Red Hat
Enterprise Linux

Sant'Anna
School of Advanced Studies – Pisa

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Efficient Runtime Verification for the Linux Kernel

Daniel Bristot de Oliveira

Principal Software Engineer @ Real-time & Scheduling team

Red Hat

Linux and Safety-critical

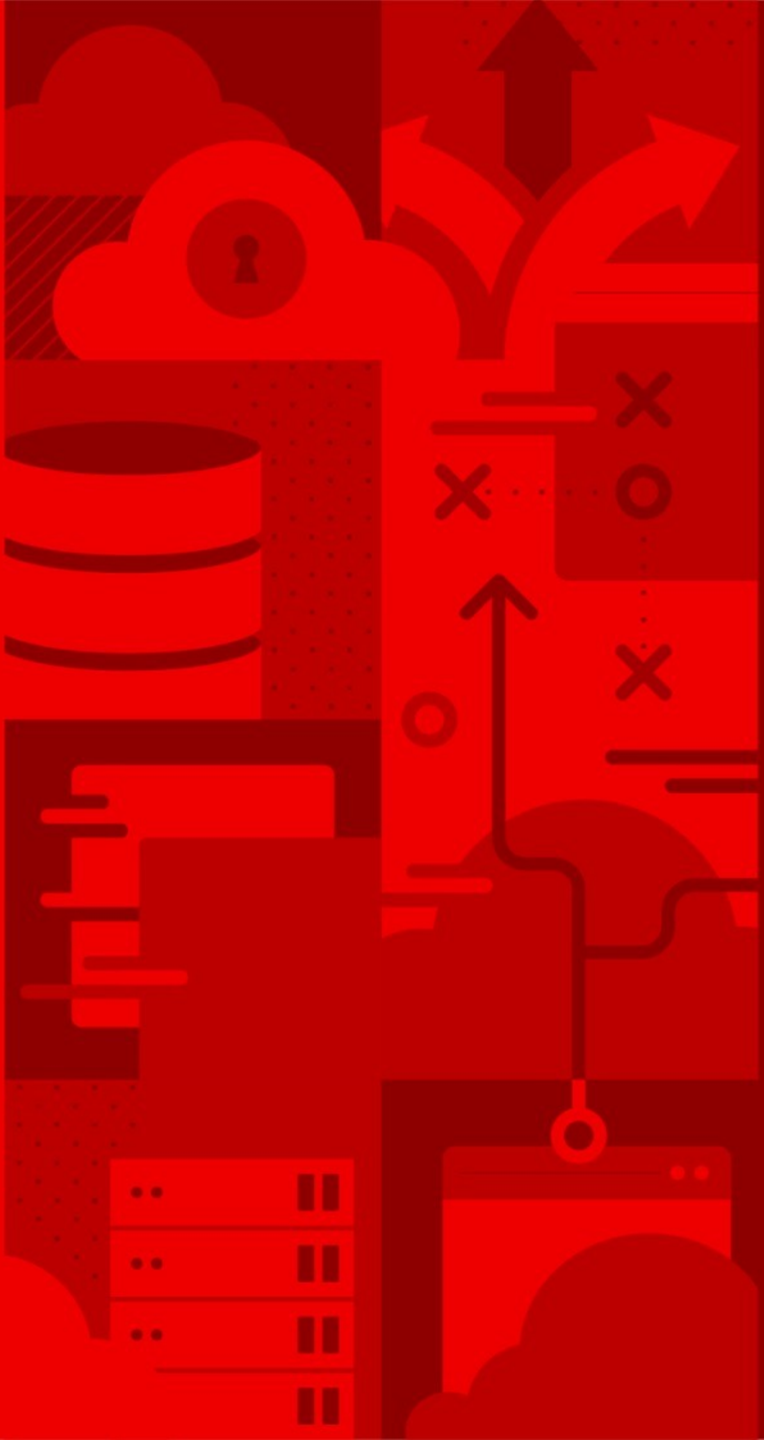
- Linux has been used on many **safety-critical** and/or **real-time systems:**
  - From **sensor networks** and **robotics,**
  - To control of military **drones** and **high-frequency trading** systems.
- Formal **verification of Linux is a non-negotiable requirement** for next-generation of cyber-physical systems, e.g., self-driven cars.

Linux and Formal methods

- Previous work using automata-based models has shown to be practical for Linux developers.
- Linux is already informally analyzed as a ***Discrete Event System*** by practitioners:
  - Understanding the ***OS as a state-machine is natural for OS developers***;
  - Also because of the rich tracing features already present on kernel.
- However, Linux lacks a methodology for runtime verification that can be applied broadly throughout all of the in-kernel subsystems, efficiently;
  - And the **Linux kernel community has shown the desire of exploring such possibility.**

In summary, I will

- **Present an efficient automata-based runtime verification method** for the Linux kernel:
  - Verifying the correct sequences of in-kernel events as happening at runtime, against an automata-based model that has been previously created.
- Present an **automatic code generation tool** for automata-based models:
  - Takes the advantage of the in-kernel tracing infrastructure to dynamically enable runtime control of the verification.
- Present a performance evaluation of the verification method.



Background

Linux Tracing

- Linux has an advanced set of tracing methods, including:
  - Tracing of Kernel functions (call and return);
  - Tracepoints: specific points in the code;
  - Dynamic tracepoints: tracepoints dynamically added to a running kernel;
  - And more...
- Trace example:

```
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
```

