8th Embedded Operating Systems Workshop (EWiLi 2018)

Modeling the Behavior of Threads in the PREEMPT RT Linux Kernel Using Automata

Daniel B. de Oliveira[1,2,3], Tommaso Cucinotta[2], Rômulo S. de Oliveira[3]
bristot@redhat.com, tommaso.cucinotta@santannapisa.it, romulo.deoliveira@ufsc.br,

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Introduction

- Real-time Linux has been a research topic for more than a decade
- But some conceptual divergence between academic real-time and real-time Linux still exists.
- From Linux-side, some common assumptions are not realistic:
  - tasks are *not* completely independent…
  - tasks are *not* (fully) preemptive…
  - operations are *not* atomic…
- From academic-side
  - Linux is a GPOS, full off “hacks”…
  - Hard to learn because of the amount of code…
  - There is no task model for Linux…

But real-time Linux exists!

- It is used in practice and theory:
  - Red Hat Enterprise Linux for Real Time
  - SCHED_DEADLINE
  - Litmus^{RT}
- This is possible because there is a set of operations that provides determinism to Linux.
- But they were never described in a formal way.

Real-time Linux analysis

- Linux developers use tracing features to analyse the system:
  - They see tracing *events* that cause *states* change of the system.
- Discrete Event Systems (DES) methods also use these concepts:
  - *events*, *trace* and *states*…
- DES is can be used in the formalization of system.
- So, why not try to describe Linux using a DES method?

About the paper

- Proposes an automata-based model for describing and verifying the behavior of thread management code in the Linux:
  - Considers only the FULLY_PREEMPTIVE mode (PREEMPT_RT).
  - Single-core.

- Presents the extension of the Linux trace features that enables the trace of the events used in by the automata in a real scenario.

- Presents how the model and tracing features helped catching an inefficiency bug in the scheduler code, leading to a kernel improvement.

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Related work

- Software modeling and verification is an active research area.
- One particularly challenge area is the modeling of OS kernel.
- Some of work use automata, like on “Lazy Abstraction” paper from Henzinger.
- Others with static code analysis for drivers (for Windows and Linux) using SLAM and BLAST.
- “seL4: Formal Verification of an OS Kernel” is an interesting work (in both practical and theoretical case).
  - But it addresses the seL4 case - a very small microkernel OS.
- Linux kernel has lockdep for lockings.
- But none of them address the behavior of real-time tasks for a complex os such as Linux.

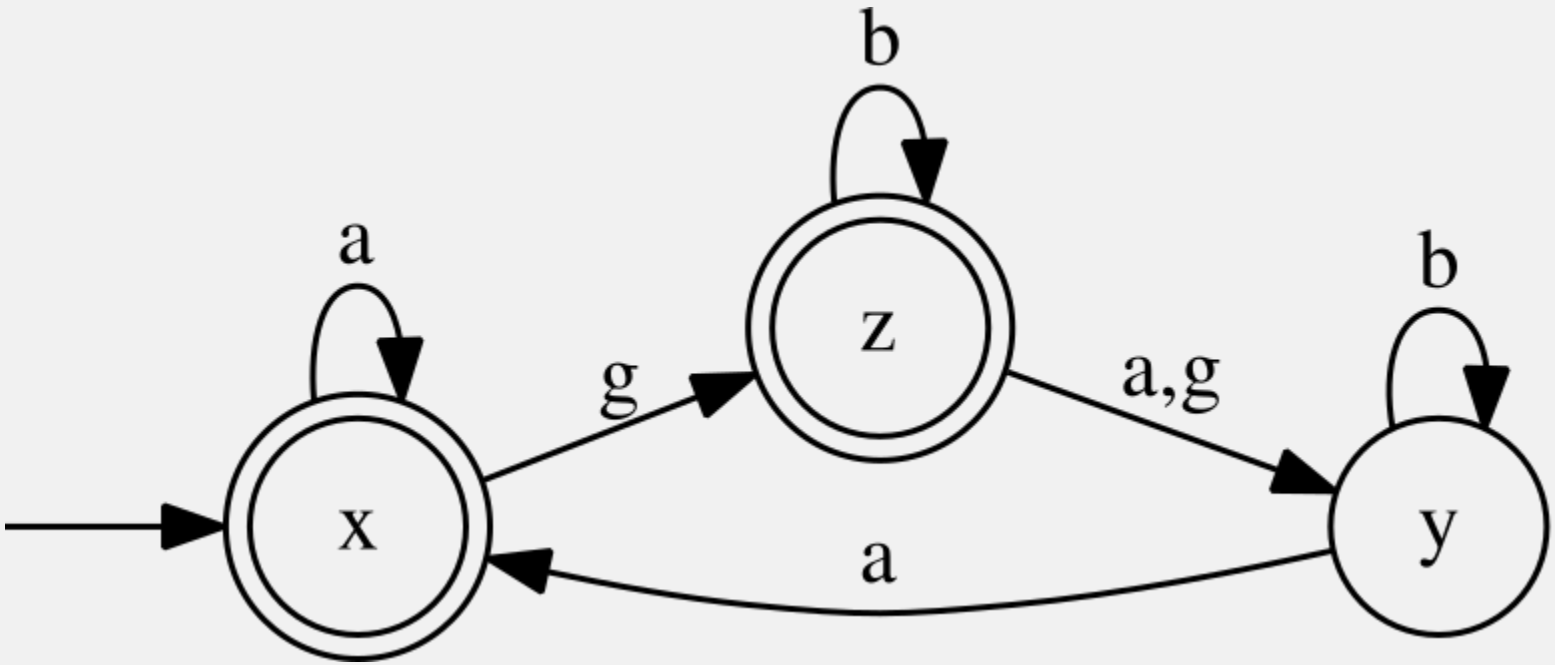
Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Background

