

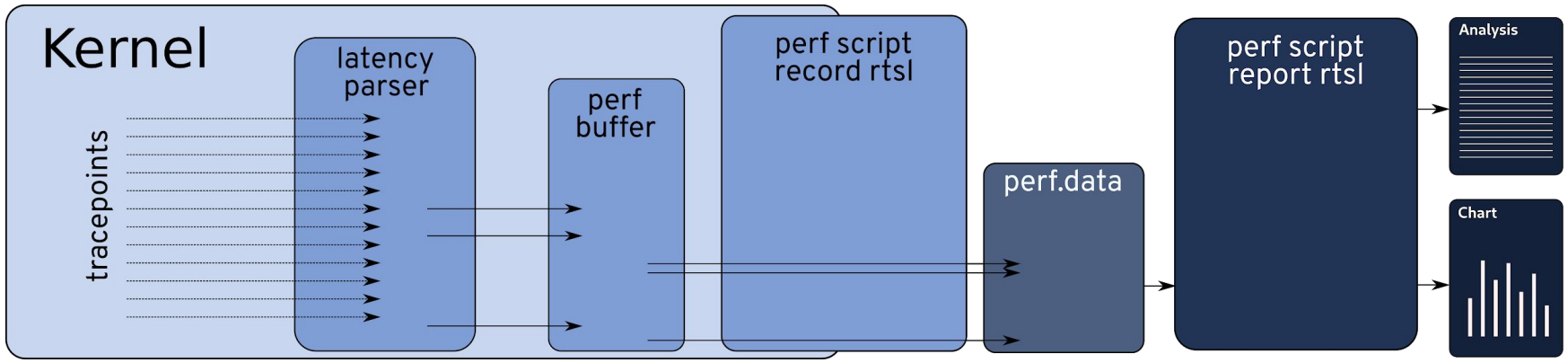
A practical scheduling latency estimation tool

Method and challenges



- Based on the latency bound
- The latency bound is based on the model
- The *model* is based on tracing of events
  - but high frequency events
    - hundreds MB/sec/CPU
- Challenges:
  - To minimize the (runtime) overhead
  - Work out-of-the-box

rt_sched_latency (rtsl)



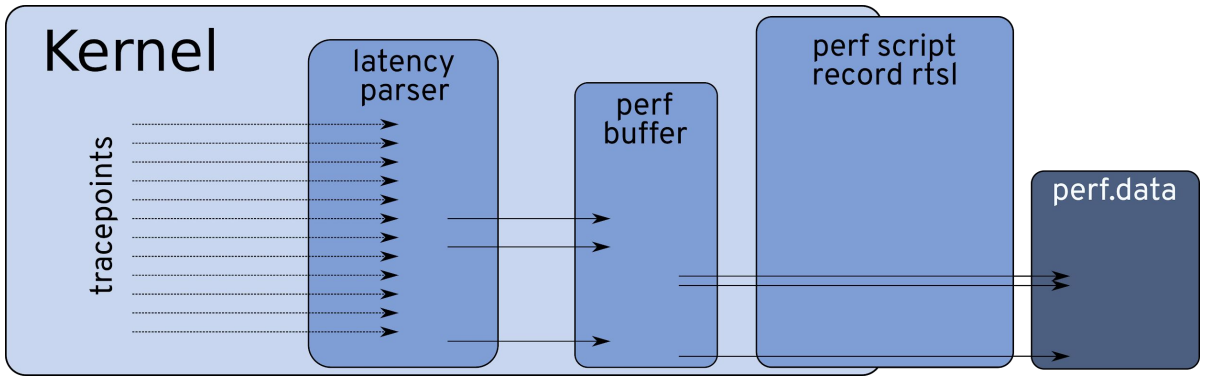
Based on **perf**

Works in two phases:

- The **record** mode saves the trace data;
- The **report** mode process the trace and does the analysis.