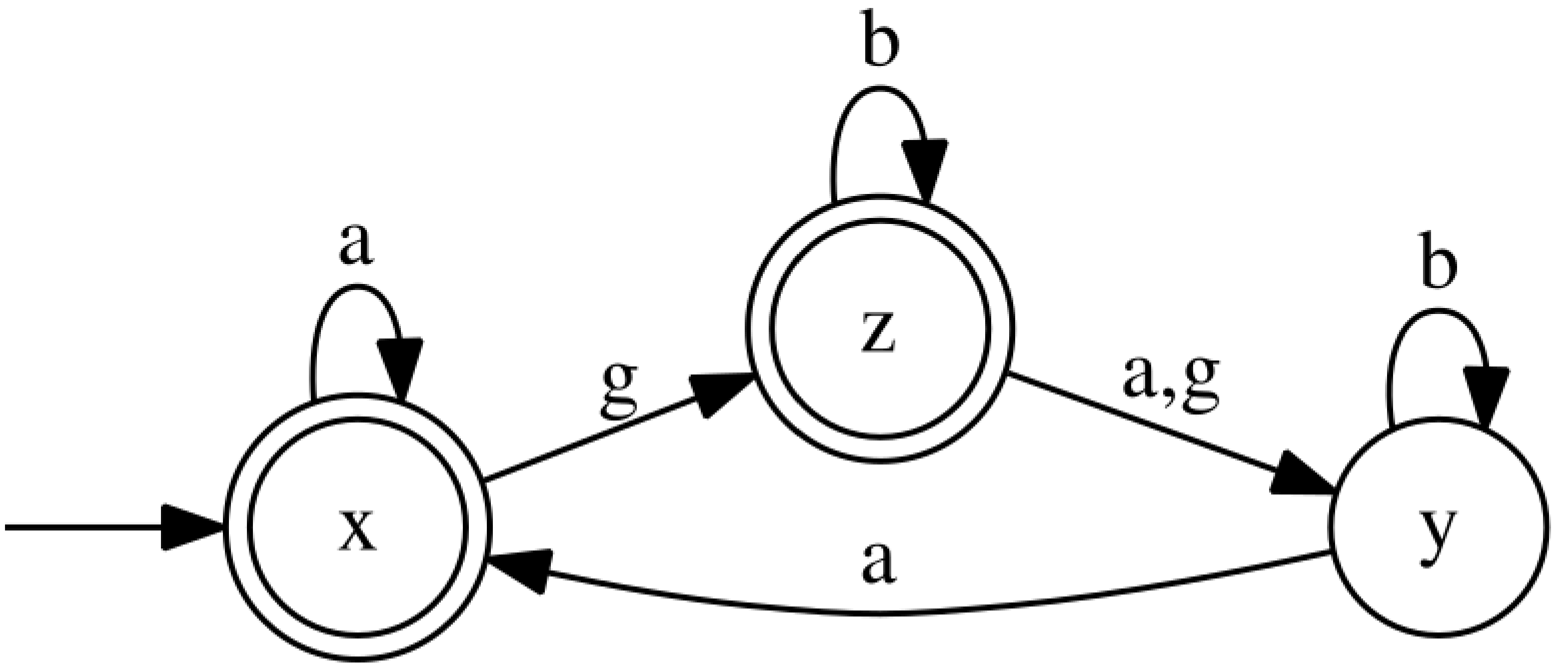
Linux Tracing

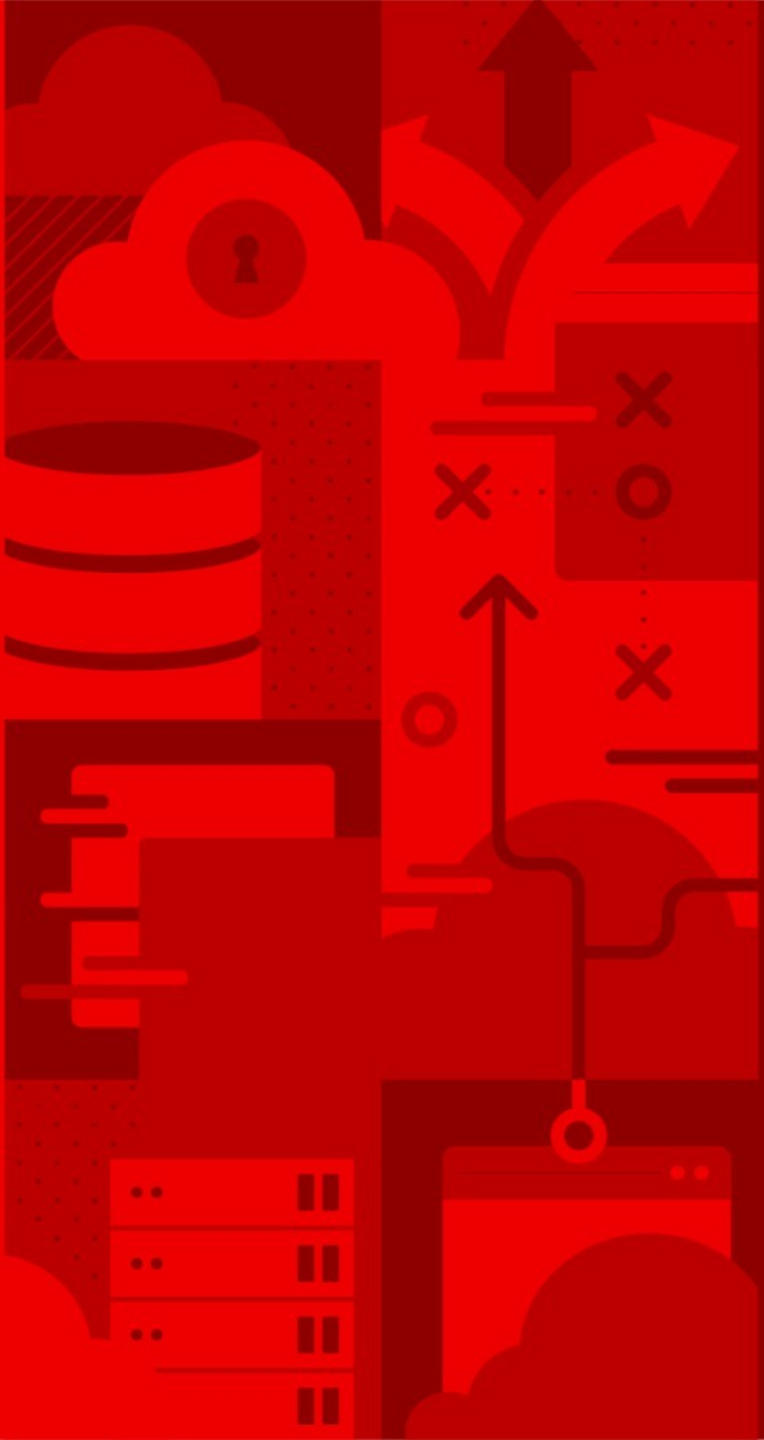
- **Tracing** code call **is not hard-coded** on Linux:
  - **Tracing calls are *no-op* in the binary:**
    - *Almost no overhead at runtime;*
    - *It is enabled on the vast majority of Linux distros.*
  - At runtime, when enabling tracing, **no-op are transformed into calls to trace functions.**
- More than one **tracer function can *hook* to an trace call**, dynamically:
  - **Live Patching** uses trace hooks to intercept a *bad function* call, deviating it to a *good one;*
  - Available for other methods as well, including modules.

Automata and DES

- Automata is a method to model **Discrete Event Systems** (DES)
  - Formally, an automaton is defined as:
    - $G = \{ X, E, f, x_0, X_m \}$, where:
      - X = finite set of states;
      - E = finite set of events;
      - f = transition function = $(X \times E) \rightarrow X$;
      - x_0 = Initial state;
      - X_m = set of final states.
- The language - or traces - generated/recognized by G is the L(G).

Automata and DES





Related work

FM and Linux Community

- **Lockdep** is in-kernel tool able to identify errors in the use of locking primitives that could eventually lead to deadlocks.
- **Linux Kernel Memory Consistency Model** (LKMM) subsystem, is an array of tools that formally describe the Linux memory consistency model, and also producing “litmus tests” in the form of kernel code that can be executed and tested directly.
- The **TLA+** formalism has been successfully applied to discover bugs in the Linux kernel. Examples:
  - Confirmed a bug w.r.t. the correctness of memory management locking during a context switch.
  - Bug w.r.t. fairness properties of the arm64 ticket spinlock implementation.

Automata and Linux

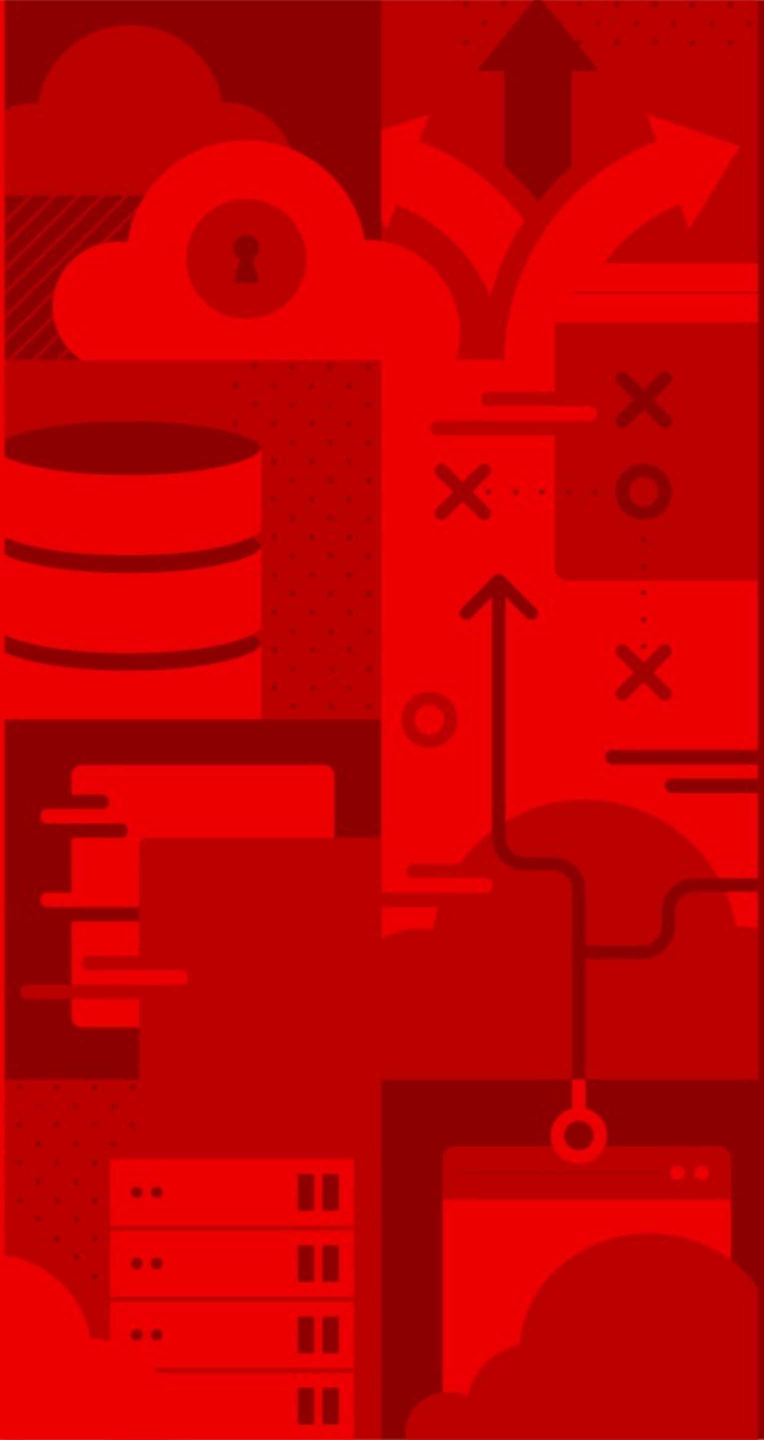
- **LTTng** [tracing tool] used to compare Linux execution against simple automata models.
  - Matni, Dagenais (2009)
- **SABRINE**: An approach using tracing and automata for *state-aware robustness testing* of OSes.
  - Trace are transformed into automata to group function into a state.
  - Cotroneo, Leo, Fucci, Natella (2013)
- **TIMEOUT** extends **SABRINE** with timing information for RTOS
  - Shahpasand, Sedaghat, Paydar (2016)