- 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 is the 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).

Graphical format



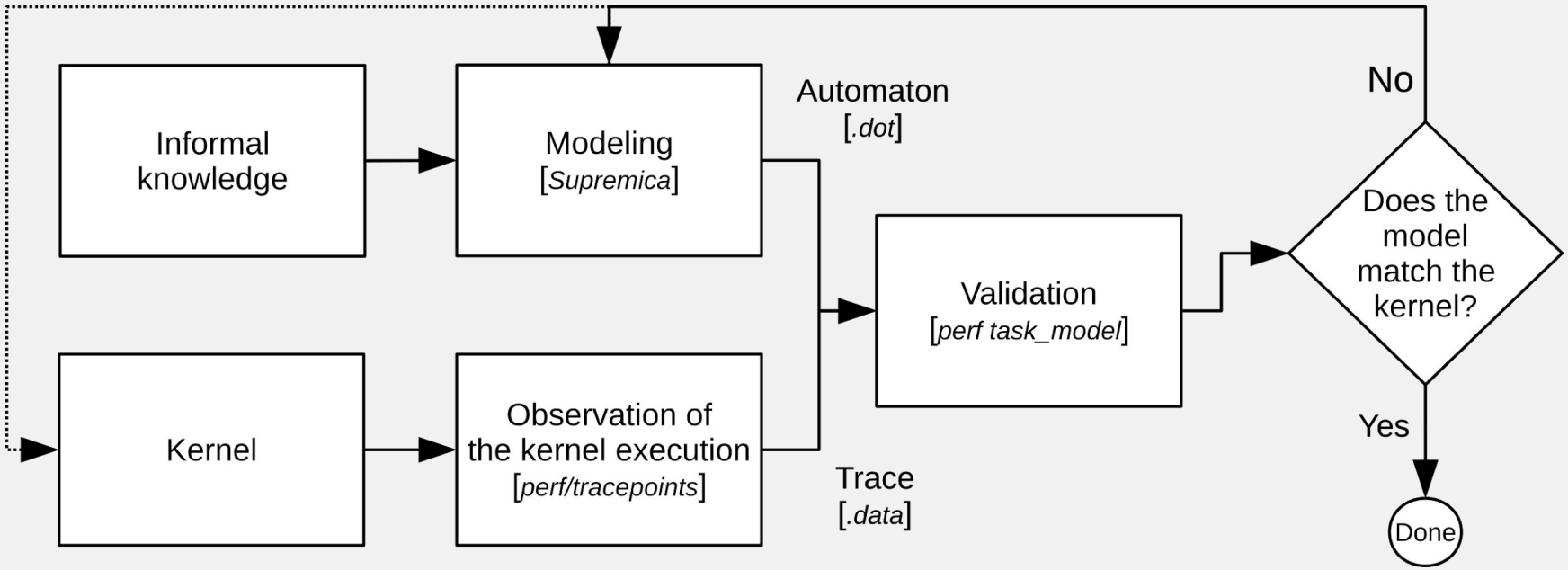
Modeling of complex systems

- Rather than 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 chain of events possible in a controlled system.

