Red Hat Enterprise Linux

Sant'Anna School of Advanced Studies – Pisa

UNIVERSIDADE FEDERAL DE SANTA CATARINA

A theorem for the RT scheduling latency

(and a measurement tool too)

Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira and Tommaso Cucinotta
Principal Software Engineer

Red Hat

Red Hat Enterprise Linux

Sant'Anna School of Advanced Studies – Pisa

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Episode III - Showing the math

Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira and Tommaso Cucinotta
Principal Software Engineer

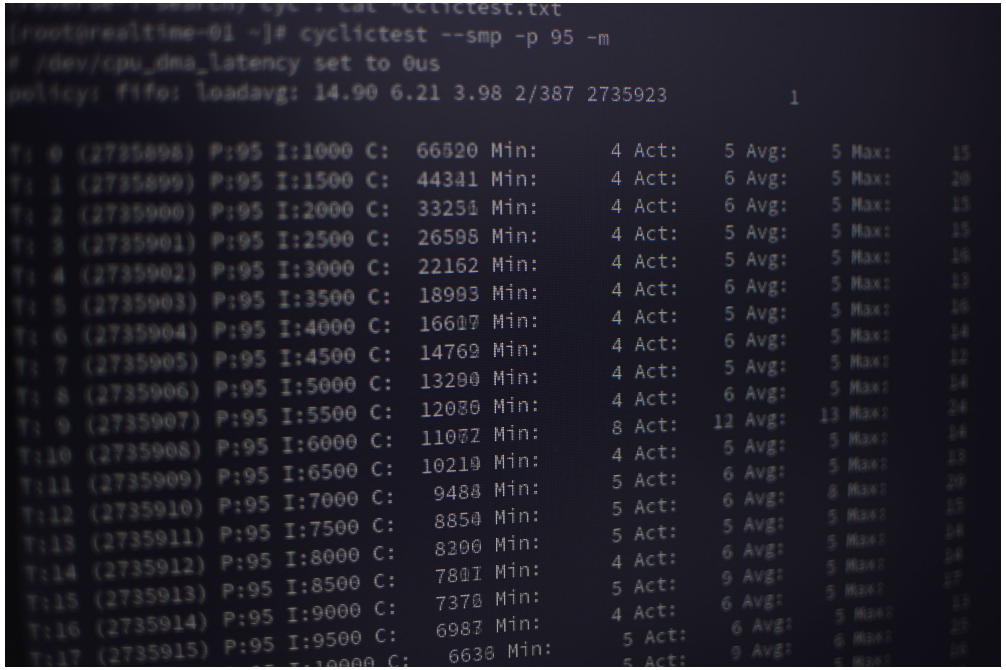
Red Hat

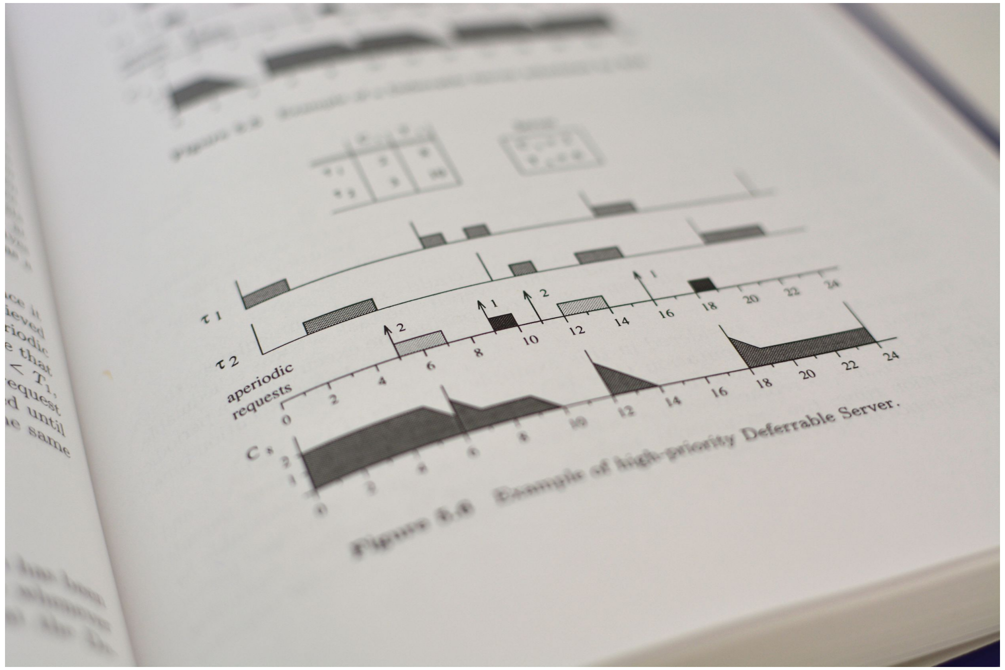
Real-Time Linux

“Real-Time” Linux

Real-Time Linux vs Real-Time theory

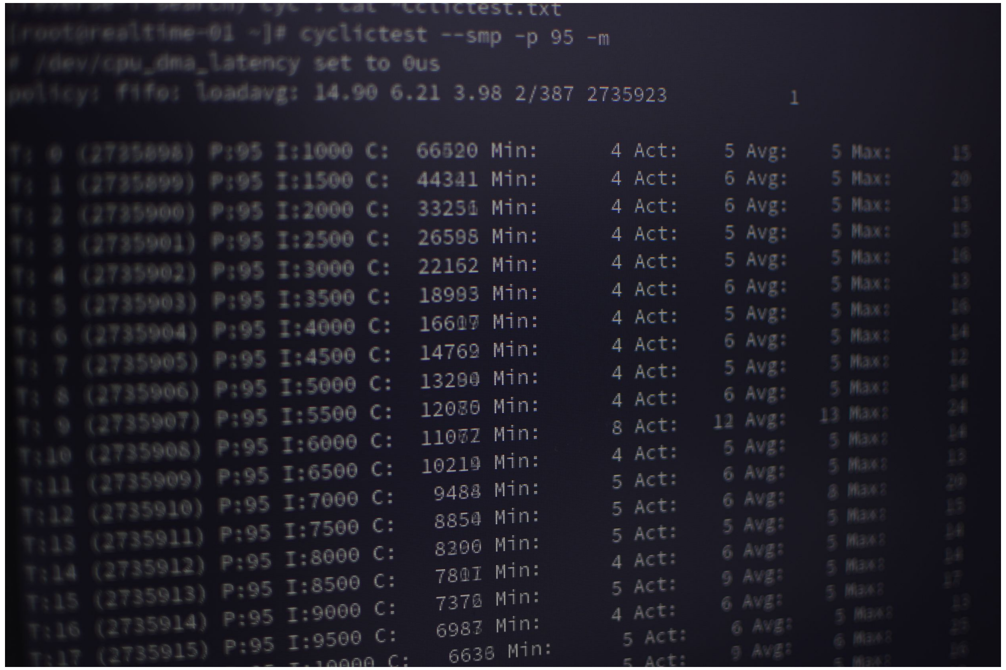
Experimental vs Analytical





Real-Time Linux vs Real-Time theory

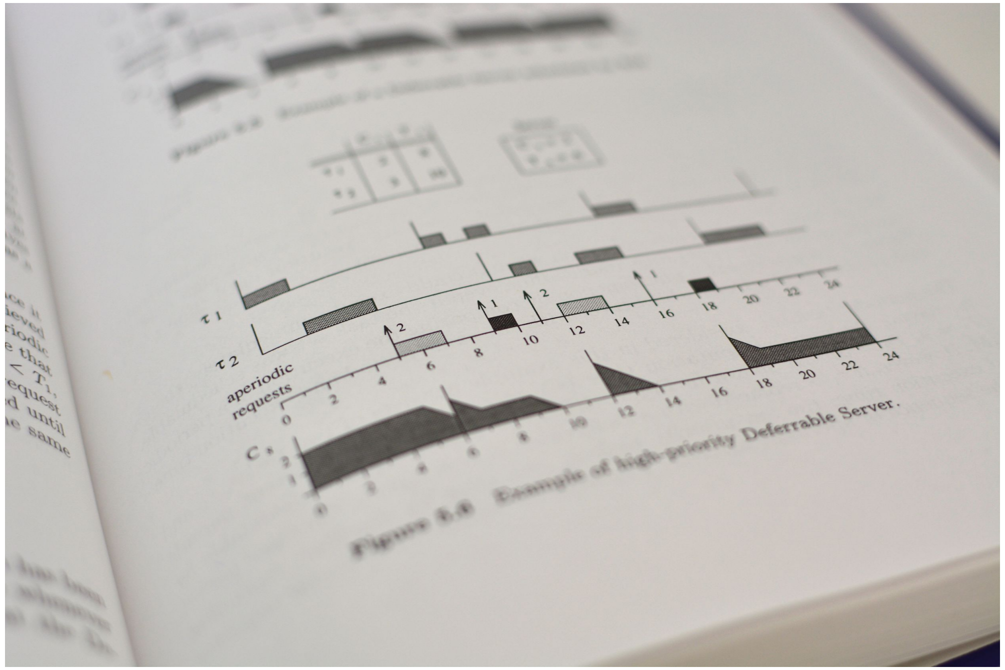
Linux approach



- 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 “closed-box” test
    - No demonstration
    - Does not provide evidence of “root-cause”