record phase

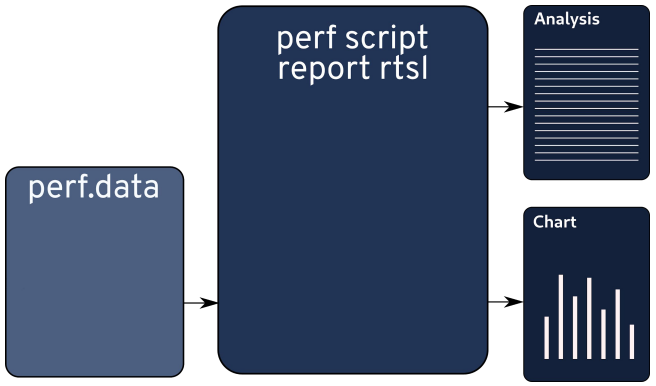
Low overhead trace recording



- Filters the high frequency trace
  - Doing in-kernel processing
- For blocking variables
  - Reports only the discover of new max values
- For IRQ and NMI
  - Reports one event for each occurrence
- Discounts the interference
  - e.g., IRQ interference on a **poid**

report phase

Low overhead trace recording



- After the capture, analyzes the trace.
  - All in user-space.
- Most of the analysis is done in python
  - Easy to extend
- Two outputs
  - Textual: good for debug
  - Chart: good comparisons (and papers :-))
- Does a per-cpu scheduling latency analysis
  - Using different IRQ/NMI characterization

rtsl report output

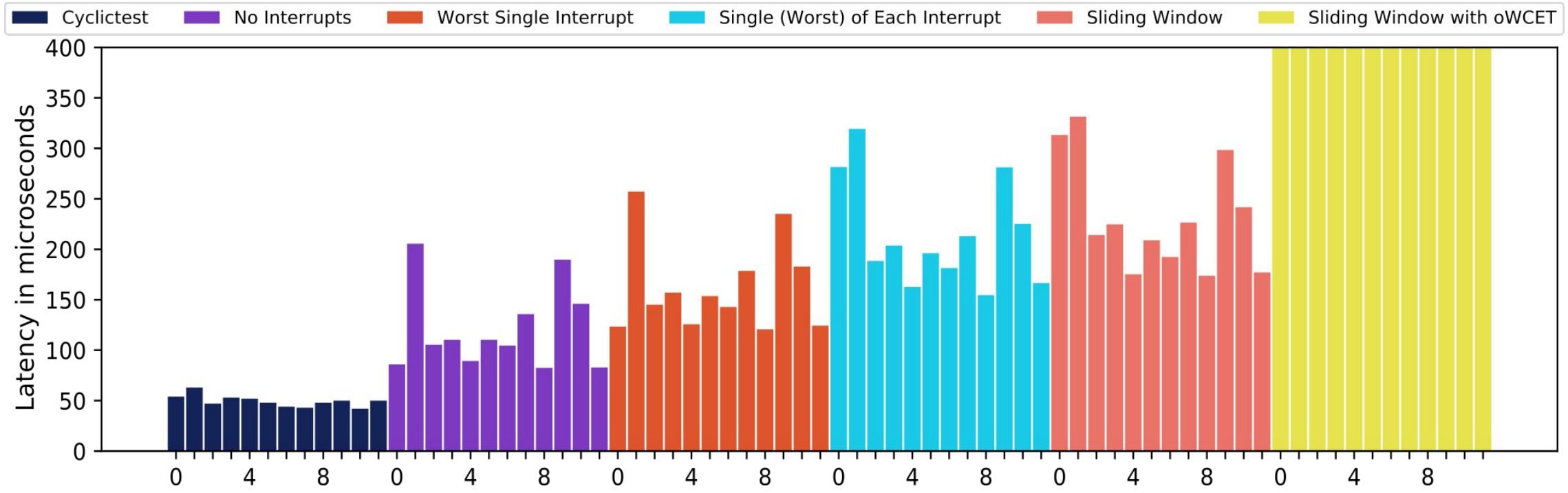
Textual output

```
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
```