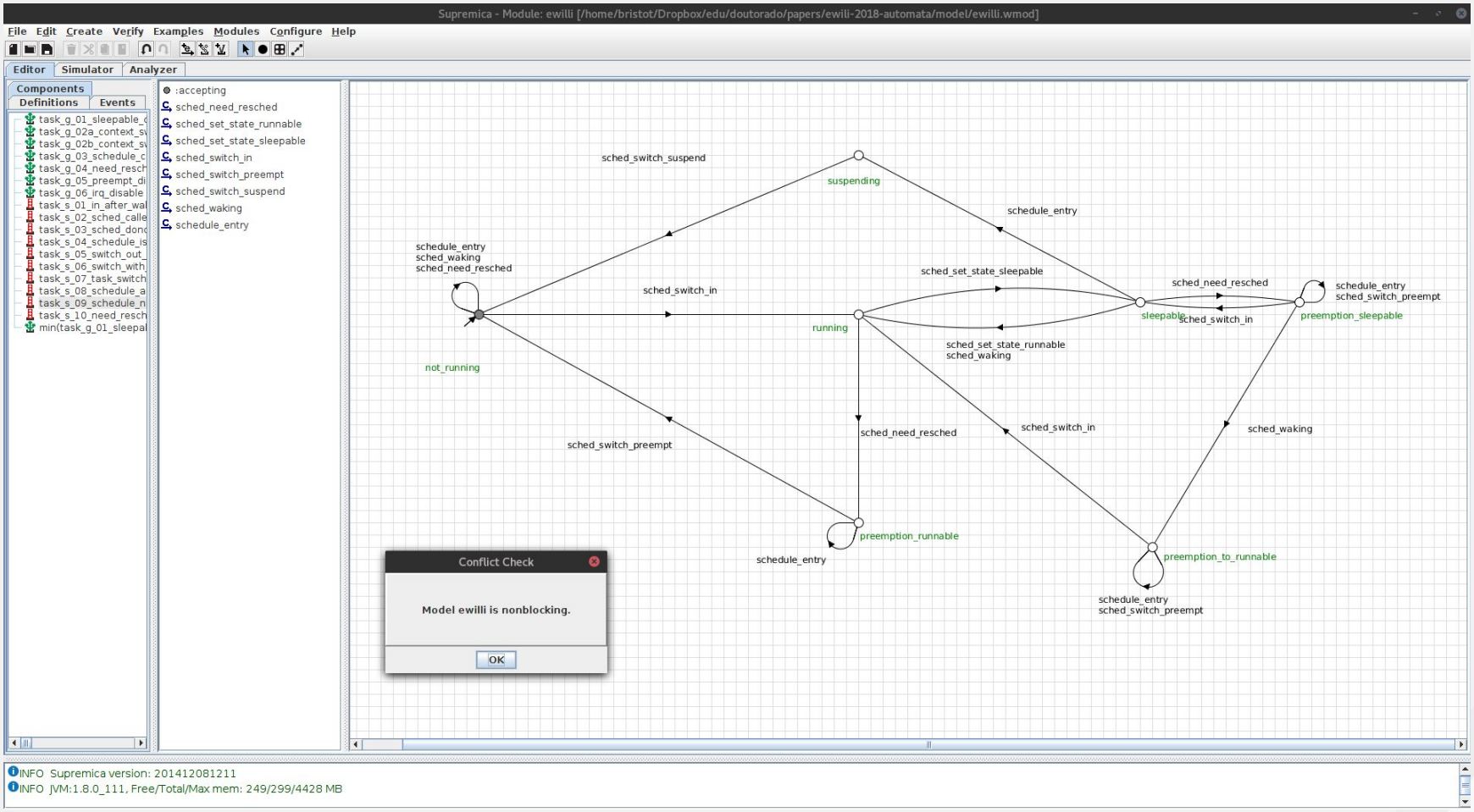
Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Proposed approach