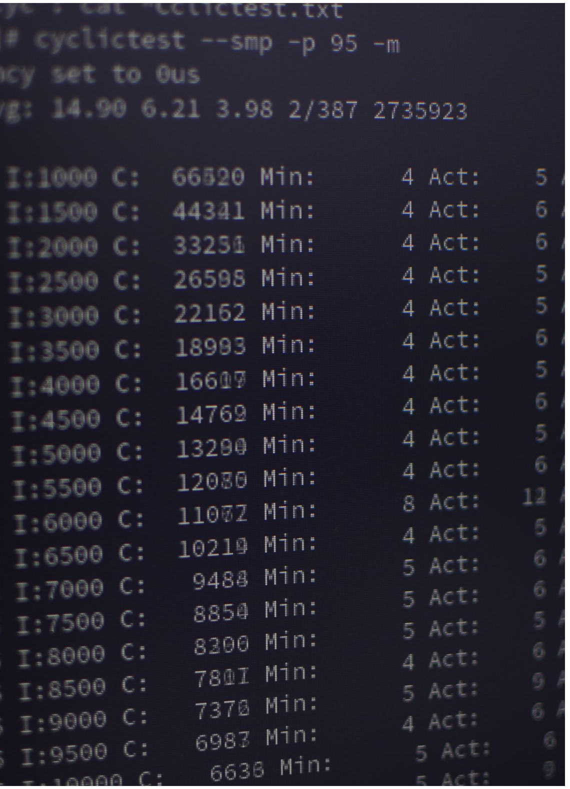
Real-Time Linux vs Real-Time theory

Real-time analysis

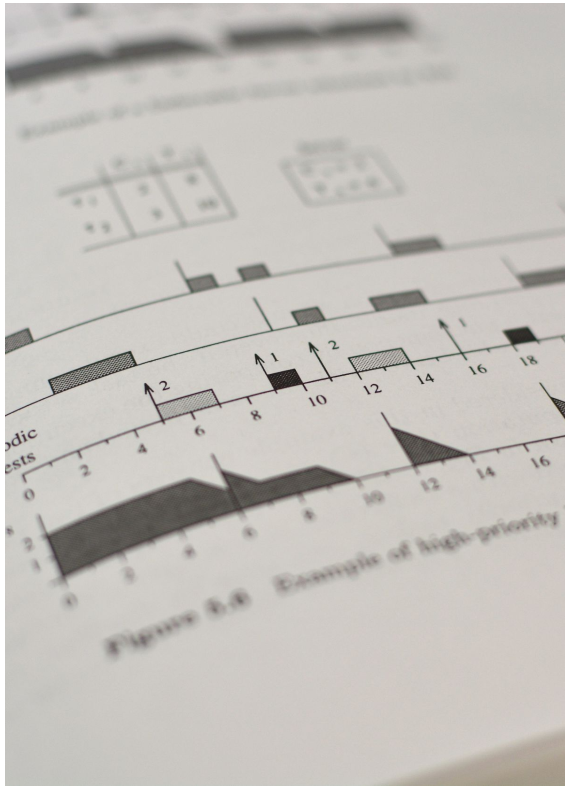


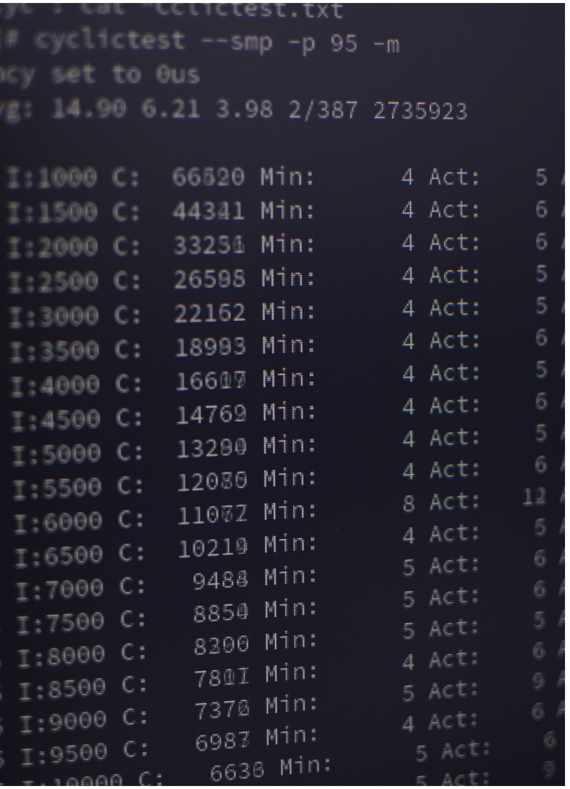
- Based on the timing description of the system
- Capture all behaviors
- Precisely define the worst cases
- But depends on a precise definition of the system
- Often overly-simplified

Red Hat



But, I like both.





Demystifying the Real-Time Linux Scheduling Latency

Daniel Bristot de Oliveira
Red Hat, Italy
bristot@redhat.com

Daniel Casini
Scuola Superiore Sant'Anna, Italy
daniel.casini@santannapisa.it

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

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

Abstract

Linux has become a viable operating system for many real-time workloads. However, the black-box approach adopted by `cyclictest`, the tool used to evaluate the main real-time metric of the kernel, the scheduling latency, along with the absence of a theoretically-sound description of the in-kernel behavior, sheds some doubts about Linux meriting the real-time adjective. Aiming at clarifying the PREEMPT_RT Linux scheduling latency, this paper leverages the *Thread Synchronization Model* of Linux to derive a set of properties and rules defining the Linux kernel behavior from a scheduling perspective. These rules are then leveraged to derive a sound bound to the scheduling latency, considering all the sources of delays occurring in all possible sequences of synchronization events in the kernel. This paper also presents a tracing method, efficient in time and memory overheads, to observe the kernel events needed to define the variables used in the analysis. This results in an easy-to-use tool for deriving reliable scheduling latency bounds that can be used in practice. Finally, an experimental analysis compares the `cyclictest` and the proposed tool, showing that the proposed method can find sound bounds faster with acceptable overheads.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems

Keywords and phrases Real-time operating systems, Linux kernel, PREEMPT_RT, Scheduling latency

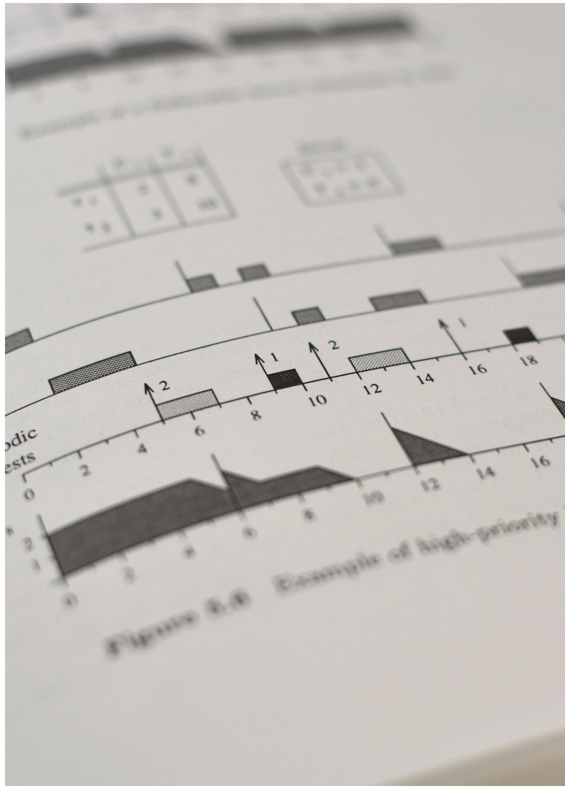
Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.9

Supplementary Material ECRTS 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.1.3`.
Supplement material and the code of the proposed tool is available at: `https://bristot.me/demystifying-the-real-time-linux-latency/`

Funding This work has been partially supported by CAPES, The Brazilian Agency for Higher Education, project PrInt CAPES-UFSC "Automation 4.0."

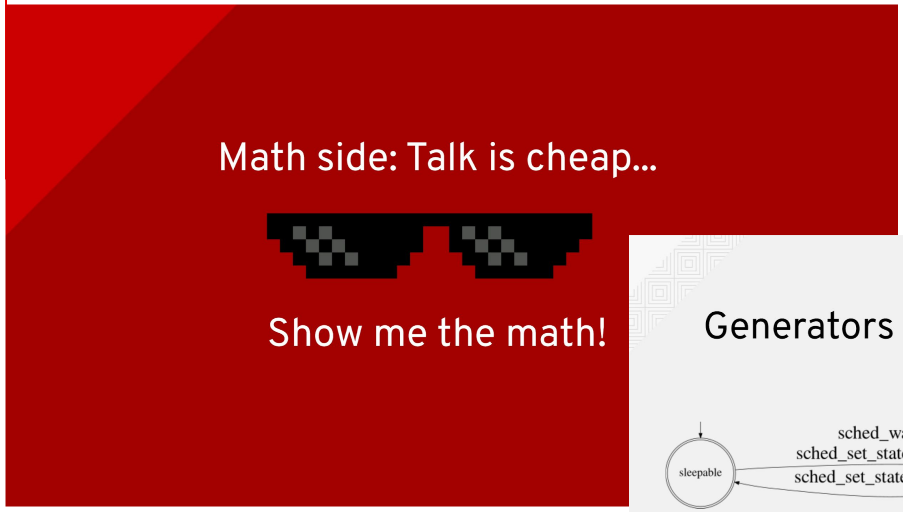
Acknowledgements The authors would like to thank Thomas Gleixner, Peter Zijlstra, Steven Rostedt, Arnaldo Carvalho De Melo and Clark Williams for the fruitful discussions about the model, analysis, and tool.

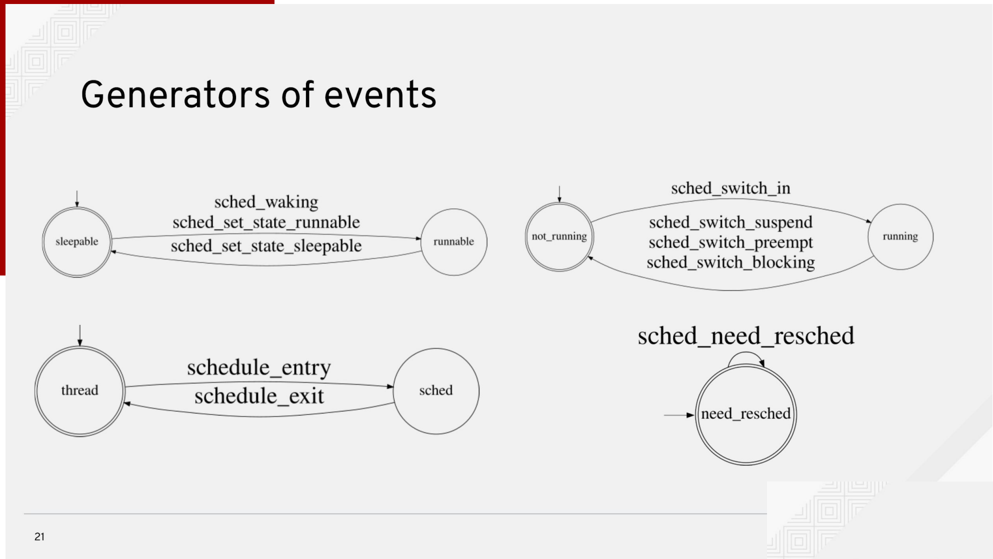
32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp; Article No. 9; pp. 9:1–9:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

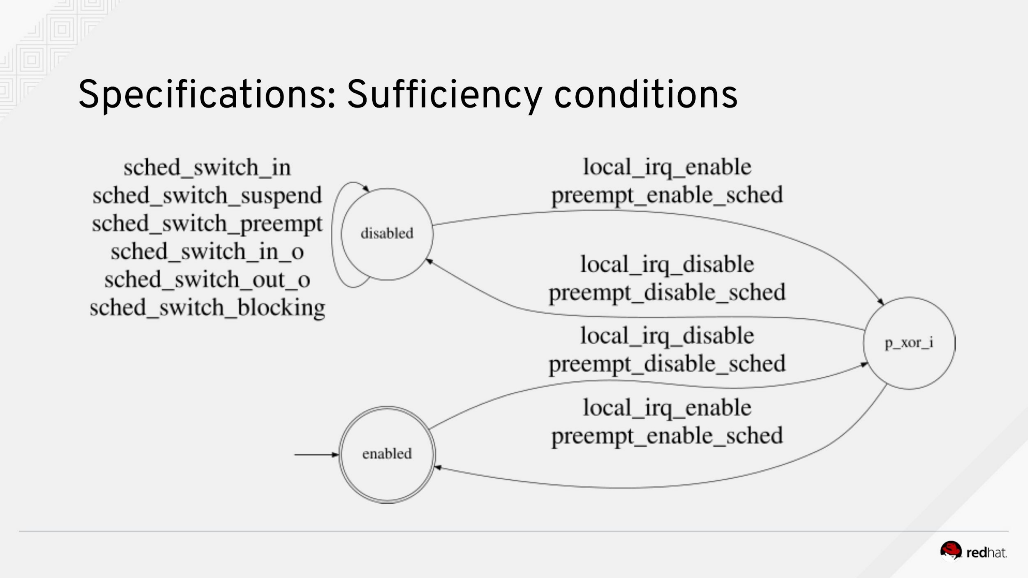


Episode I: getting formal

Red Hat


Math side: Talk is cheap...
Show me the math!

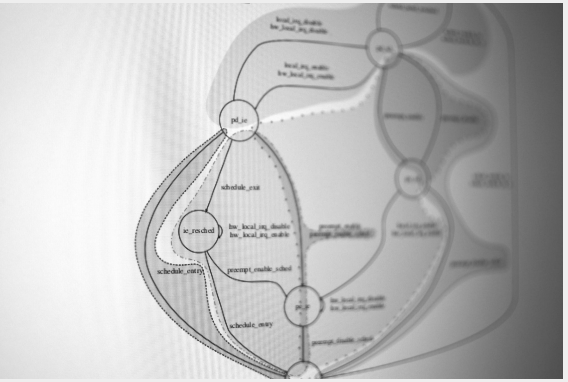

Generators of events
sleepable
sched_waking
sched_set_state_runnable
sched_set_state_sleepable
runnable
not_running
sched_switch_in
sched_switch_suspend
sched_switch_preempt
sched_switch_blocking
running
thread
schedule_entry
schedule_exit
sched
sched_need_resched
need_resched
21


Specifications: Sufficiency conditions
sched_switch_in
sched_switch_suspend
sched_switch_preempt
sched_switch_in_o
sched_switch_out_o
sched_switch_blocking
disabled
local_irq_enable
preempt_enable_sched
local_irq_disable
preempt_disable_sched
local_irq_disable
preempt_disable_sched
p_xor_i
local_irq_enable
preempt_enable_sched
enabled
redhat.