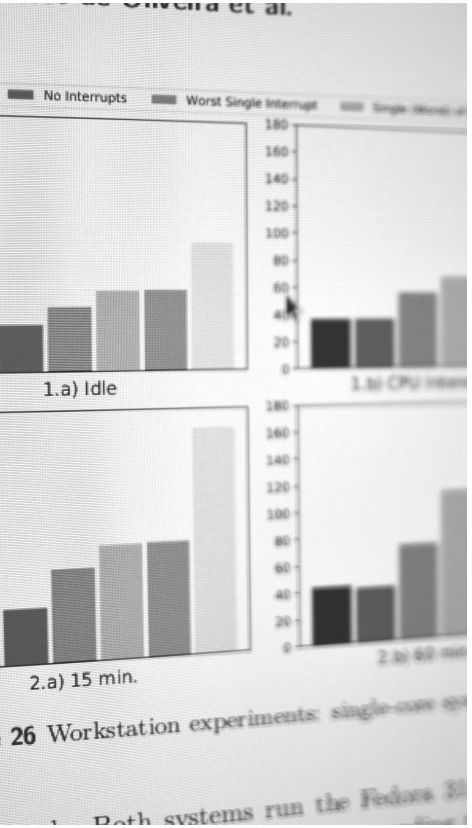
rtsl report output

Chart output



Red Hat

Experiments

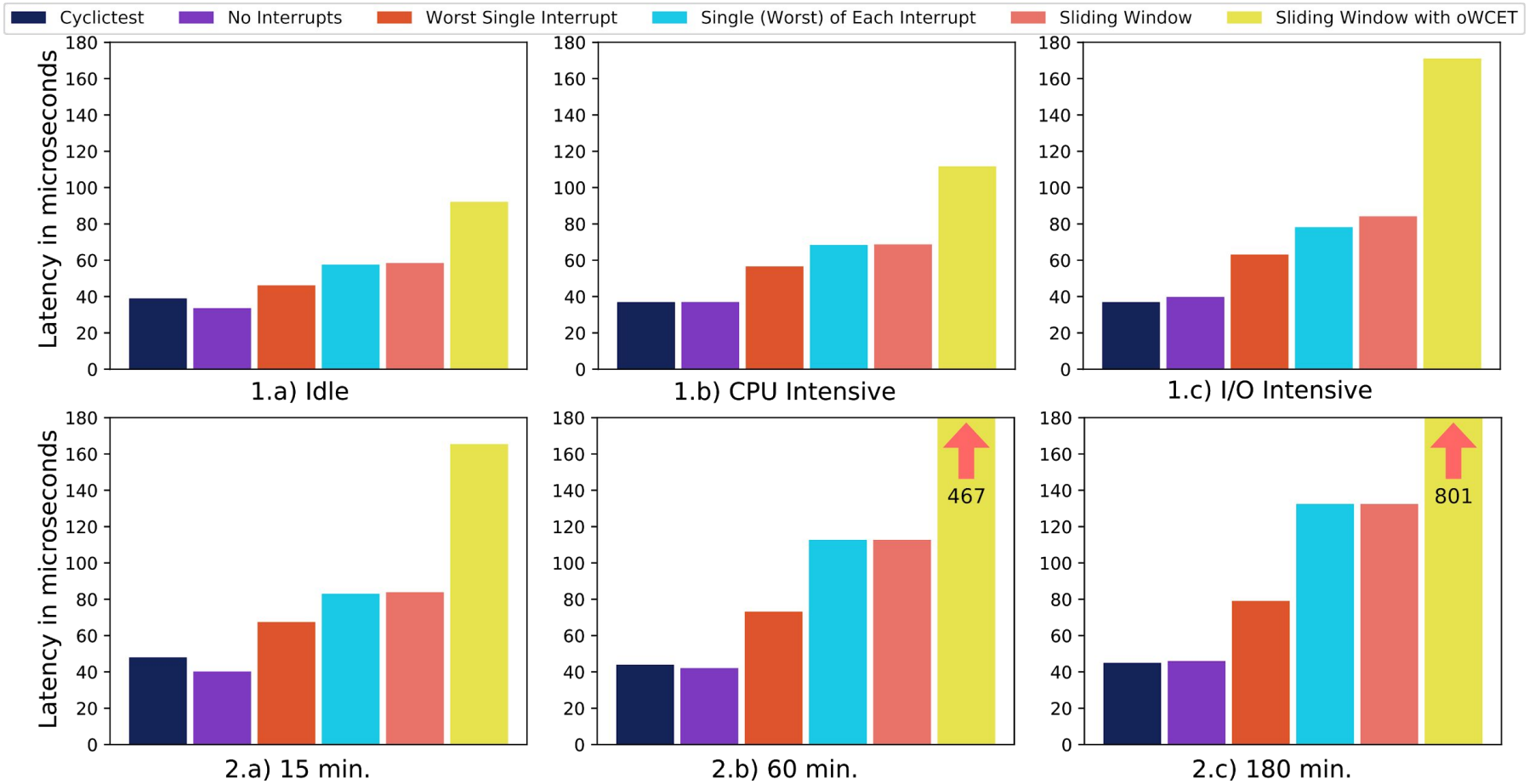


- Scheduling latency measurements on two systems:
  - workstation: eighth CPUs
  - server: twelve CPUs server
- Experiments:
  - Single-core
    - Different duration
    - Different workload
  - Multi-core
- Running in parallel with cyclictest
- Note: The goal of the experiments is to demonstrate the tool, not to define worst values.

The experiments passed by the artifact evaluation!