PREEMPT_RT Model

- **PREEMPT_RT Model** is an automata model that describe the interaction of the synchronization mechanisms and scheduling for threads, IRQs and NMIs.
  - Aiming to formally describe the dynamics of Real-time Linux.
  - de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S.: **Untangling the Intricacies of Thread Synchronization in the PREEMPT RT Linux Kernel**. In: Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). Valencia, Spain (May 2019)
  - > 9k states and & > 23k transitions
- During the development, we found 4 bugs in kernel
  - 3 of them that could not be detected by any other tool

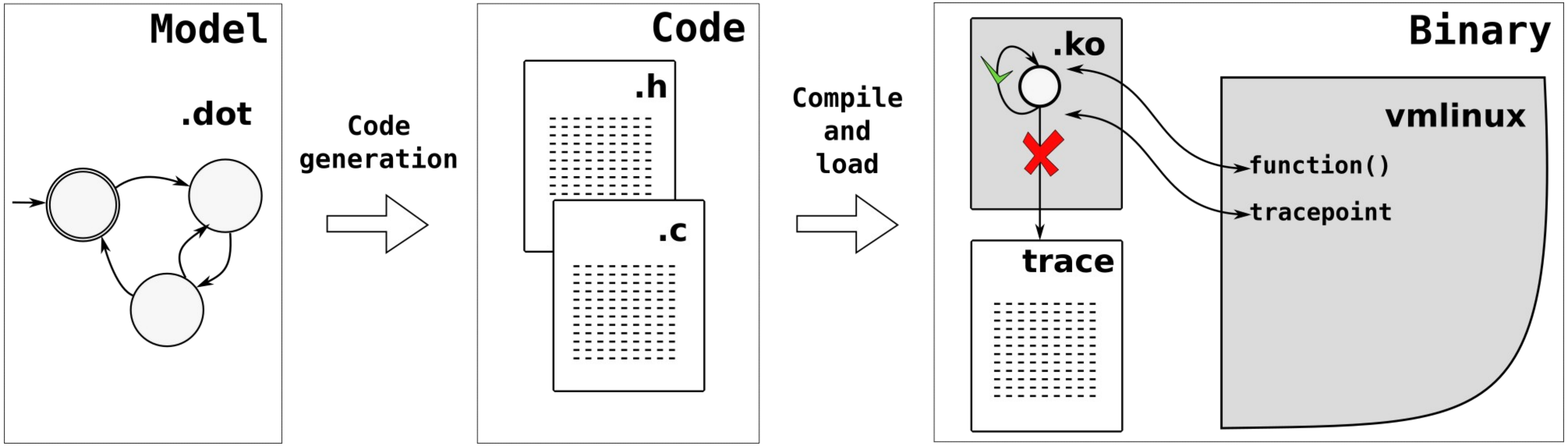
PREEMPT_RT Model

- **Linux kernel community** found value in the model for discovering bugs
  - **Automata seems to be a good abstraction** because of **tracing**
- The limitation of previous work:
  - The verification was done in user-space;
  - Required the transfer of a considerable amount of data from kernel to user-space
    - 30 seconds of trace generates 2.5 GB of data/CPU!
    - No in-kernel actions could be taken in the case of an unexpected event;
      - e.g., stacktrace, create a crash dump...

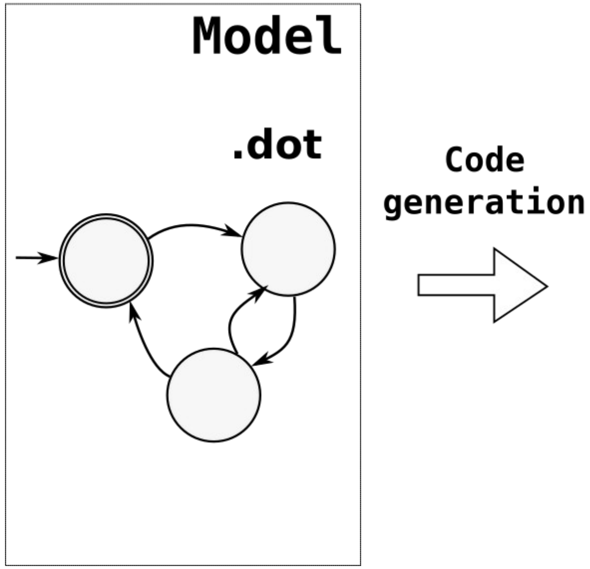


Efficient automata verification for the Linux Kernel

Proposed approach



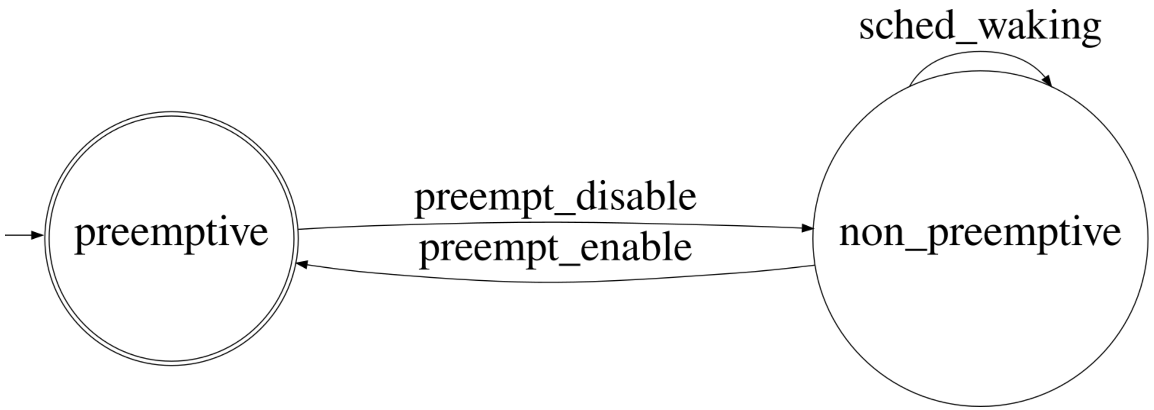
1) Code generation



- We develop the **dot2c** tool to translate the model into code
- 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

Wakeup in preemptive model:

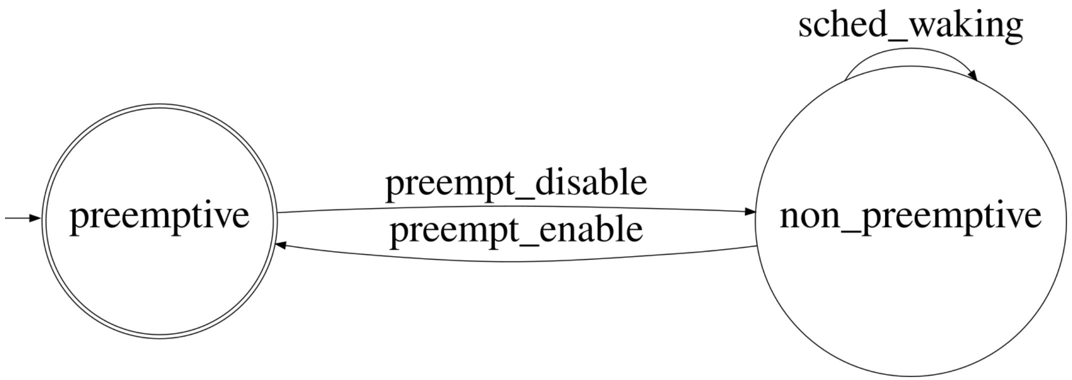


Code generation:

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

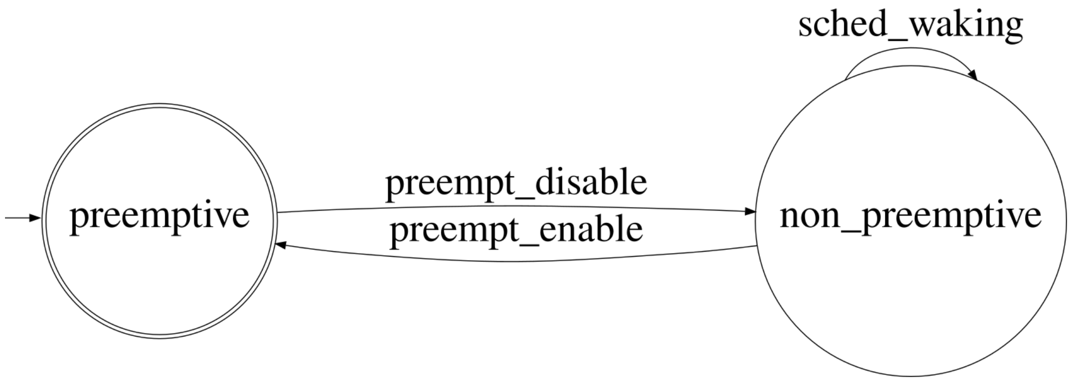
Automaton in C

```
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];
};
```

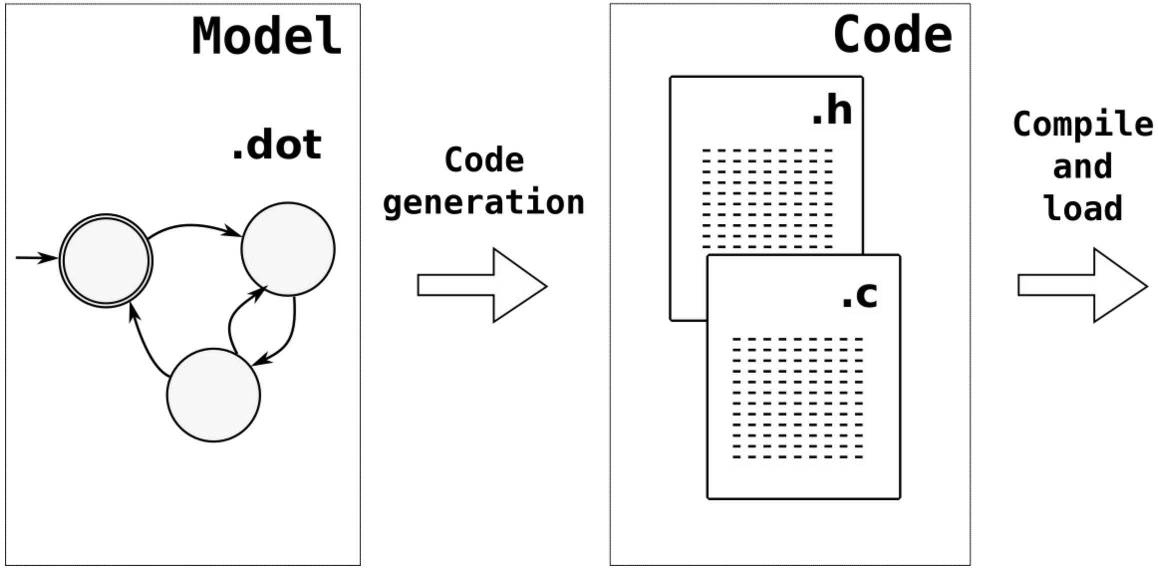


Automaton in C

```
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 }
};
```



Processing functions



Processing one 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;
}
```

Processing one event

```
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;
}
```

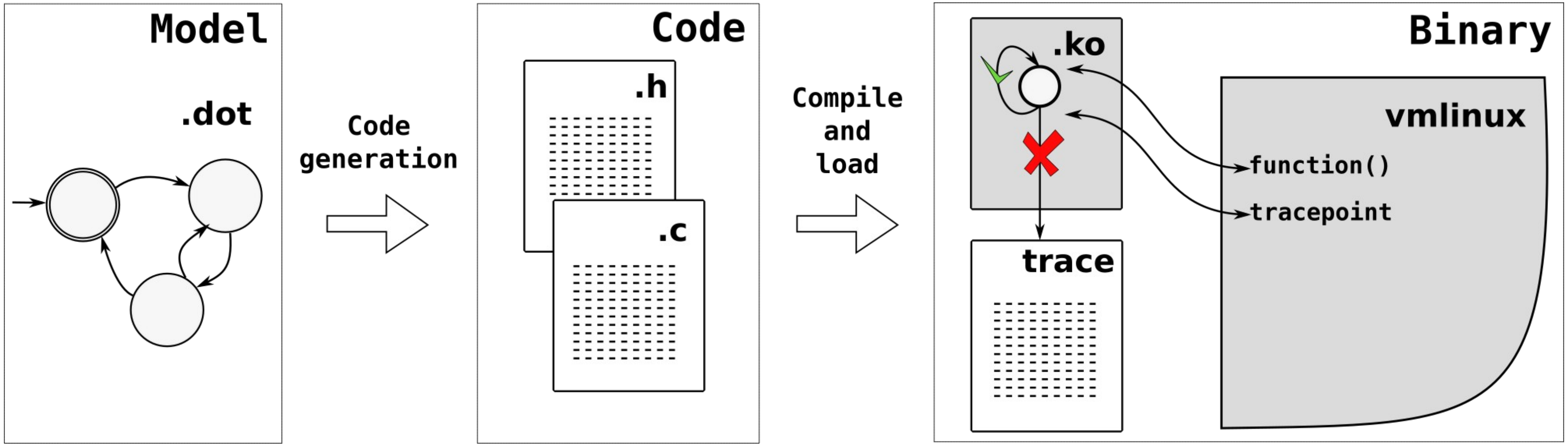
Processing one event

```
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)!

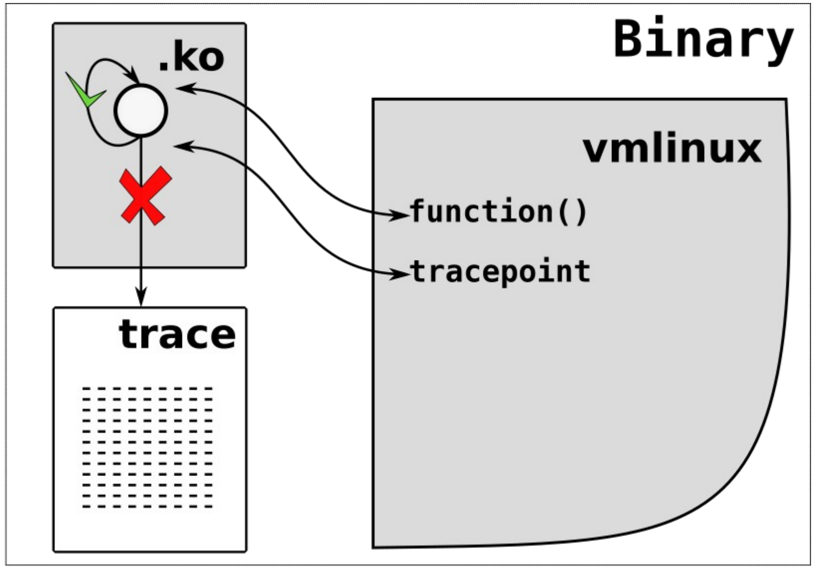
Only one variable to keep the state!