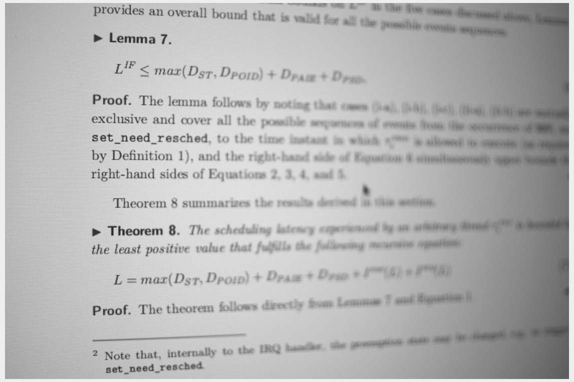
Demystifying the Real-Time Linux Scheduling Latency

Approach

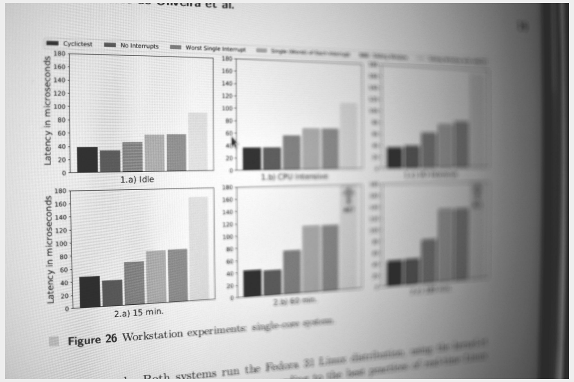
Formal specification



Scheduling latency bound



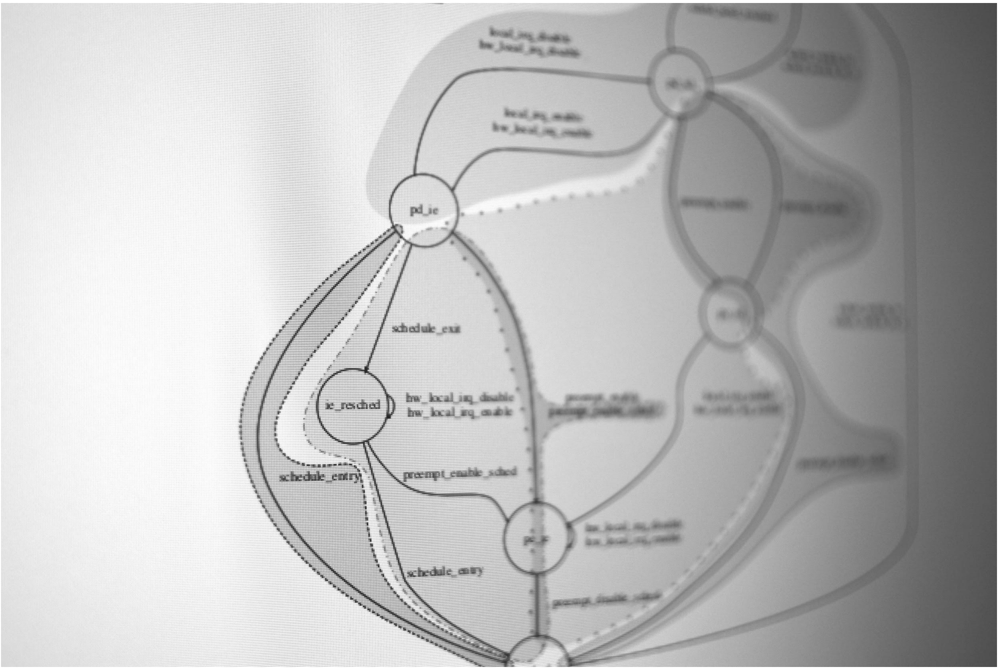
Measurement and analysis



Red Hat

From formal specification to synchronization rules

Formally backed natural language arguments



- Generators
  - Basic/Independent behavior
  - e.g., irq_disable/enable, scheduler call
- Translated into a set of operations

- Specifications
  - Relations among generators
  - e.g., necessary conditions to call the scheduler
- Translated into a set of synchronization rules

Scheduling latency definition

The **scheduling latency** experienced by an arbitrary thread τ is:

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

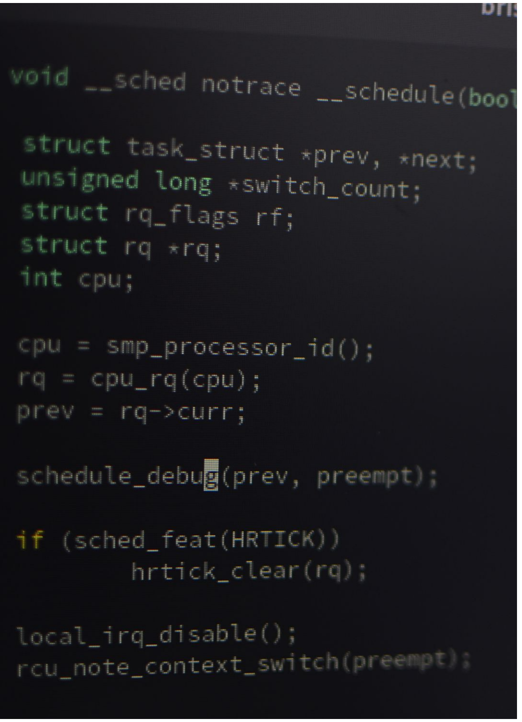
ready AND **with the highest priority:**

It covers the case in which these **two actions are not a single event.**

It is **scheduler independent.**

There is only one highest priority thread on a CPU: it is **the one selected to run** by the scheduler.

Interference and blocking



The **scheduling latency** is caused by:

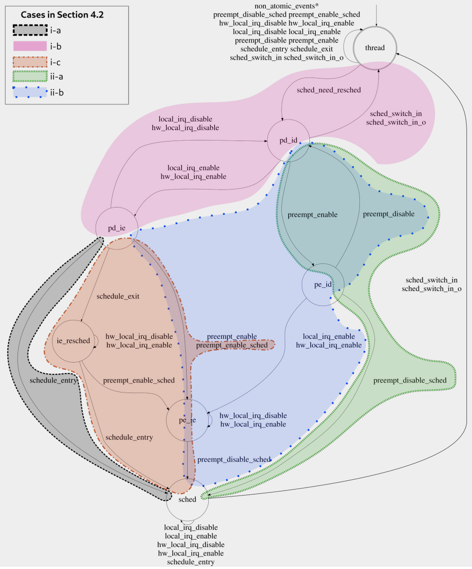
- **Blocking** from the current (and so lower) priority thread;
  - Including scheduling.
- **Interference** from IRQs and NMI.

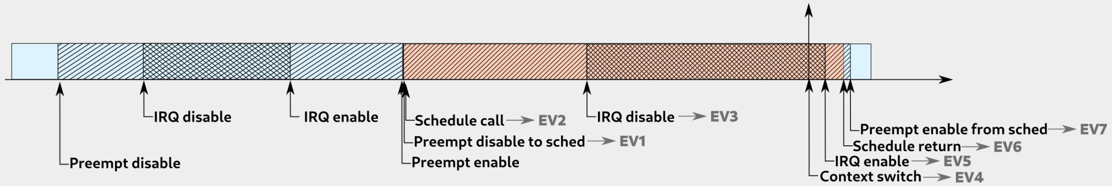
This are well established terms in the real-time scheduling literature:

Interference from higher priority, **blocking from lower priority.**

Blocking bound

From the specification that bounds the block to a timeline





Scheduling latency: start

- The **longest time** elapsed **between** the *time A* in which any job of τ becomes **ready and with the highest priority**:
  - Generalized to the **need_resched** event
    - Works for all schedulers
      - cyclictest does not work for DL with NR_TASKS > CPUs.
    - Works for all conditions
      - E.g., a throttled DL task after a replenishment will cause a need resched without a wakeup.
  - Has **preempt and IRQ disabled** as **necessary conditions**
    - So we use the occurrence of the first **necessary condition** as the starting point of the critical window.
      - E.g., when preemption was disabled for the first time.

The wakeup is the only event that causes a need resched, and that is why it was not used here.

But ready means that the task was awakened.

Scheduling latency: end

- And the *time F* in which the scheduler returns and allows τ to **execute its code**.
  - Generalized to the **preempt_enable** after **__schedule()**
    - Implies that the system **crossed the context switch** code path.
    - Context switch implies **__schedule()**
    - **Context switch needs:**
      - **Preempt disable to schedule as necessary condition**
      - **irqs disabled by thread as necessary condition**

We are looking for a safe-bound, and so we have to put pessimism values.

We can latter reduce the pessimism, but with safe arguments.

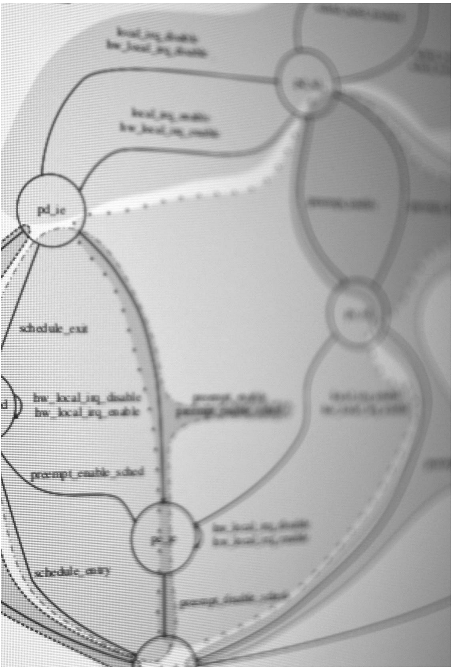
How do we bound that?