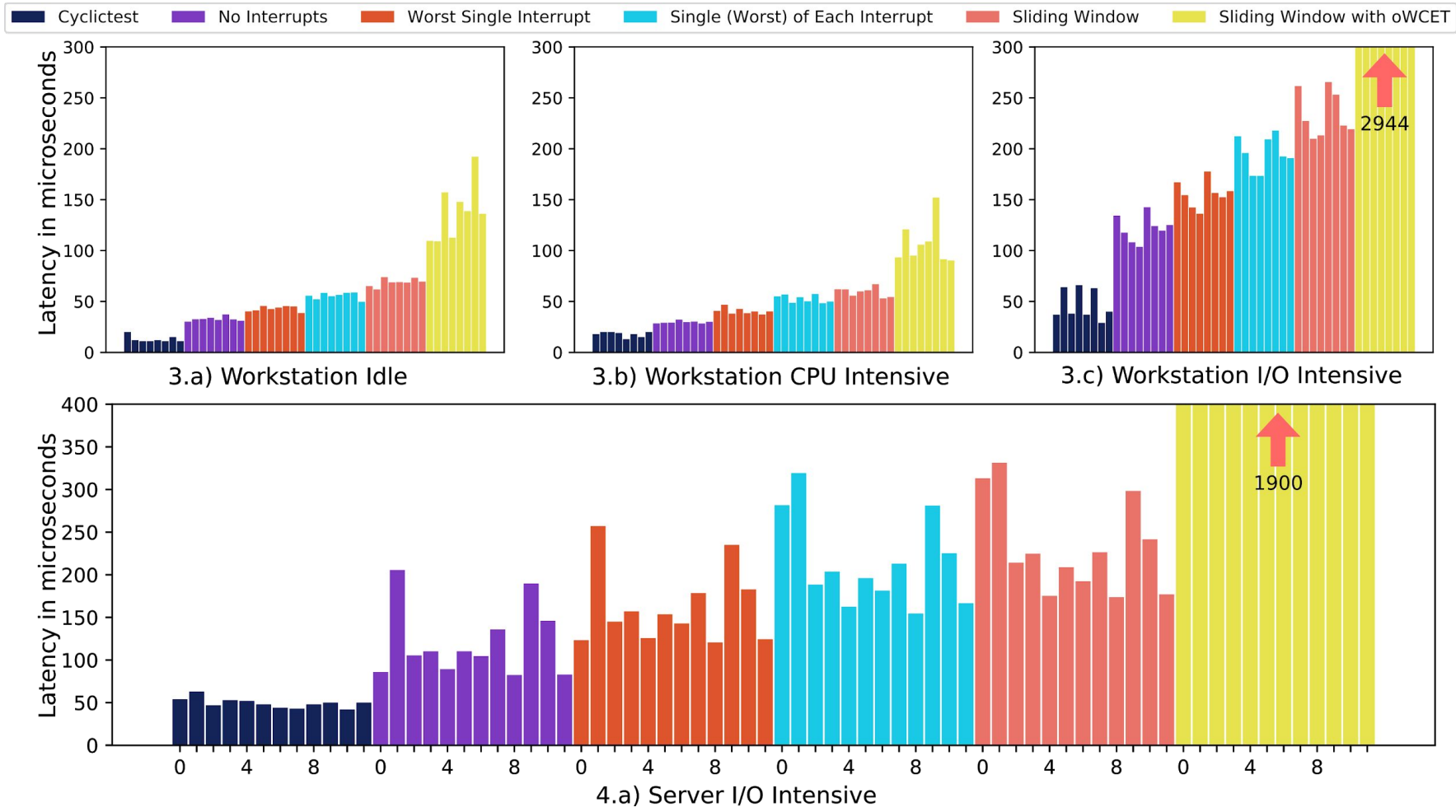
Red Hat

Single-core experiments



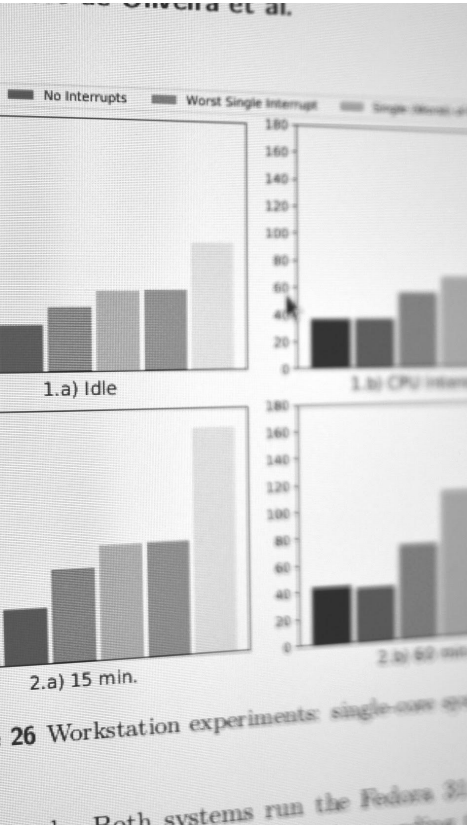
Red Hat

Multicore experiments



Red Hat

Remarks

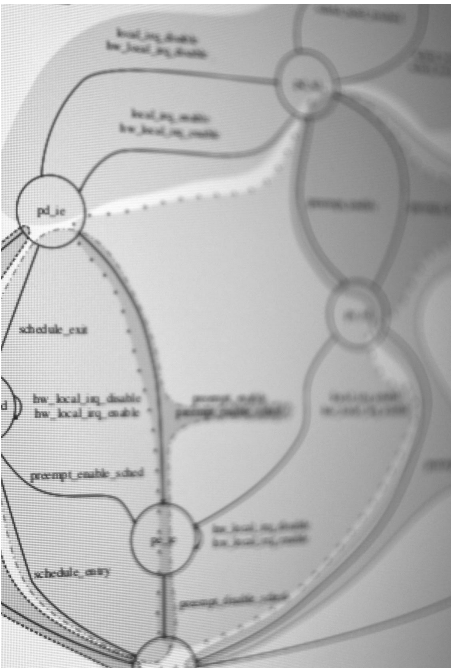


26 Workstation experiments: single-core

- The PREEMPT_RT preemption model is deterministic, and the scheduling latency is bounded
- The approach presented in this thesis opens the door for a new set of real-time analysis for Linux
  - The analytical interpretation of Linux thread model developed in this paper untight the Linux complexity, enabling the reasoning at a more sophisticated level
- Even though this rtsl finds higher scheduling latency values, they are still low enough to justify Linux as RTOS on the current scenarios
- rtsl is practical, and resolves many problems of cyclictest.
  - E.g., it can be used to point to the root causes of the latency
  - But still can, and should, be improved
    - Both with code, and other analysis.

For more information about the paper, like source code, other comments, Q&A, check its companion page!

Red Hat



Results

Papers

- **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.

- **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. **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

Main topic

- One workshop
- Four conferences
- One journal

All available here:

Red Hat

Papers

- **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. **A thread synchronization model for the PREEMPT_RT Linux kernel**, Elsevier Journal of Systems Architecture (JSA), Vol. 107, August 2020.
- **D. B. De Oliveira**, D. Casini, R. S. De Oliveira. T. Cucinotta. **Demystifying the Real-Time Linux Scheduling Latency**, in the Proceedings of the 32th Euromicro Conference on Real-time Systems (ECRTS), July 7-10th, 2020, Modena, Italy.