3) Verification



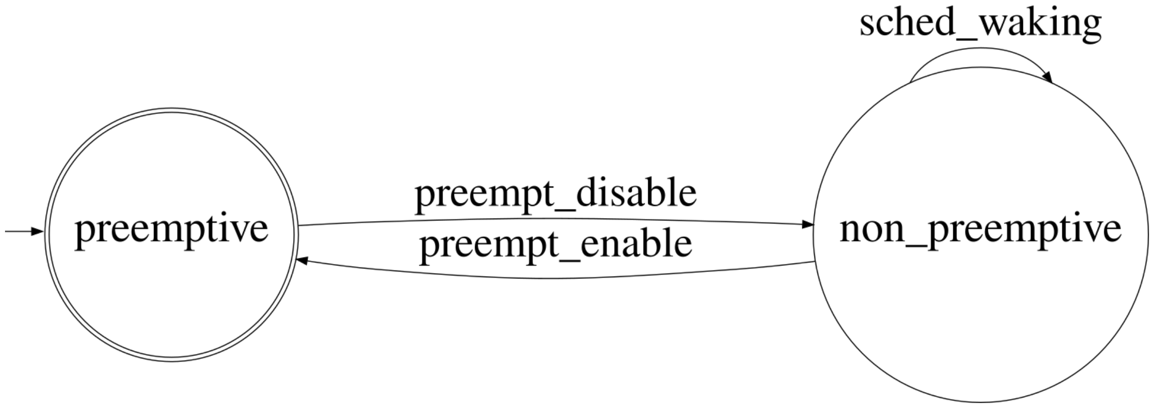
Verification

- Verification code is compiled as a kernel module
- Kernel module is loaded to a running kernel
  - While no problem is found:
    - Either print all event's execution;
    - Or run silently.
- If an unexpected transitions is found:
  - Print the error on trace buffer.



Error output

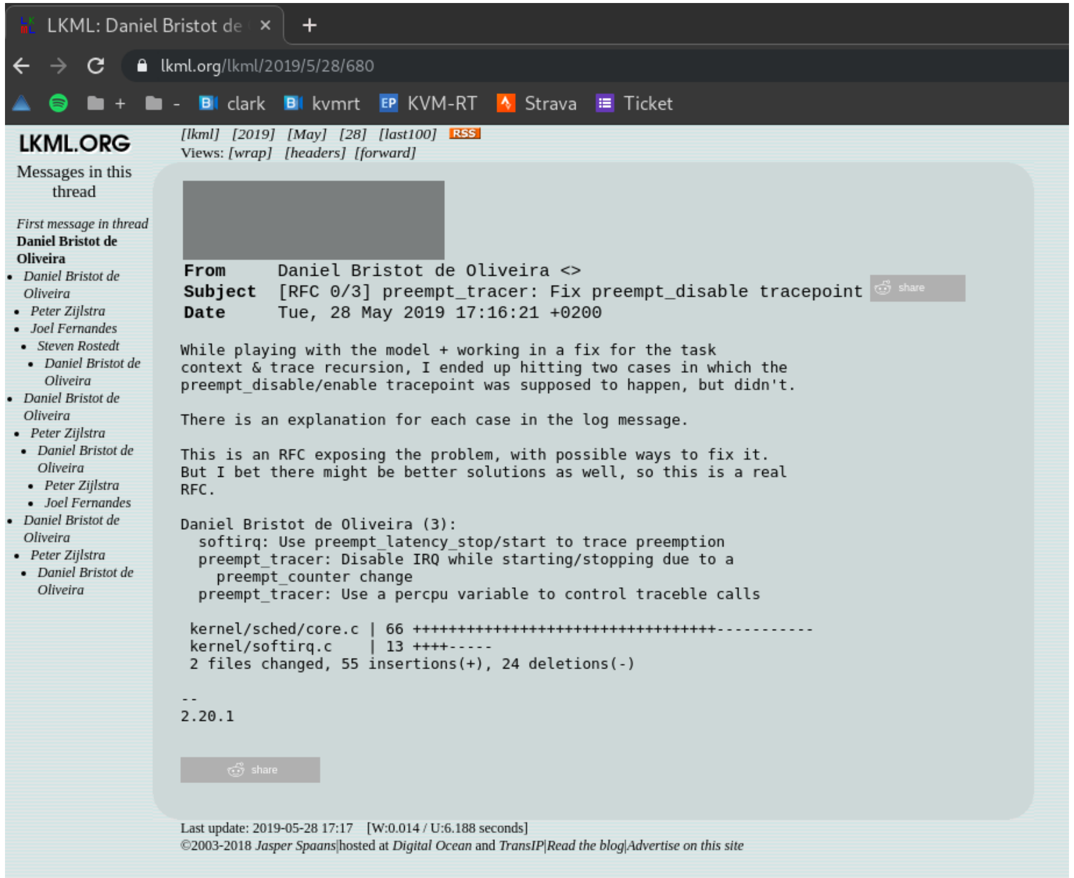
```
bash-1157  [003] ....2..   191.199172: process_event: non_preemptive -> preempt_enable = preemptive safe!
bash-1157  [003] dN..5..   191.199182: process_event: event sched_waking not expected in the state preemptive
bash-1157  [003] dN..5..   191.199186: <stack trace>
=> process_event
=> __handle_event
=> 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
```



Practical example

- A problem with tracing subsystem was reported using this model's module
  - https://lkml.org/lkml/2019/5/28/680