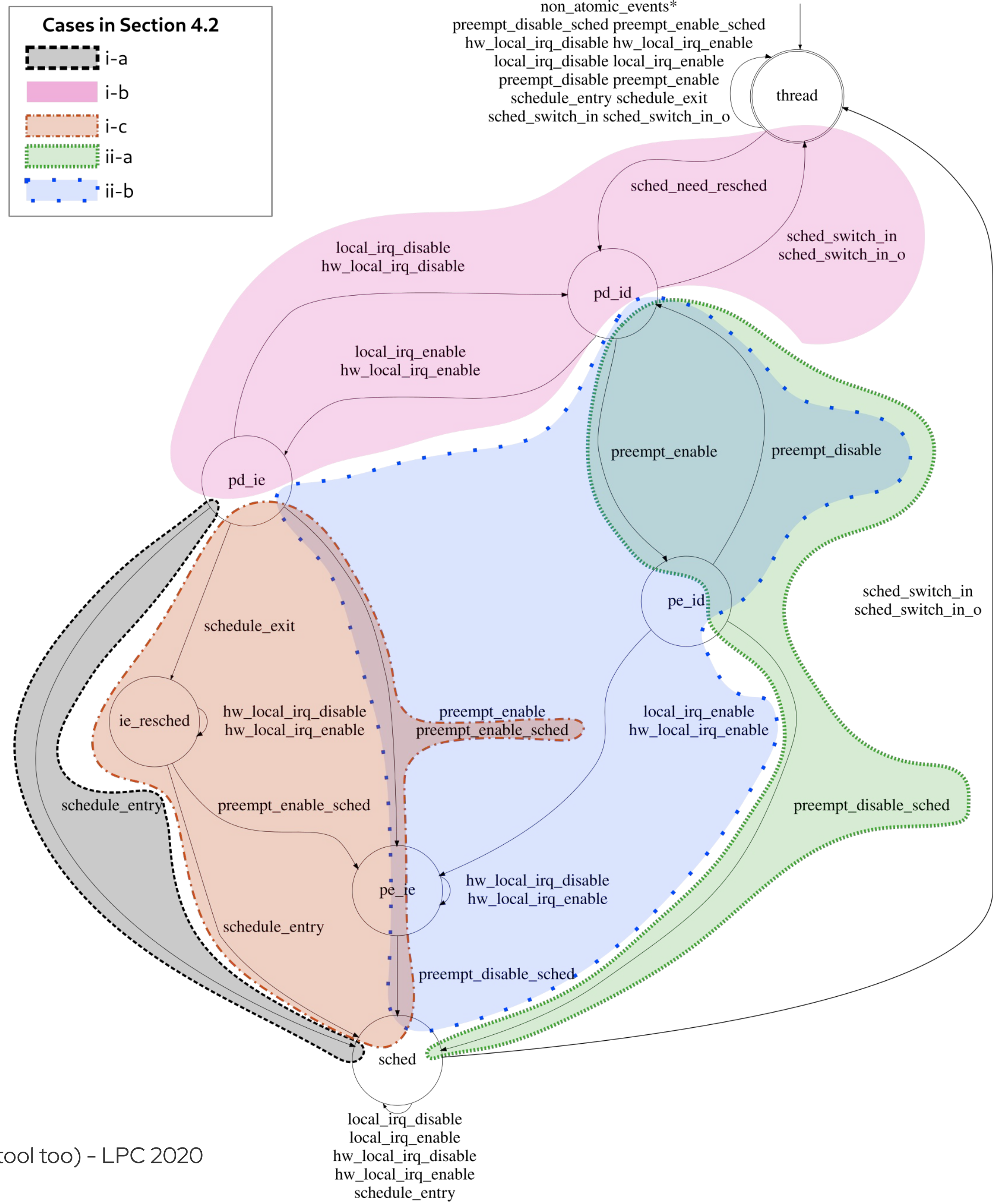
Red Hat

Blocking bound

Need resched -> ctxsw

All possible cases

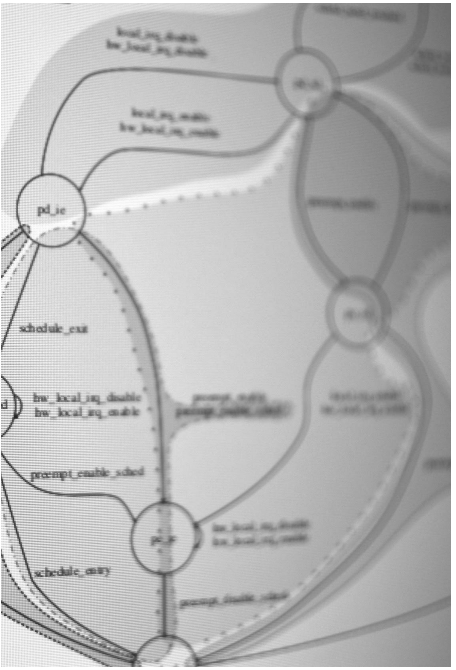


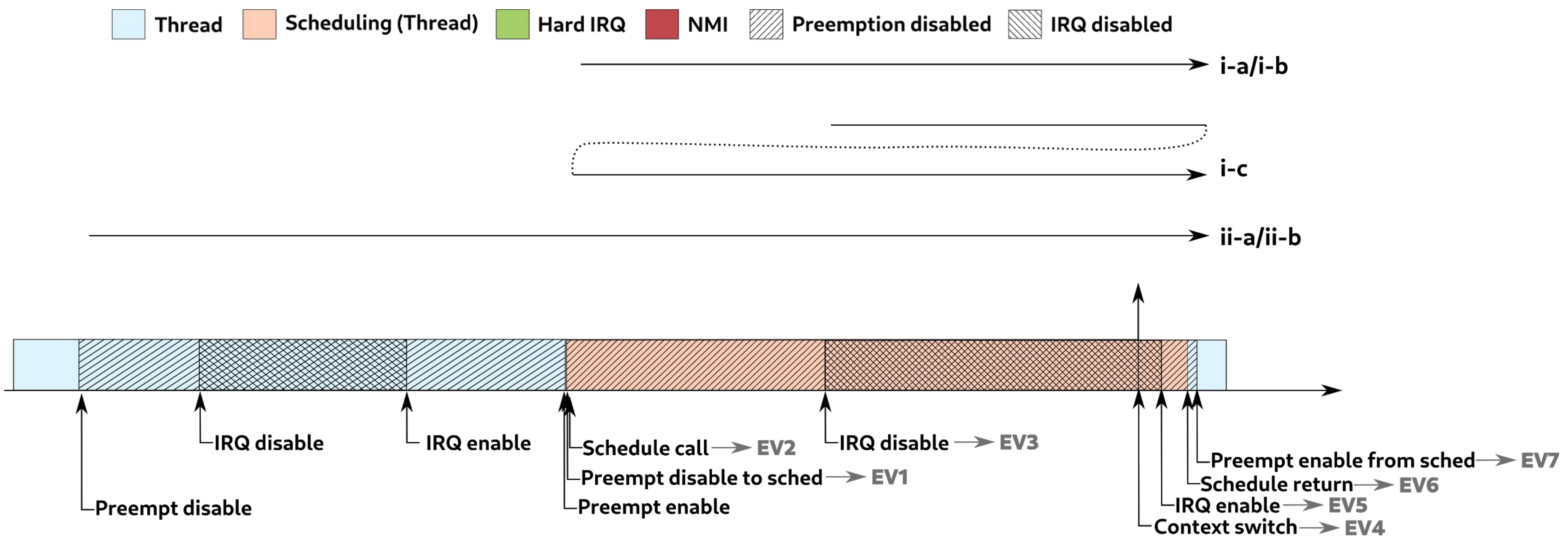


Red Hat

Timeline and cases

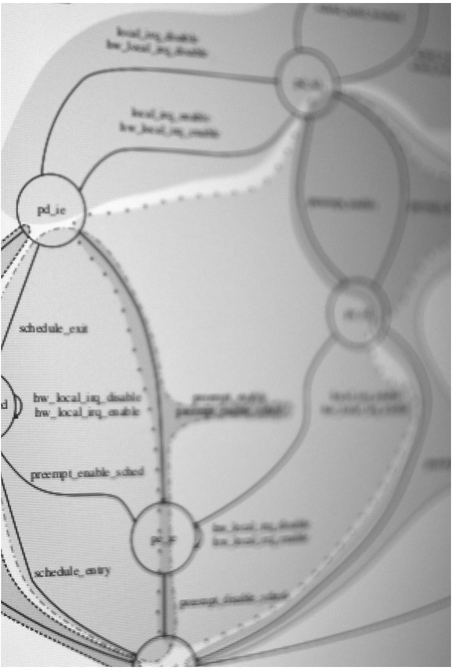
All possible cases

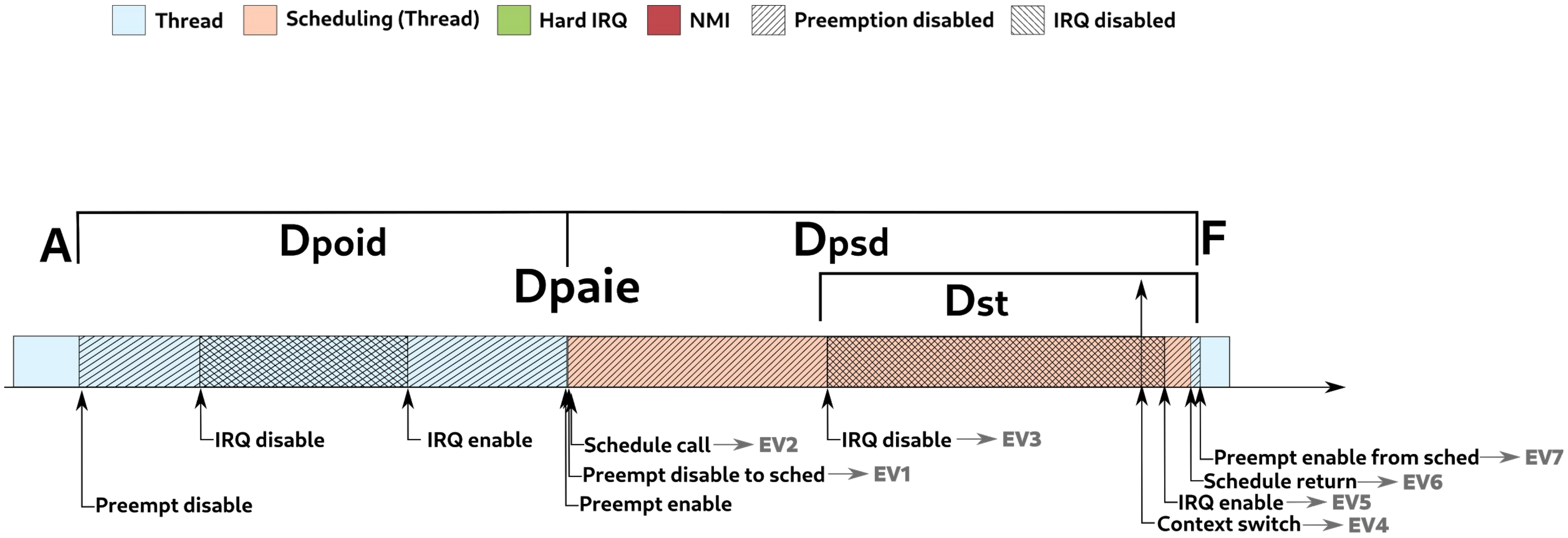




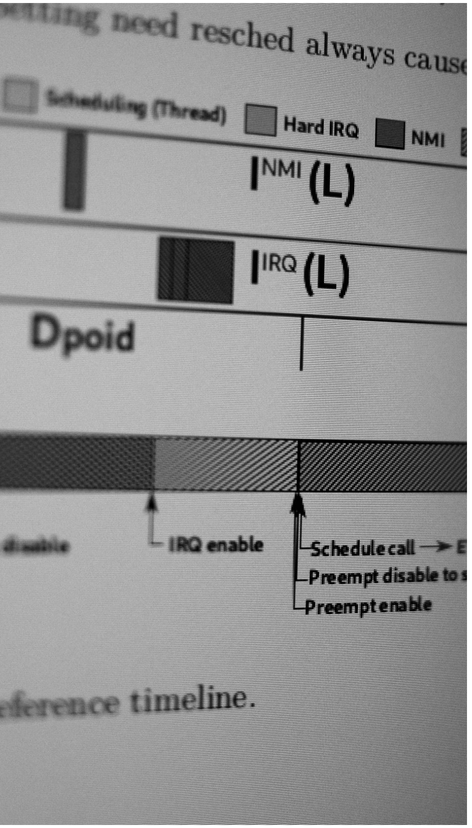
Timeline and cases

Variables in the the timeline





Blocking variables

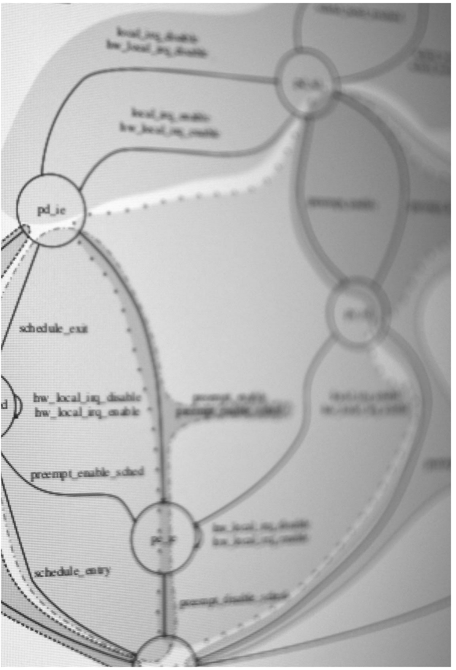


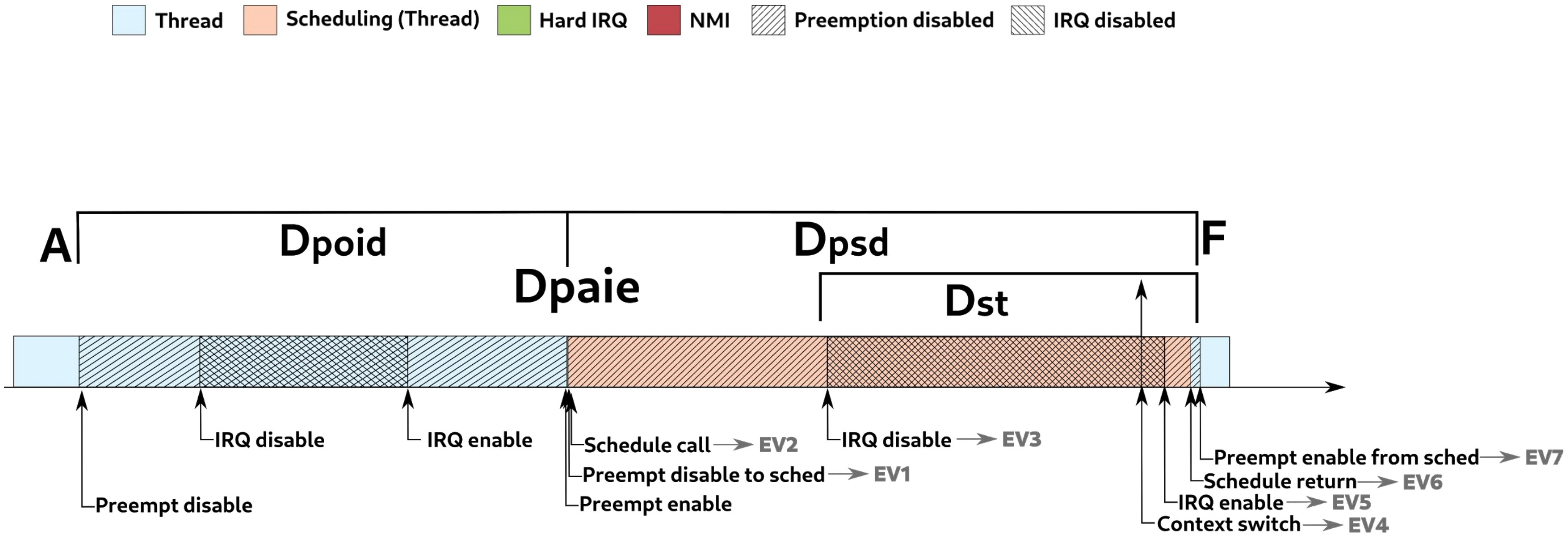
- **Dpoid**: preemption or interrupts disabled to postpone the scheduler;
- **Dpaie**: preemption and interrupts enabled, as a transient state from **poid** to **psd**; when scheduling a new highest priority thread.