Main topic

Journal = Consolidated results

SEFM = Lost the fear of FM community

ECRTS = A top RT conference explaining the math behind the PREEMPT RT (**my goal**)

Other papers

- **D. B. De Oliveira**, R. S. De Oliveira (2016). **Timing analysis of the PREEMPT RT Linux kernel,** Softw. Pract. Exper., 46: 789– 819. doi: 10.1002/spe.2333.
- K. P. Silva, L. F. Arcaro, **D. B. de Oliveira**, R. S. 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.
- **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.
- **D. B. De Oliveira**, D. Casini, R. S. De Oliveira. T. Cucinotta. **Demystifying the Real-Time Linux Scheduling Latency (Artifact)**, in the Proceedings of the 32th Euromicro Conference on Real-time Systems (ECRTS), July 7-10th, 2020, Modena, Italy.

Other papers

One conference as third author

One workshop

One journal in the *informal* part of the Ph.D.

One artifact evaluation

Linux related conferences

- **18 talks at Linux/Open Source related conferences**
  - CZ 4, CA 4, FR 3, PT 2, UK 1, US 1, BR 1, IT 1, Online 1.
  - Mostly about the topics of the thesis
  - But also about other RT and trace topics
- **I organized:**
  - Real-time micro conference at Linux Plumbers 2019
  - Real-time Linux Summit 2019
  - Real-time micro conference at Linux Plumbers 2020...
  - Real-time Linux Summit 2020...
- **Helped on:**
  - Scheduling micro conference at Linux Plumbers 2019
  - Scheduling micro conference at Linux Plumbers 2020

Slides of my talks are all here:

Red Hat

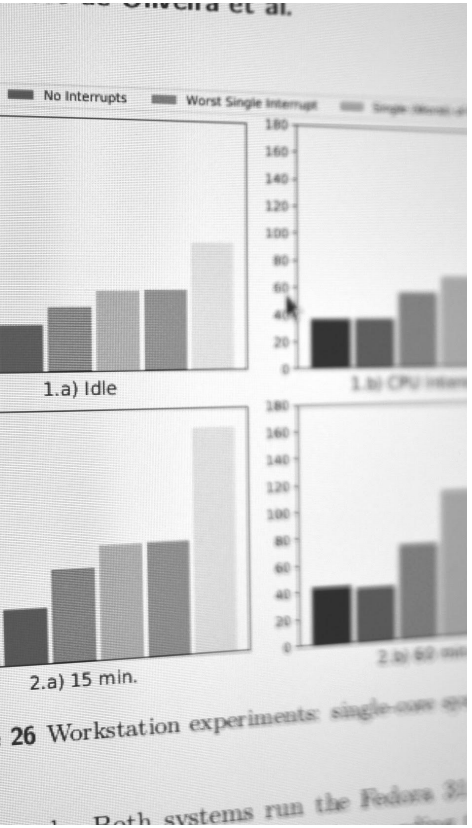
Other academic activities

- **Classes:**
  - Real-time Linux at Real-time course (UFSC)
  - Formal verification at Component-based software design course (SSSUP)
- **Managed the cotutela agreement**
  - Lots of work to merge IT/BR Ph.D. rules
- **Collaborations with other research groups**
  - **Boston University** - Unikernel
  - **ETH Zurich** - FM
- Reviewed papers for SBESC
- PC of EWiLi and (postponed to 2021) RT Cloud Workshop inside ECRTS
- Participated in a **European project submission**
  - Ericsson/Red Hat/Uni Torino/Lund University/Sant'Anna
  - Not as a student but as Red Hat (industrial partner)

Slides of my talks are all here:

Red Hat

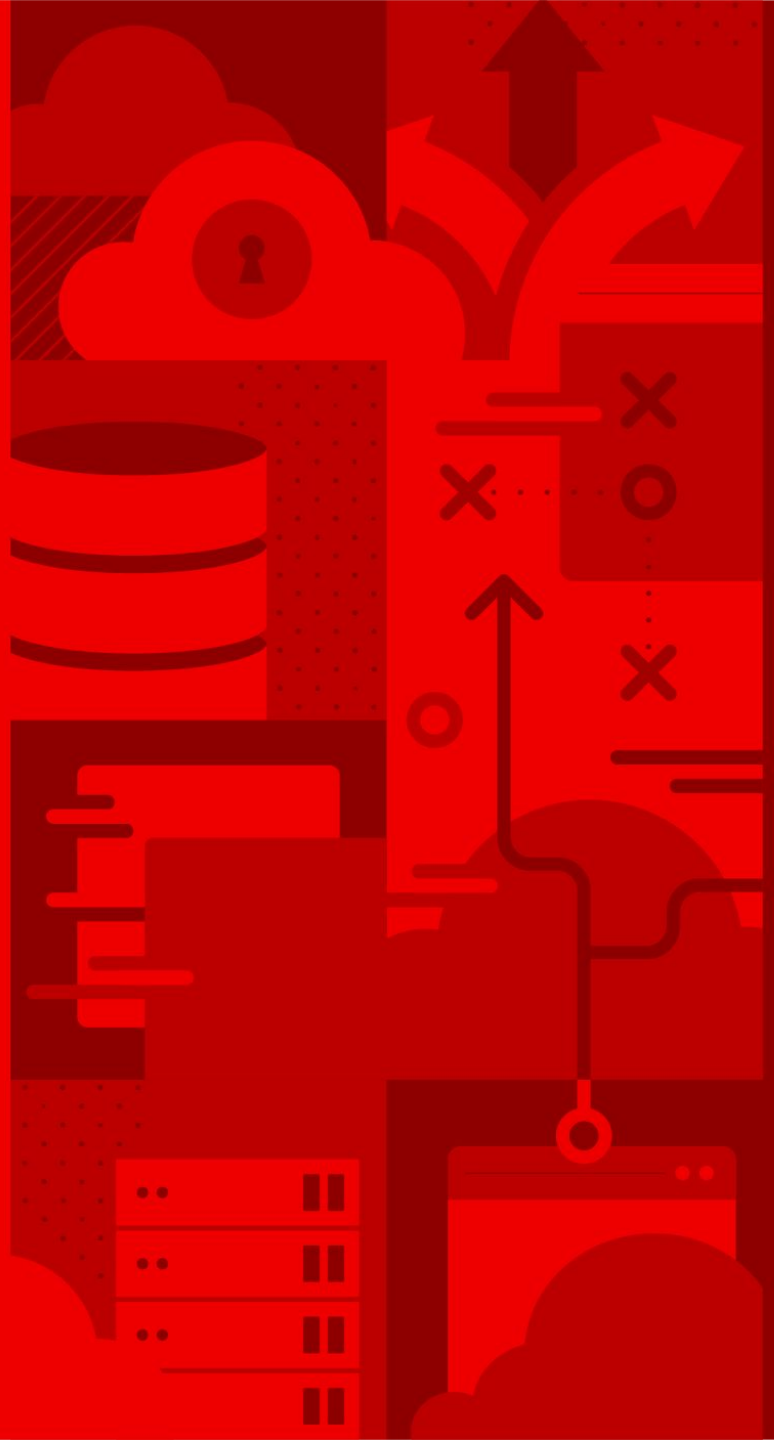
Final words



- The idea of using formal methods to explain Linux was risky:
  - I touched state-explosion many times
  - Kernel generates GB of events per second
- The simplicity of automata was the key factor
  - It was simple on purpose
- The RV results were WAY better than expected
- The Latency paper was the goal and, with that in a top conference, I could finally sleep in peace with myself
- This is just the beginning, because there is a lot of work to be done
- Thanks Tommaso, Romulo, Casini, Luca and Clark

For more information about the thesis, like source code, other comments, Q&A, check its companion page!

Red Hat



Questions?

Red Hat

That is all for today, thanks for watching, and have a nice day!

Red Hat
Enterprise Linux

Sant'Anna
School of Advanced Studies – Pisa

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Thanks!

Red Hat