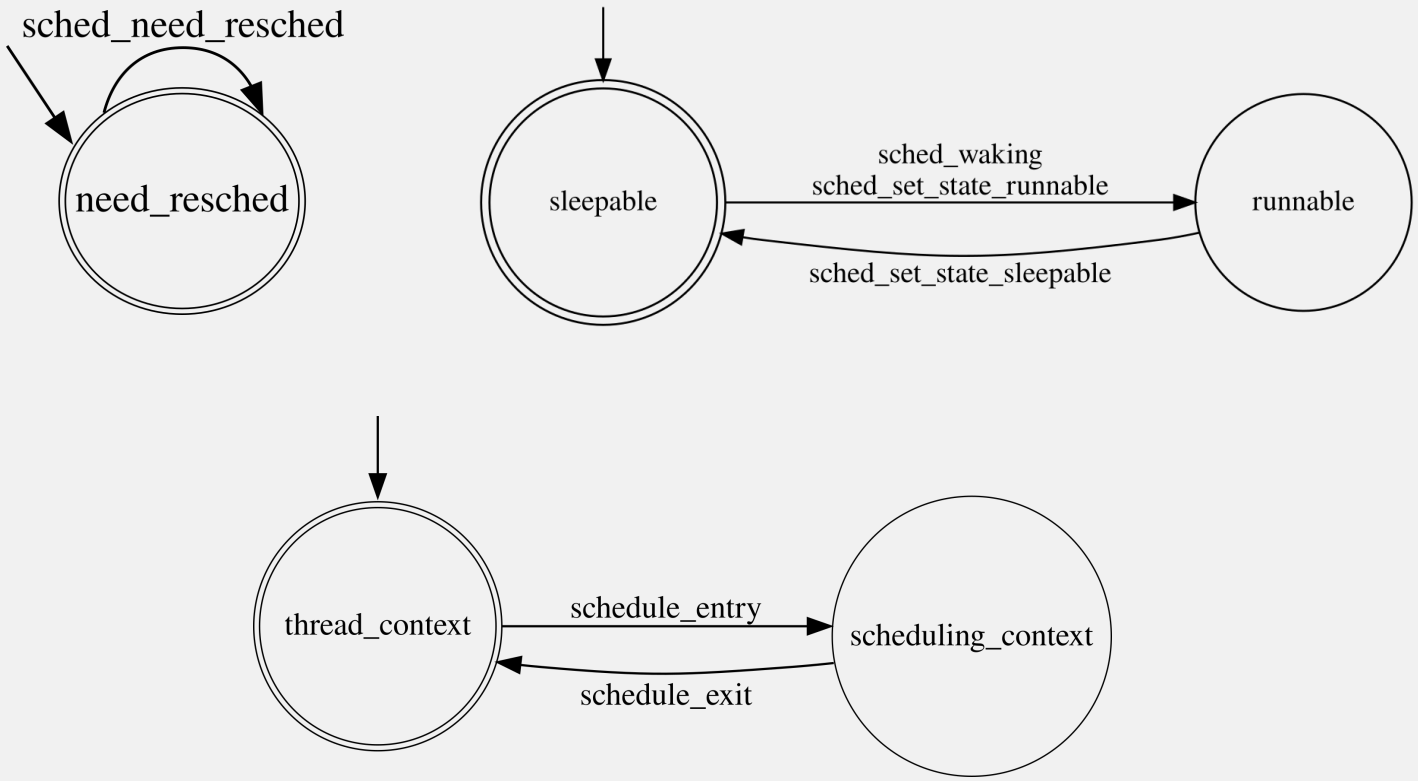
Modeling


Supremica - Module: ewilli [/home/bristot/Dropbox/edu/doutorado/papers/ewili-2018-automata/model/ewilli.wmod]
File Edit Create Verify Examples Modules Configure Help
Editor Simulator Analyzer
Components
Definitions Events
:accepting
sched_need_resched
sched_set_state_runnable
sched_set_state_sleepable
sched_switch_in
sched_switch_preempt
sched_switch_suspend
sched_waking
schedule_entry
sched_switch_suspend
suspending
schedule_entry
schedule_entry
sched_waking
sched_need_resched
sched_switch_in
sched_set_state_sleepable
sched_need_resched
schedule_entry
sched_switch_preempt
sleepable
sched_switch_in
preemption_sleepable
running
sched_set_state_runnable
sched_waking
not_running
sched_switch_preempt
sched_need_resched
sched_switch_in
sched_waking
preemption_runnable
schedule_entry
preemption_to_runnable
schedule_entry
sched_switch_preempt
Conflict Check
Model ewilli is nonblocking.
OK
INFO Supremica version: 201412081211
INFO JVM:1.8.0_111, Free/Total/Max mem: 249/299/4428 MB

Modeling details

- The final model has:
  - 149 states;
  - 327 transitions;