- **Dpsd**: preemption disable to schedule;
- **Dst**: delay caused by the scheduling 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

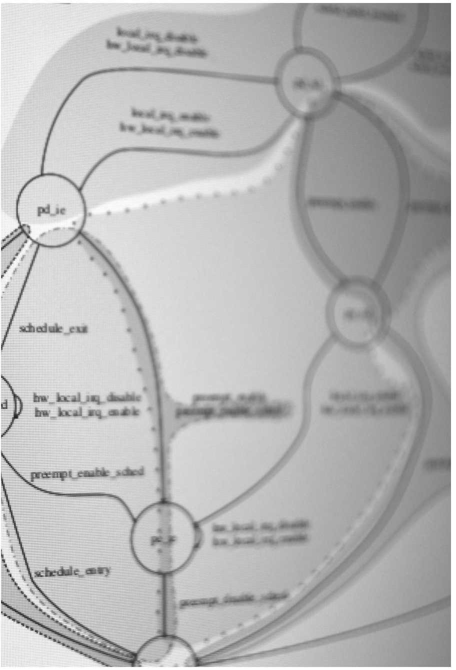


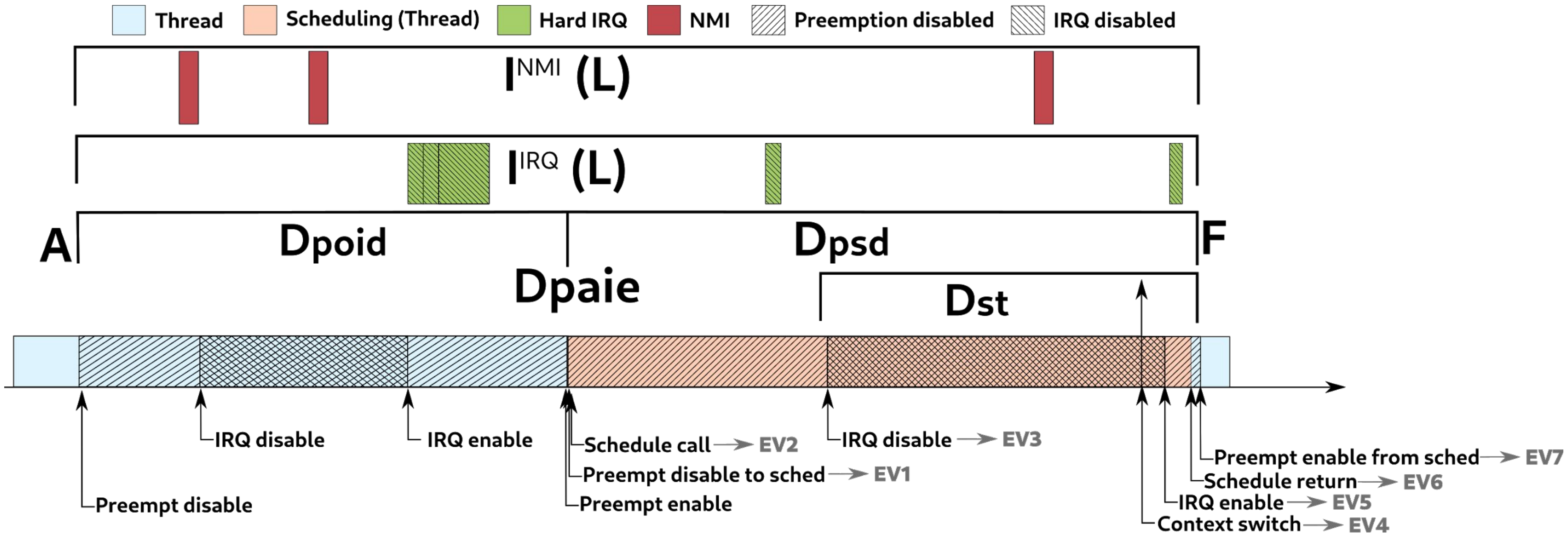


Red Hat

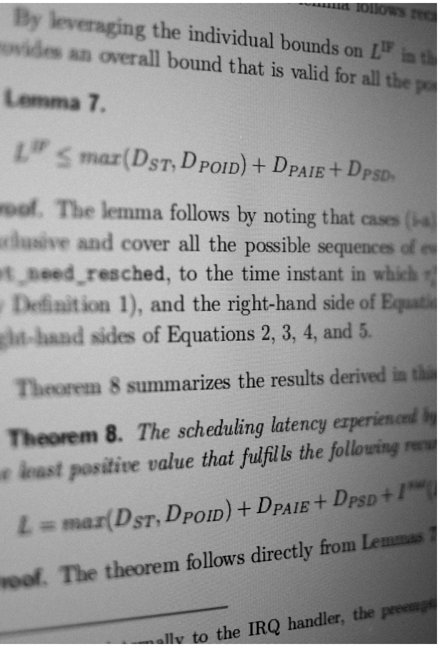
Timeline and cases

IRQ and NMI interference





And the *scheduling latency* bounds to:



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

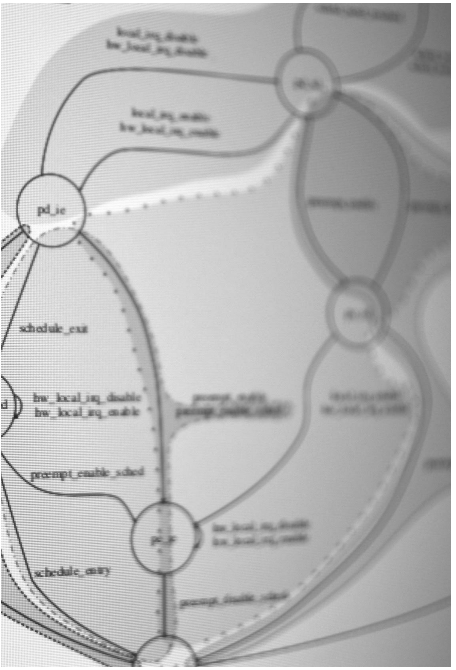
The bound considers all possible cases. Note that the Latency *L* is present in both sides of the equation.

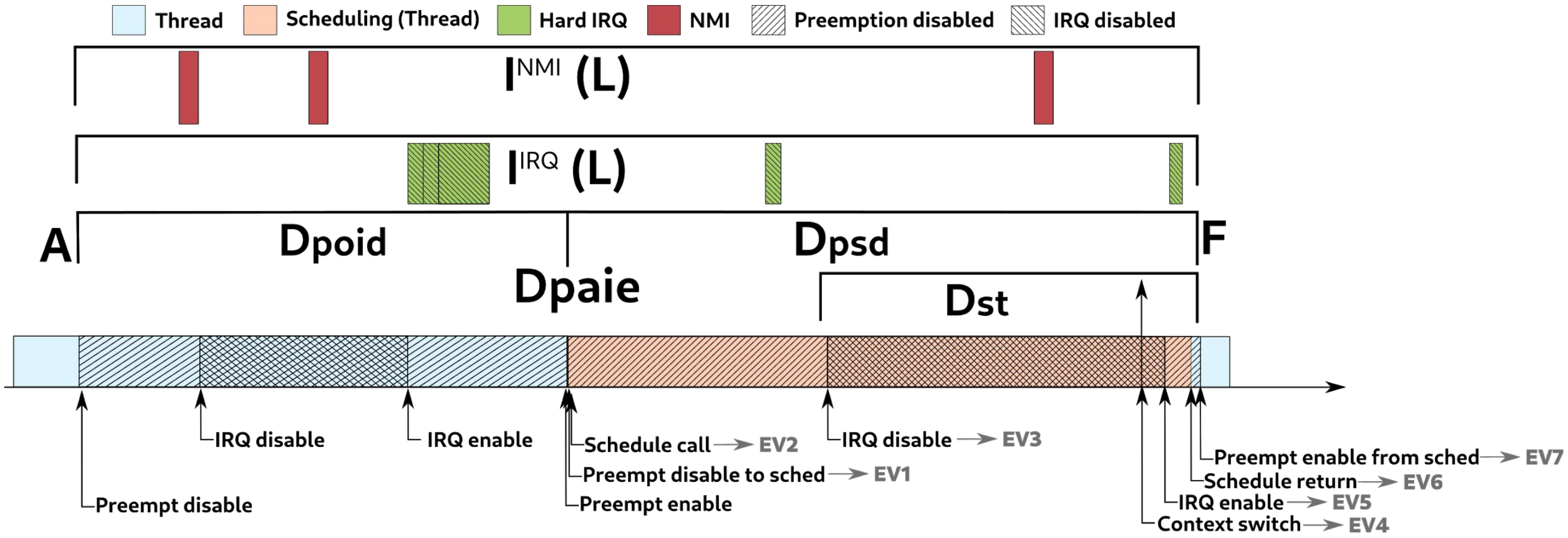
So, L is bounded by the least positive value fulfilling the equation (like on RTA).