    <recall to open the link>



LKML.ORG

[lkml] [2019] [May] [28] [last100] RSS
Views: *[wrap] [headers] [forward]*

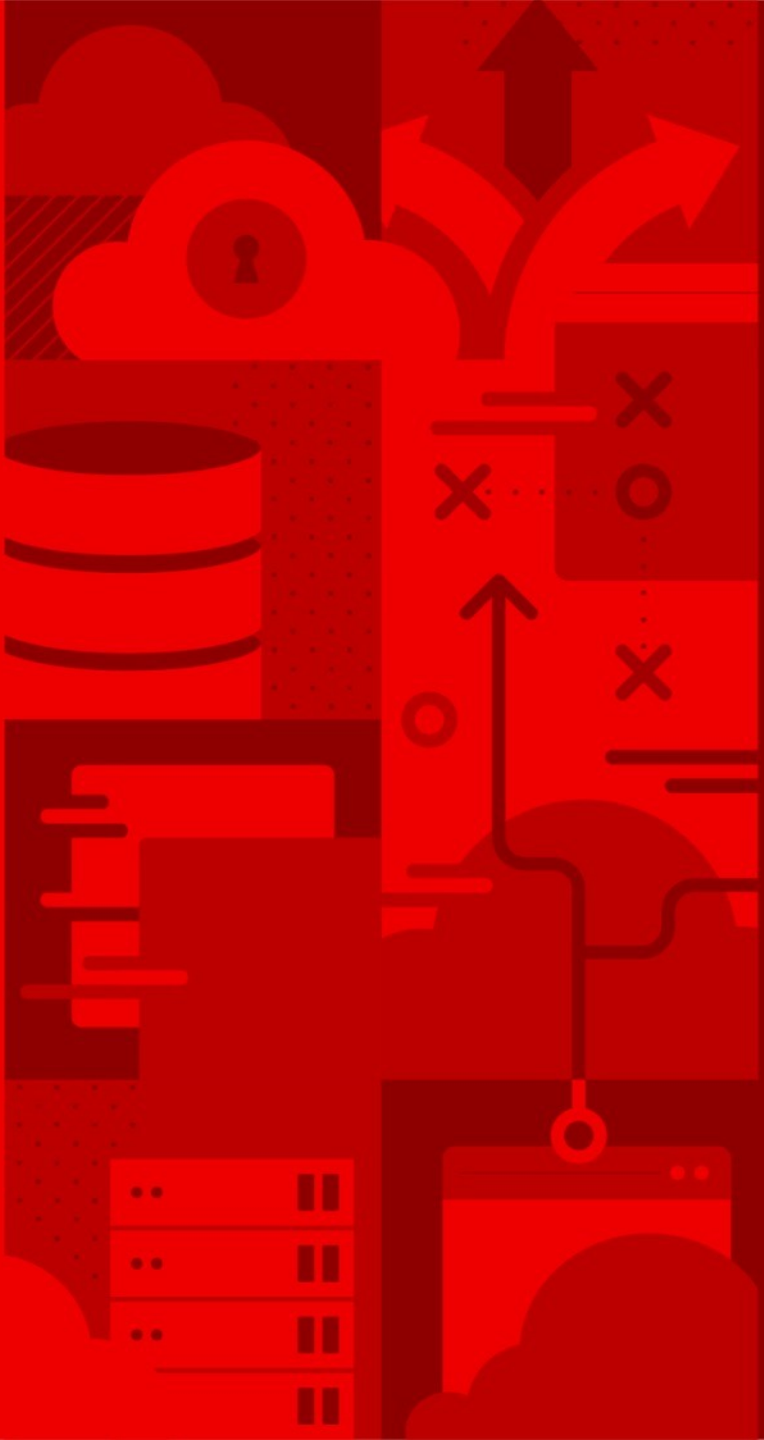
Messages in this thread

First message in thread
Daniel Bristot de Oliveira

- *Daniel Bristot de Oliveira*
  - *Peter Zijlstra*
  - *Joel Fernandes*
    - *Steven Rostedt*
      - *Daniel Bristot de Oliveira*
- *Daniel Bristot de Oliveira*
  - *Peter Zijlstra*
    - *Daniel Bristot de Oliveira*
      - *Peter Zijlstra*
      - *Joel Fernandes*
- *Daniel Bristot de Oliveira*
  - *Peter Zijlstra*
    - *Daniel Bristot de Oliveira*

```
From      Daniel Bristot de Oliveira <>
Subject   [RFC 0/3] preempt_tracer: Fix preempt_disable tracepoint
Date      Tue, 28 May 2019 17:16:21 +0200

While playing with the model + working in a fix for the task
context & trace recursion, I ended up hitting two cases in which the
preempt_disable/enable tracepoint was supposed to happen, but didn't.

There is an explanation for each case in the log message.

This is an RFC exposing the problem, with possible ways to fix it.
But I bet there might be better solutions as well, so this is a real
RFC.

Daniel Bristot de Oliveira (3):
  softirq: Use preempt_latency_stop/start to trace preemption
  preempt_tracer: Disable IRQ while starting/stopping due to a
    preempt_counter change
  preempt_tracer: Use a percpu variable to control traceble calls

 kernel/sched/core.c | 66 ++++++++++++++++++++++++++++++++++++++++----------
 kernel/softirq.c    | 13 ++++-----
 2 files changed, 55 insertions(+), 24 deletions(-)

--
2.20.1
```

Last update: 2019-05-28 17:17 [W:0.014 / U:6.188 seconds]
©2003-2018 *Jasper Spaans*|hosted at *Digital Ocean* and *TransIP*|*Read the blog*|*Advertise on this site*

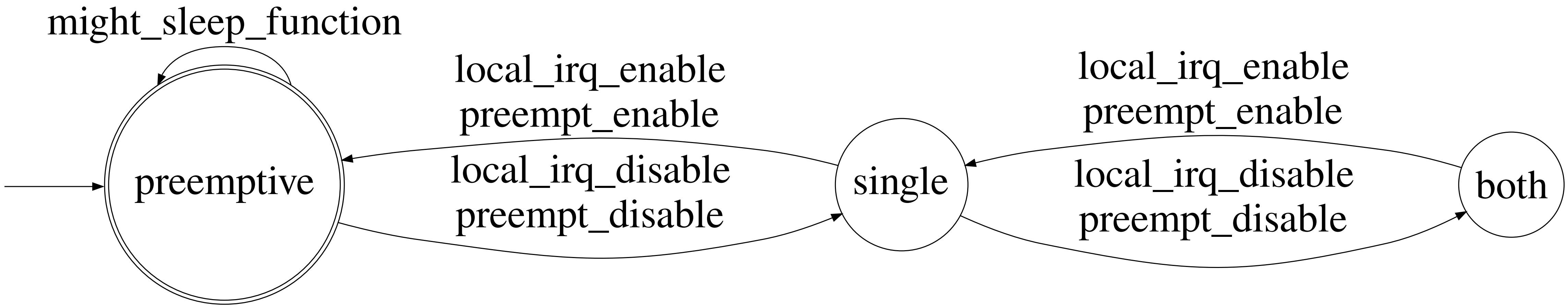


Performance evaluation

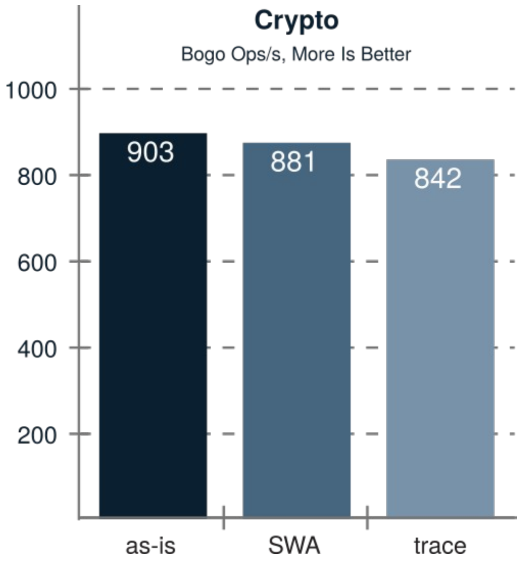
Efficiency in practice: a benchmark

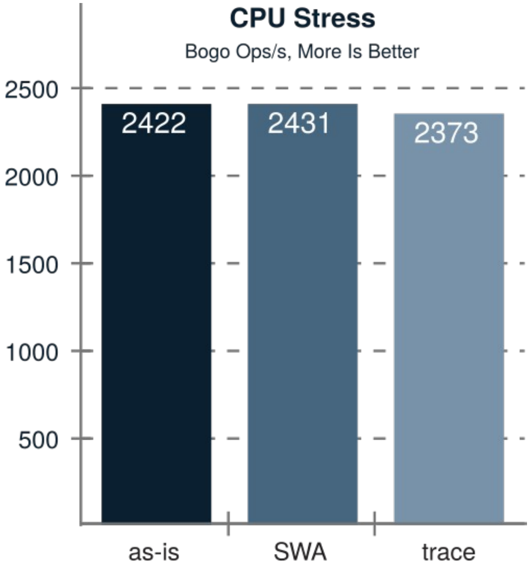
- Two benchmarks:
  - Throughput: Using the Phoronix Test Suite;
  - (highest prio thread wake-up) Latency: Using cyclictest.
- Base of comparison:
  - **as-is**: The system without any verification or trace;
  - **trace**: Tracing (ftrace) the same events used in the verification;
    - Only trace! No collection or interpretation.