  - It would be barely impossible to model it directly.
- Using the modular approach, the final model is composed of:
  - 15 events;
  - 7 generators;
  - 10 specifications.
    - The most complex module (a specification) has 7 states and 29 transitions.

Generator model examples



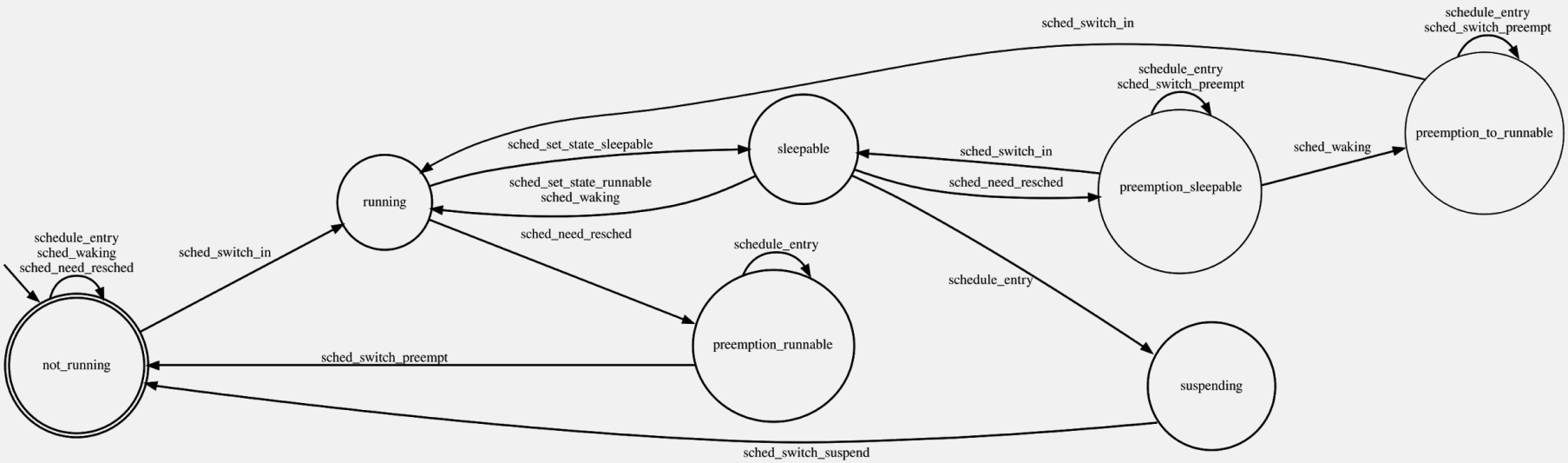
Tracing and events

Kernel event	Automaton event	Status
irq:local_irq_disable	**local_irq_disable**	**new**
irq:local_irq_enable	local_irq_enable	new
sched:sched_preempt_disable	**preempt_disable**	**new**
sched:sched_preempt_enable	preempt_enable	new
sched:sched_need_resched	**sched_need_resched**	**new**
sched:sched_set_state	sched_set_state_runnable	new
sched:sched_set_state	**sched_set_state_sleepable**	**new**
sched:sched_entry	schedule_entry	new
sched:sched_exit	**schedule_exit**	**new**
sched:sched_switch	sched_switch_in	exist
sched:sched_switch	**sched_switch_in_o**	**exist**
sched:sched_switch	sched_switch_out_o	exist
sched:sched_switch	**sched_switch_preempt**	**exist**
sched:sched_switch	sched_switch_suspend	exist
sched:sched_waking	**sched_waking**	**exist**