Red Hat

Timeline and cases

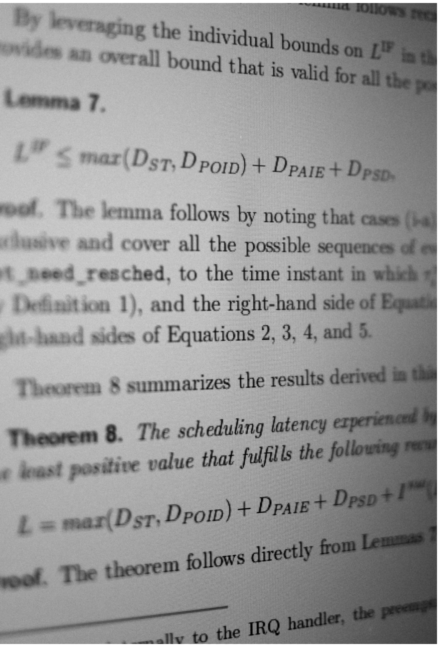
IRQ and NMI interference





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

And the *scheduling latency* bounds to:

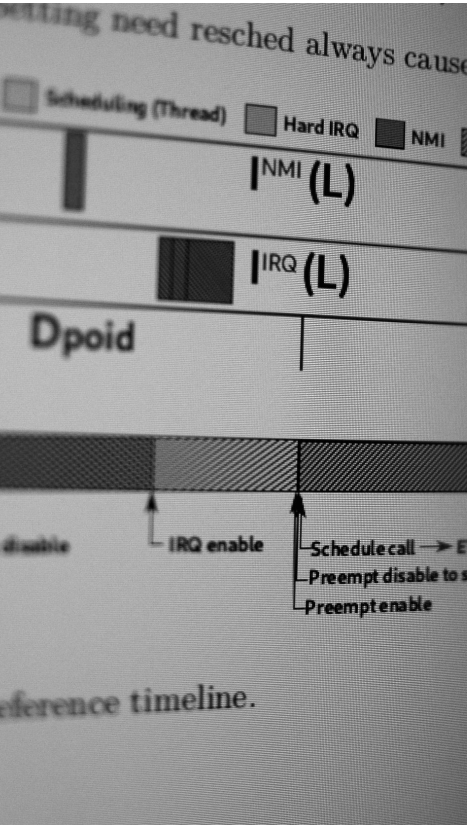


$$L = \max(D_{ST}, D_{POID}) + D_{PAIE} + D_{PSD} + I^{NMI}(L) + I^{IRQ}(L)$$

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).

Red Hat



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

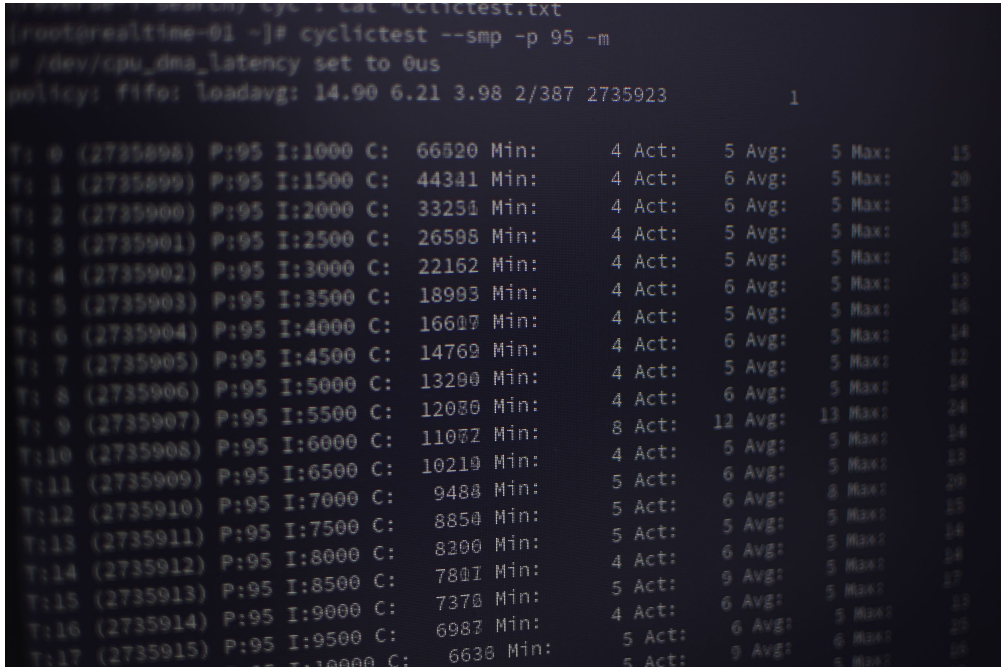
This topic was heavily discussed at the Real-time Micro Conference (inside Linux Plumbers) in 2019, more info here:

Episode II: getting practical (and efficient)

Red Hat

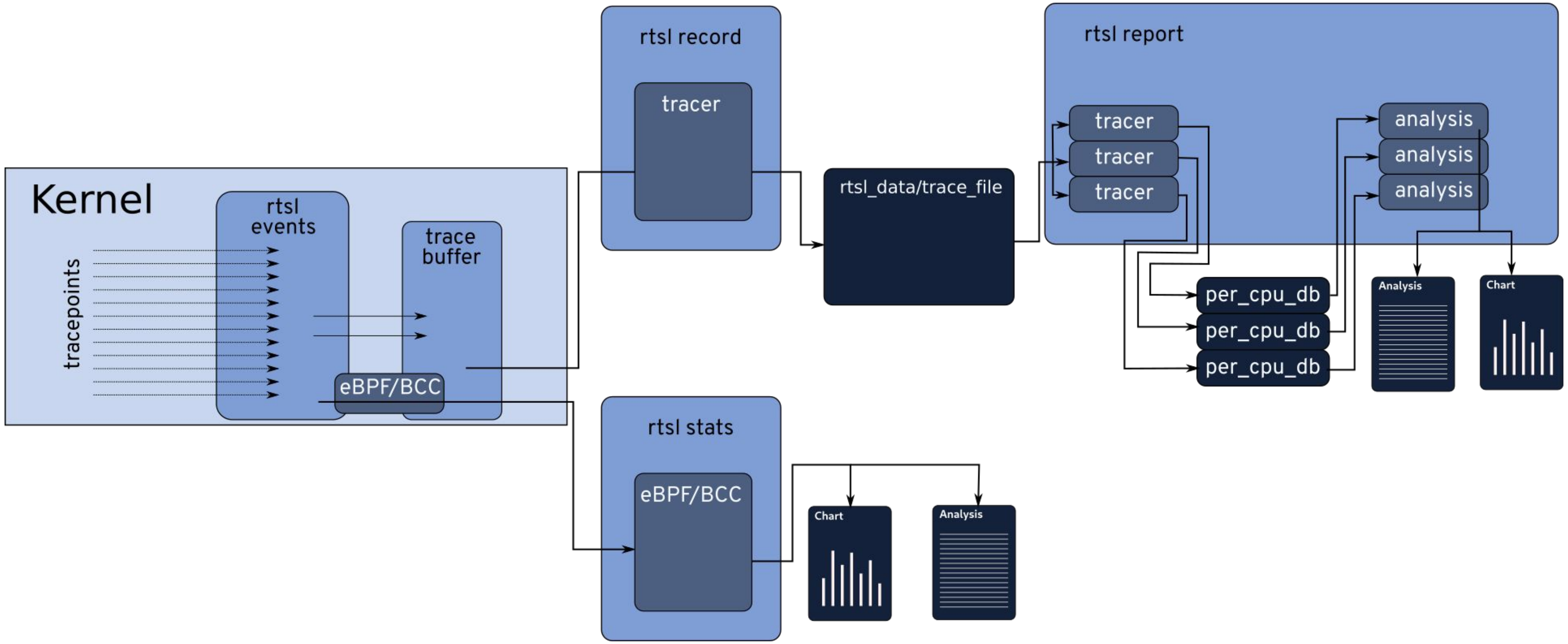
A practical scheduling latency estimation tool

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

Rtsl: a measurement tool



Kernel space:

- Rtsl events

User space:

- Rtsl command
  - Python

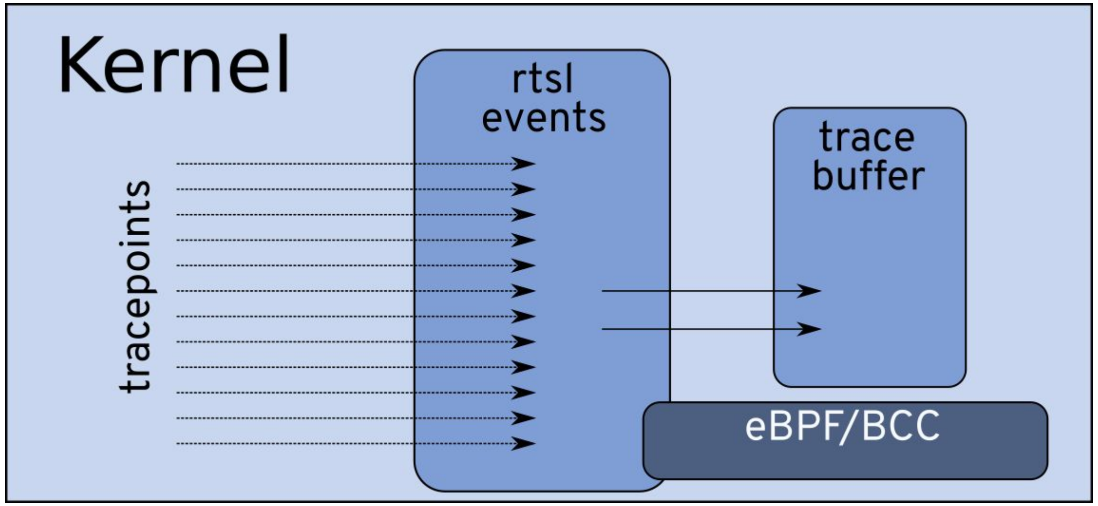
Has three commands:

- The **record** command saves the trace data;
- The **report** command process the trace and does the analysis.
- The **stats** command produces a histogram of the thread variables

Red Hat

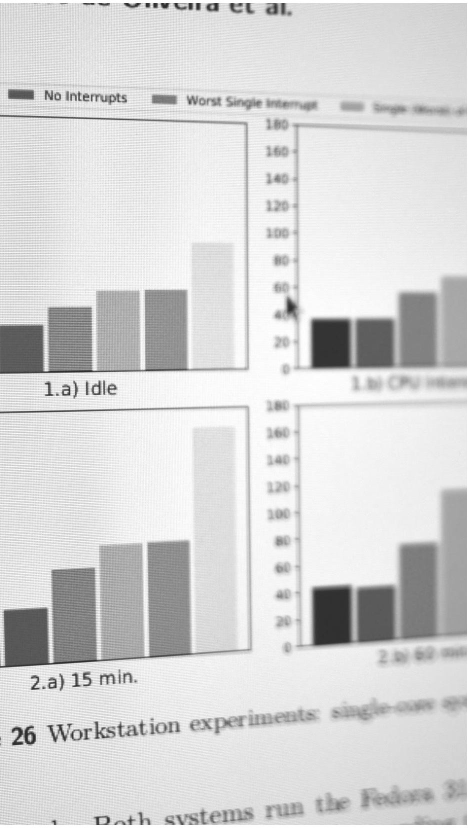
rtsl events

Low overhead tracer



- Hooks to events
  - Filters the high frequency trace
  - Doing in-kernel processing
  - Use a knob on debugfs to enable the tracing
- For blocking variables:
  - Reports all values or 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**