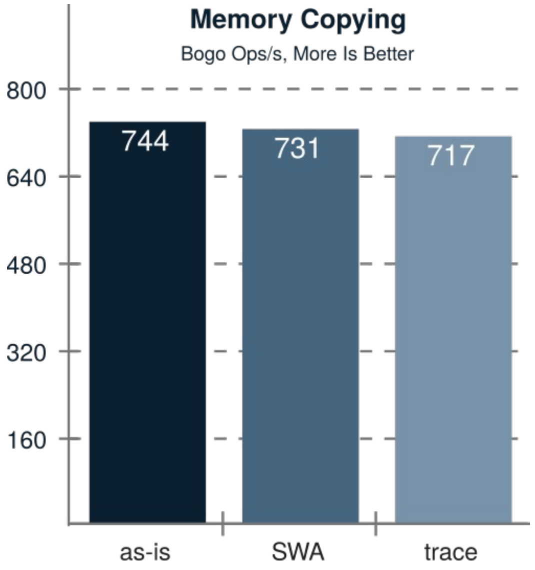
Throughput: SWA model



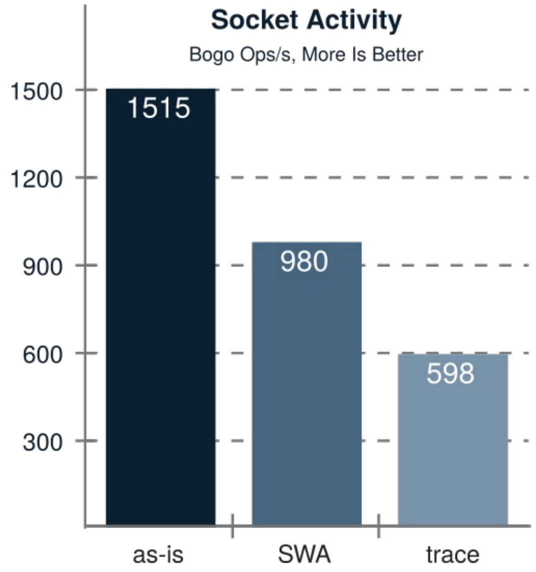
Benchmark: Thoughput – Low kernel activation

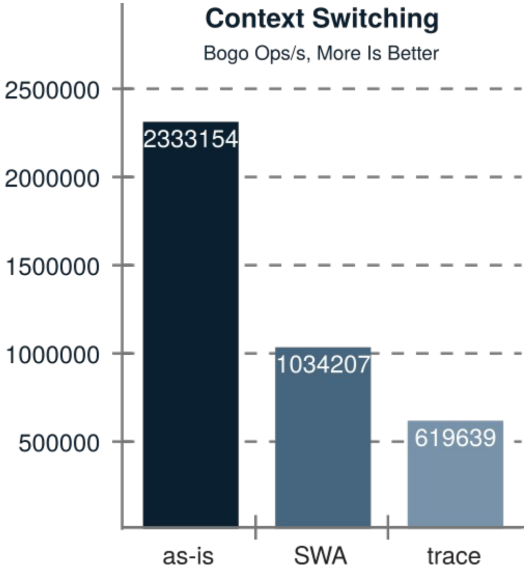


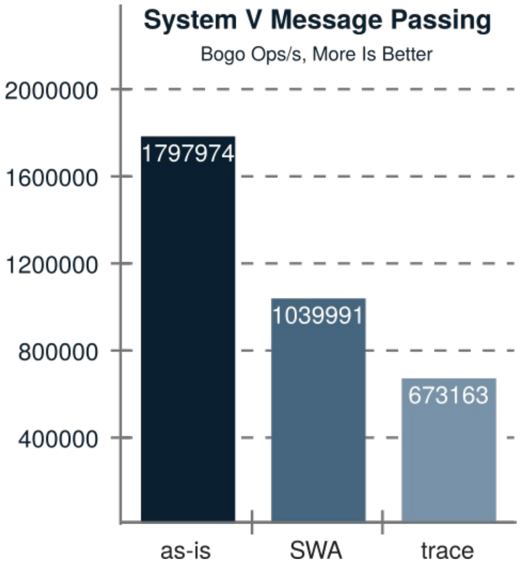




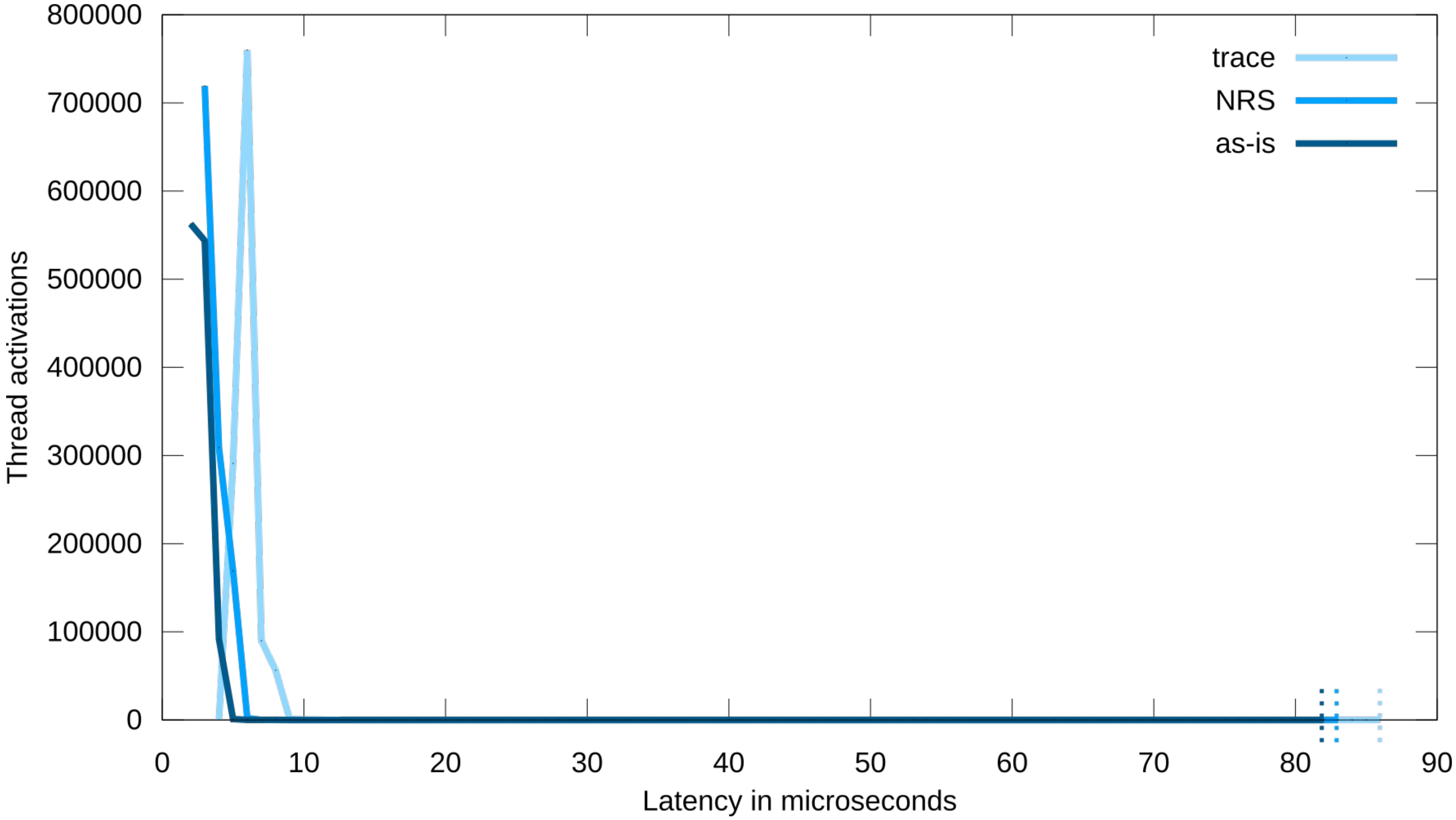
Benchmark: Thoughput – High kernel activation