Kernel changes



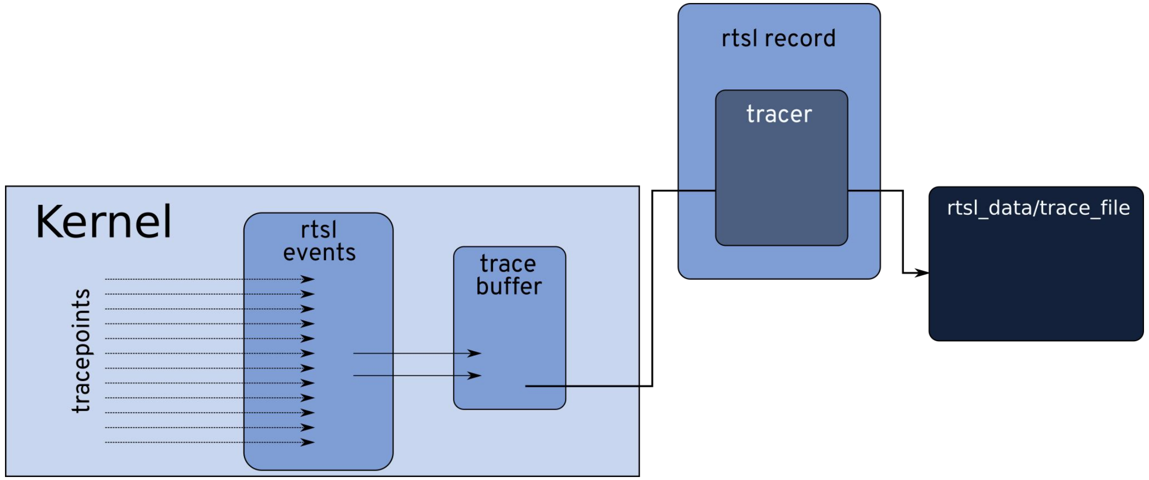
- The rtsl events depends on:
- **preemptirq tracepoints**
  - So it needs a “debug/trace” kernel (yeah…)
  - But life finds a way
- Annotations on the preempt_disable to sched
  - No functional changes
- NMI tracepoints
  - Or change in the current one to the extreme points of the handler

The parser was developed as a kernel module. In this way I can leave it off tree… but it would be better to have it in.

If we get it in, we can change the debugfs for the tracefs.

rtsl record

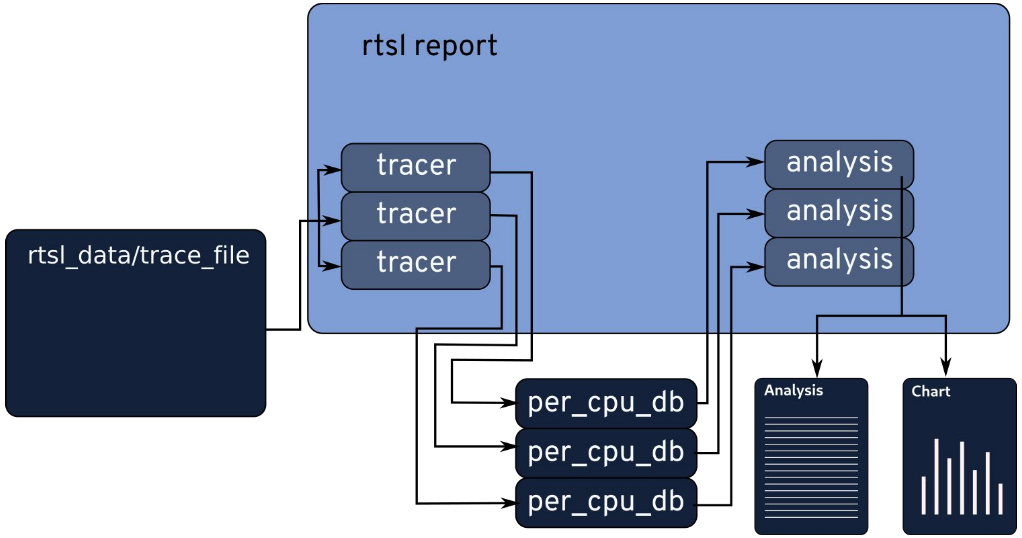
Trace recording



- Captures the values for the variables
  - Only new max values for thread variables
  - Saving them into a trace file
- Calls real tracers to do the tracing:
  - Perf
  - Ftrace
- Controls the trace section
- Saves the trace in the rtsl_data/ dir

rtsl report

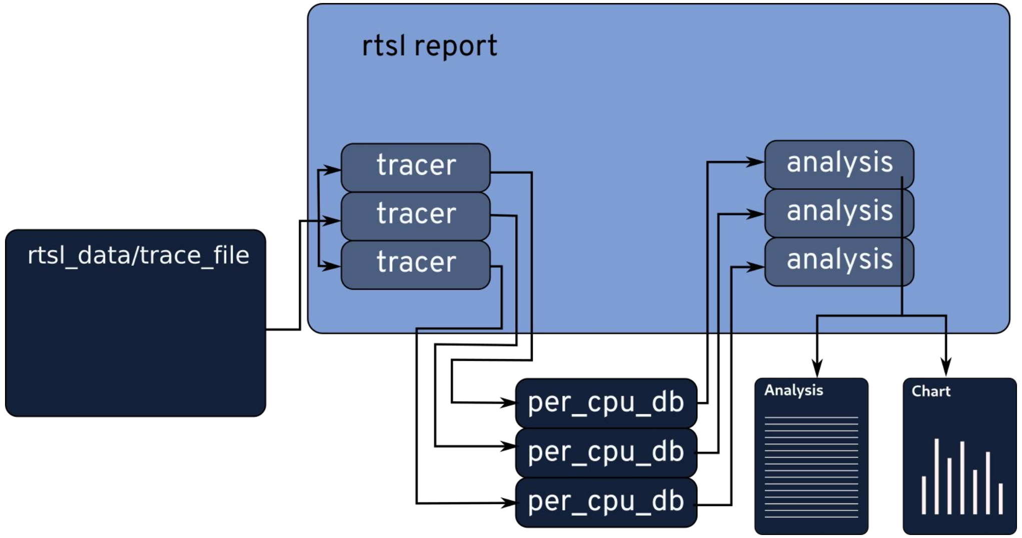
Trace processing



- Analyzes the trace!
  - All in user-space
- Most of the tool is done in python
  - Easy to extend the analysis (researchers like)
- Parses the trace file in parallel
  - Per cpu trace parsing (e.g., perf script-c $...)
  - Generates per-cpu database with the data
    - In the rtsl_data/ dir
  - Uses a C trace-plugin create the database
- Database in a sqlite3 file

rtsl report

Data processing



- The analysis is done on the database
  - IRQ analysis needs to read data back and forth
  - Trace can reach tens of GB/per-cpu
- The analysis is done in parallel
- Two outputs:
  - Textual output
  - Charts
    - Using matplotlib

rtsl report output

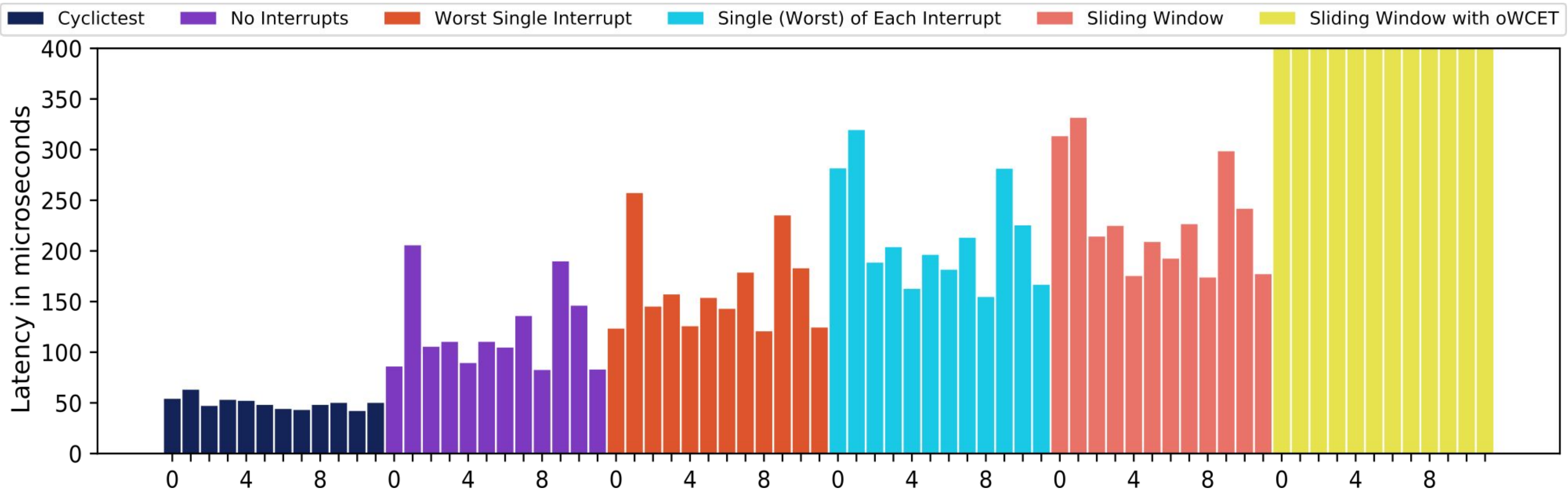
Textual output

```
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
```

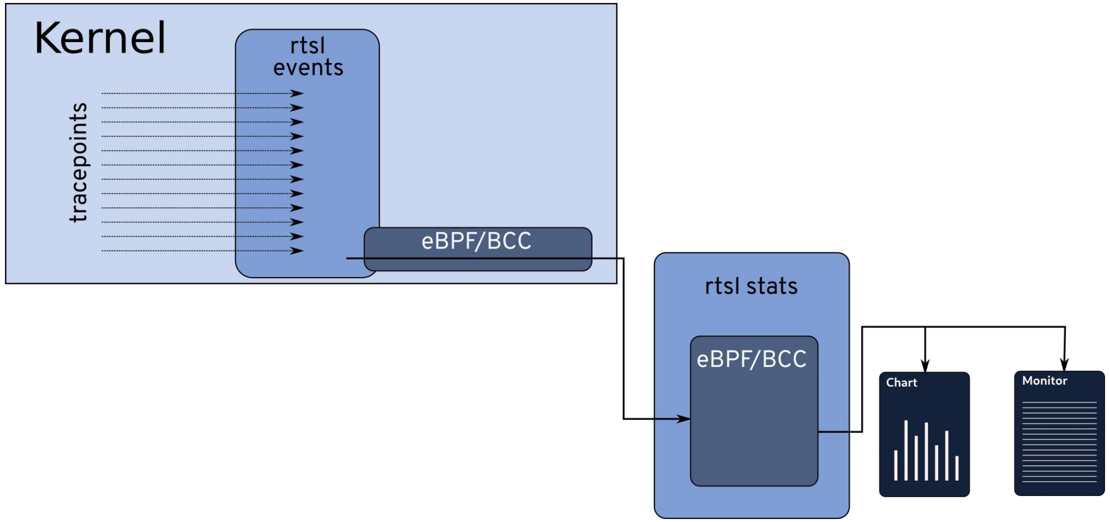
rtsl report output

Chart output



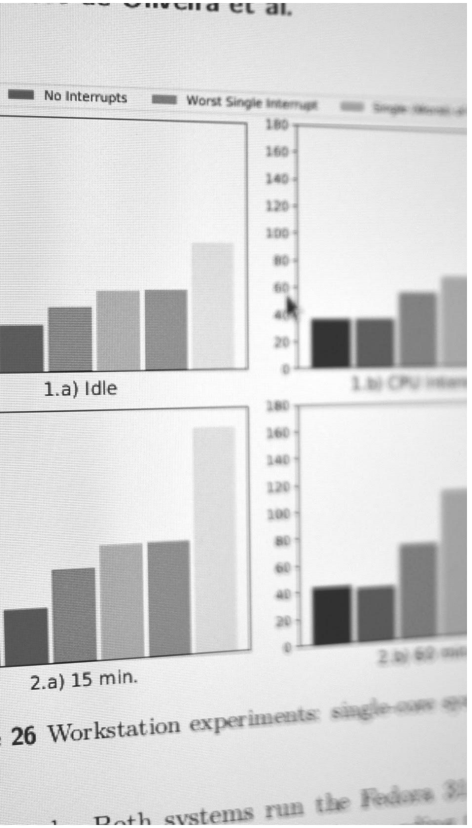
rtsl stats

Online view



- Monitor the thread variables
  - poid/psd/dst/paie...
- Uses BCC
  - Saves histograms in kernel
  - Display in user-space
  - Can plot data

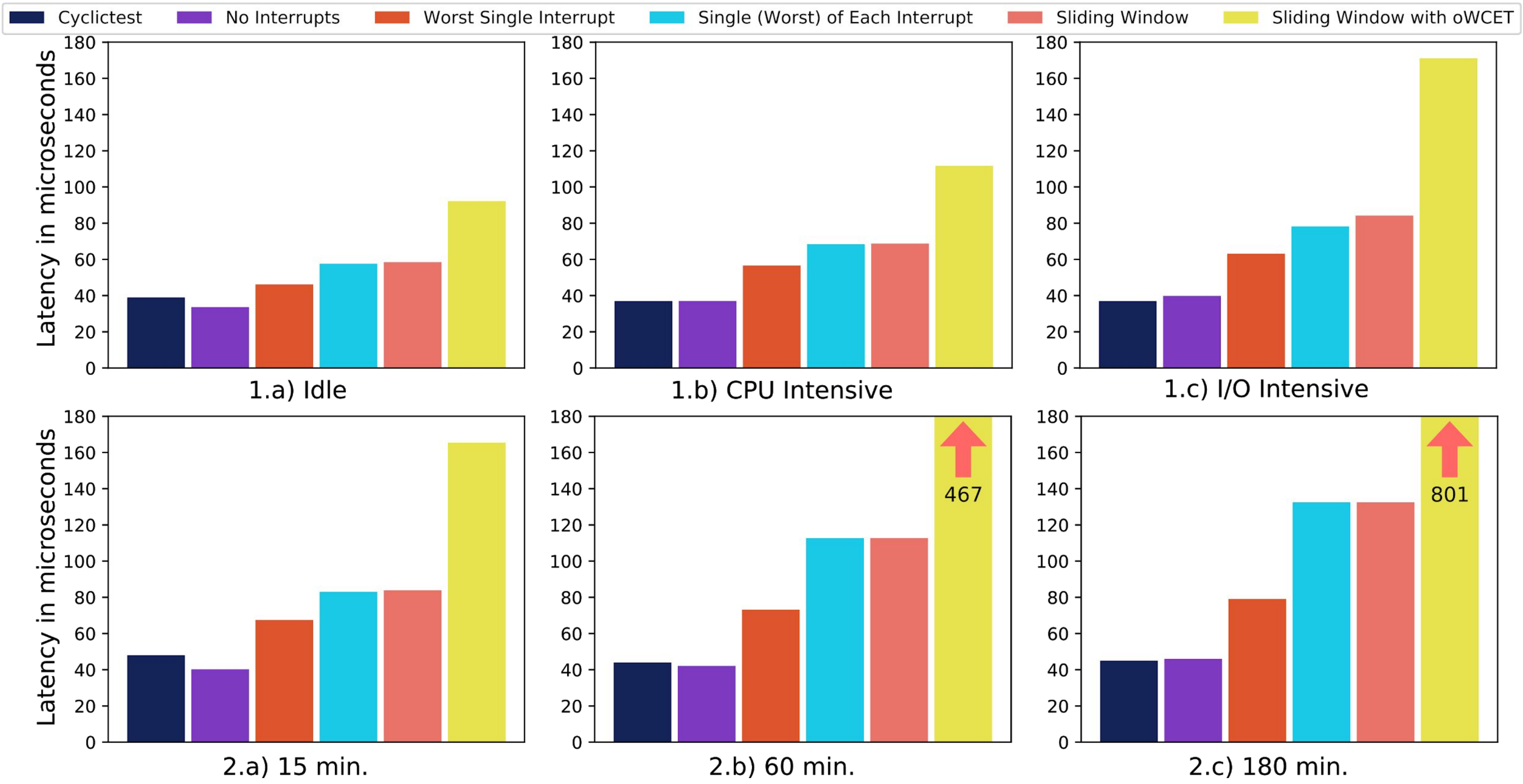
Experiments



- 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.

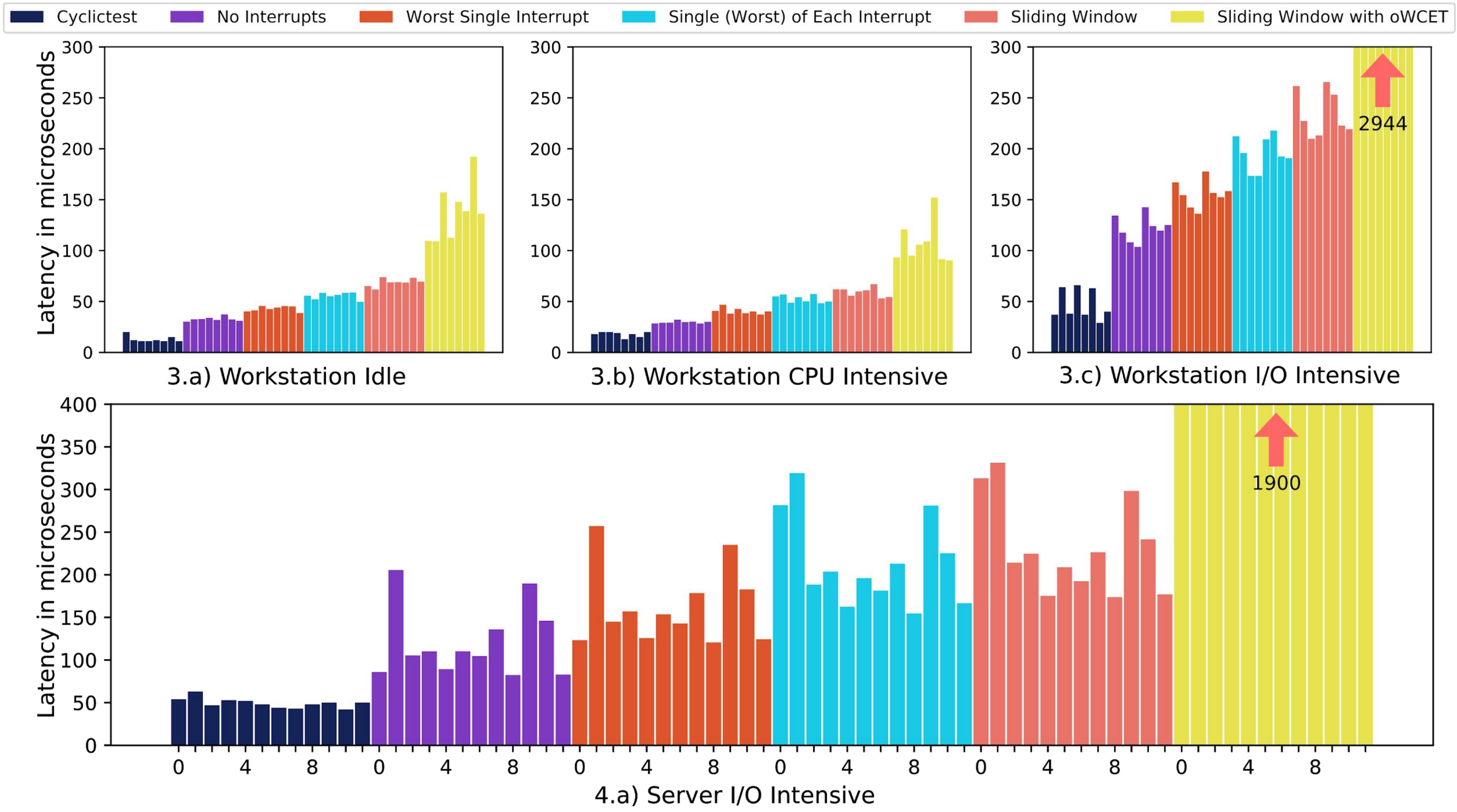
The experiments passed by the artifact evaluation!

Single-core experiments

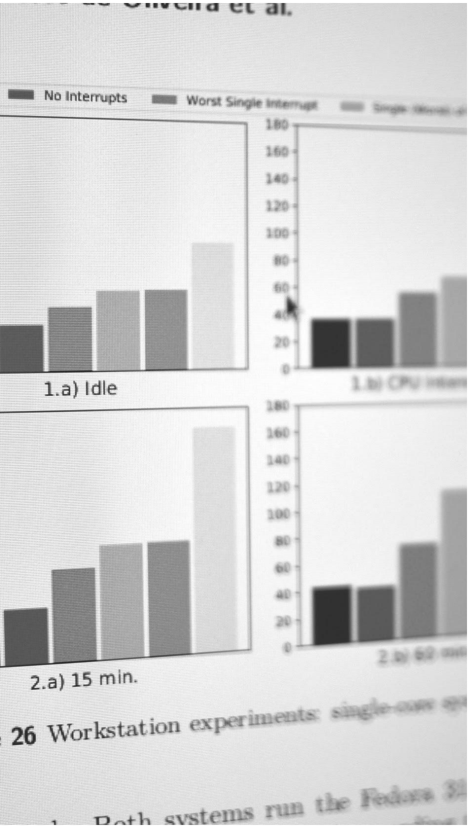


Red Hat

Multicore experiments



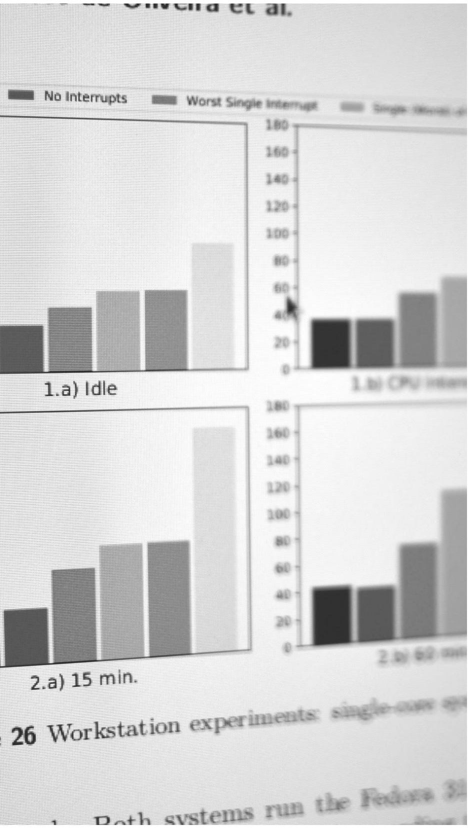
Remarks



- The PREEMPT_RT preemption model is deterministic, and the scheduling latency is bounded.
- The approach presented in the paper 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 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 this paper, like source code, other comments, Q&A, check its companion page!

rtsl vs cyclictest? nah



- They help the same people
  - But they do different things
- rtls is a more specific tool
  - Covers a single aspect: sched latency
    - Covers all cases at synchronization level
      - In the worst condition, even those that happened at different points in time.
    - With strong arguments
  - Depends on kernel features (PREEMPT_RT/preemptirq...)
- cyclictest is a more generic tool
  - Covers many aspects: external activation of the timer
    - Hardware delays? Hardware bugs?
    - Without analysis - a closed-box test
  - Run on the potato that runs Linux
- rtsl adds only 4-ish us of overhead on cyclictest

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

Red Hat

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

In the next episode....

Red Hat