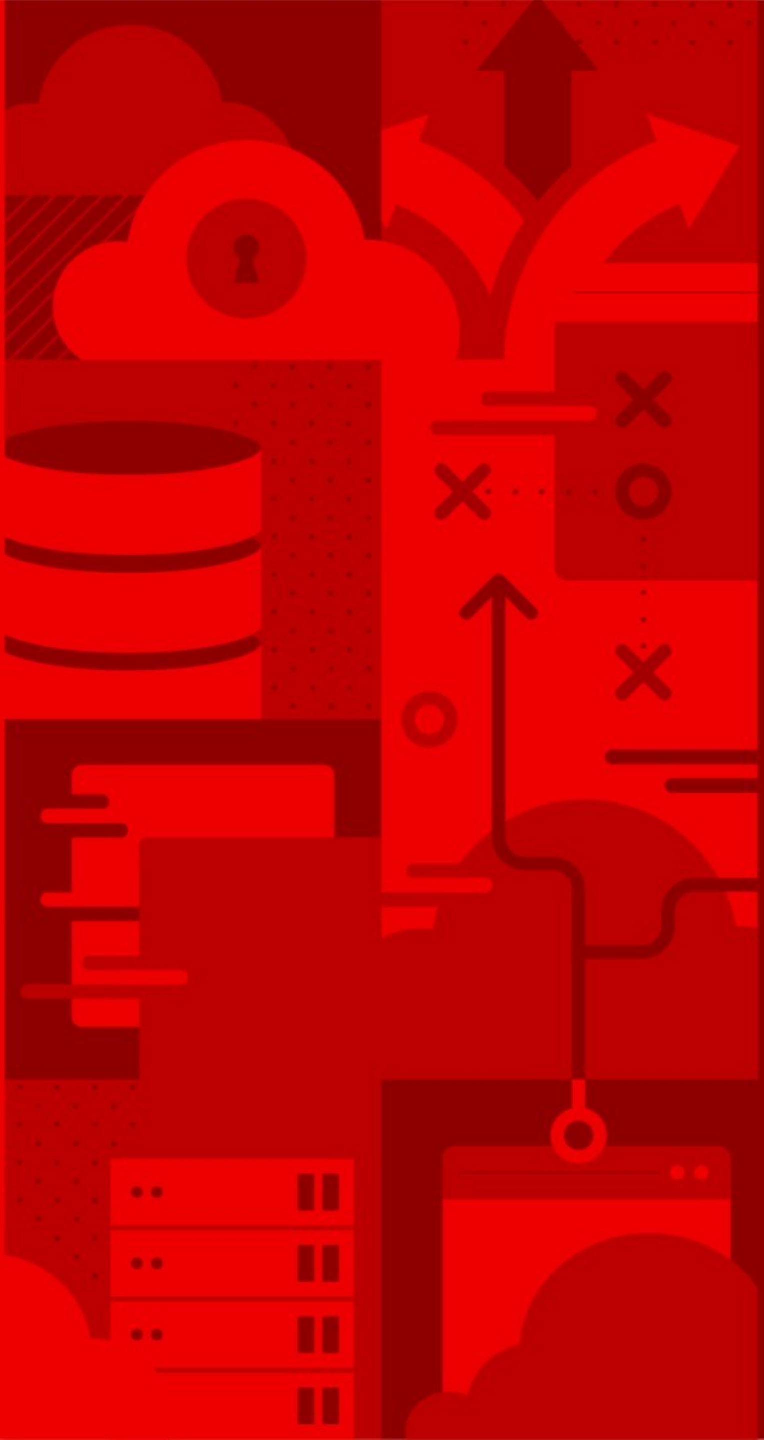
Benchmark: Cyclictest latency



Red Hat

What it means?

- **Trace is enable in production systems.**
- And is broadly used:
  - Hence, **the verification can be done in production;**
  - This is useful mainly for debugging problems:
    - Model the expected behavior;
    - Wait for an unexpected event to happen.



Future work

Future work

- Integrate the PREEMPT_RT model and this approach
- Create a better interface:
    - Having some models ready to be used;
- Creation of new models for the kernel:
    - We will try to model RCU;
- Working with Department of Information Security/CS @ ETH Zurich:
    - For integration of other RV methods, mainly involving TL and time.
- I've heard about people creating automatically generated models.

Further reading

Efficient Formal Verification for the Linux Kernel

Daniel Bristot de Oliveira, Rômulo Silva de Oliveira & Tommaso Cucinotta

17th International Conference on Software Engineering and Formal Methods

A Thread Synchronization Model for the PREEMPT_RT Linux Kernel

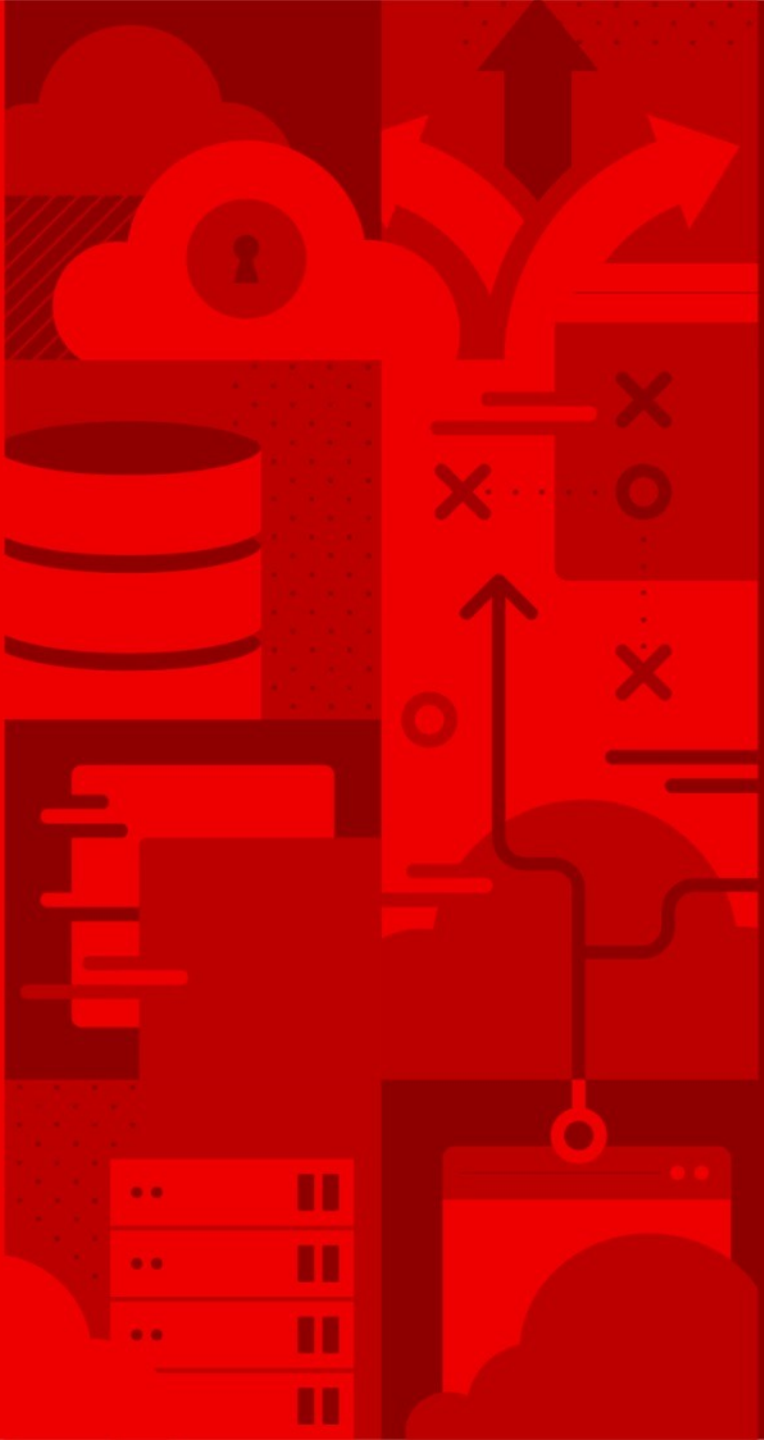
Daniel Bristot de Oliveira, Rômulo Silva de Oliveira & Tommaso Cucinotta

Accepted at the Journal of Systems Architecture

Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel

Daniel Bristot de Oliveira, Rômulo Silva de Oliveira & Tommaso Cucinotta

2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)



Questions?

Work in collaboration with:

- Retis Lab at the Scuola Superiore Sant’Anna, Italy; and
- Automation and Systems department at the Universidade Federal de Santa Catarina, Brazil.