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Scheduling call specification

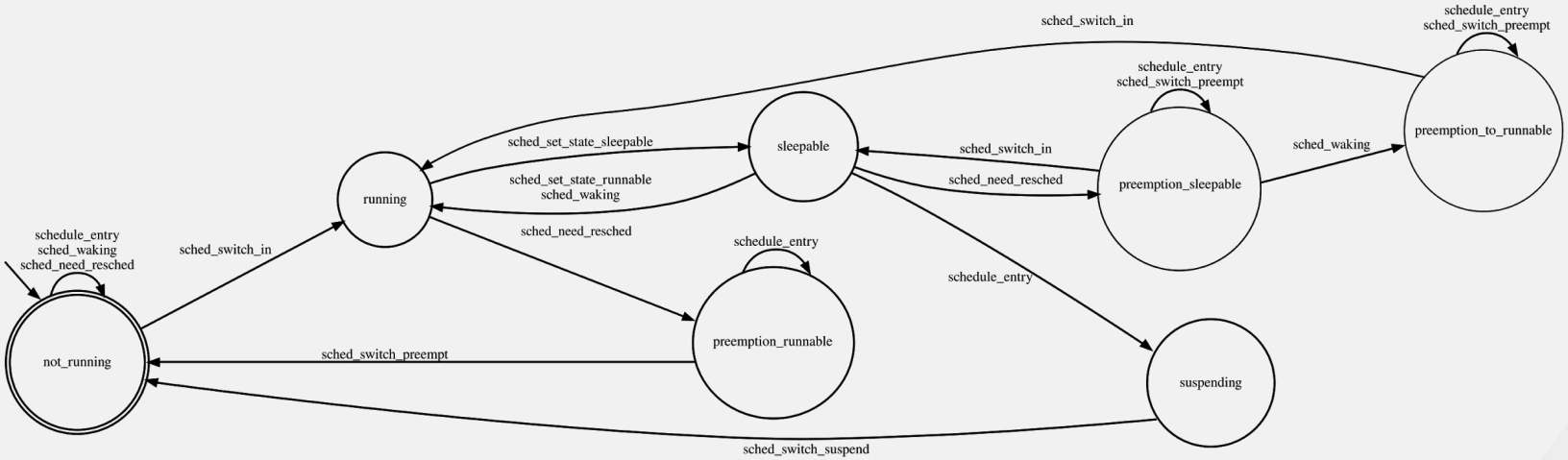


Tracing example

```
1:   ktimersoftd/0   8 [000]   784.425631:  sched:sched_switch: ktimersoftd/0:8 [120] R ==> kworker/0:2:728 [120]
2:     kworker/0:2 728 [000]   784.425926:  sched:sched_set_state: sleepable
3:     kworker/0:2 728 [000]   784.425936:  sched:set_need_resched: comm=kworker/0:2 pid=728
4:     kworker/0:2 728 [000]   784.425941:  sched:sched_entry: at preempt_schedule_common
5:     kworker/0:2 728 [000]   784.425945:  sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:1:724 [120]
6:  irq/14-ata_piix 86 [000]   784.426515:  sched:sched_waking: comm=kworker/0:2 pid=728 prio=120 target_cpu=000
7:     kworker/0:1 724 [000]   784.426610:  sched:sched_switch: kworker/0:1:724 [120] t ==> kworker/0:2:728 [120]
8:     kworker/0:2 728 [000]   784.426616:  sched:sched_entry: at schedule
9:     kworker/0:2 728 [000]   784.426619:  sched:sched_switch: kworker/0:2:728 [120] R ==> kworker/0:2:728 [120]
```

Tracing and events

Line	Event	State
1	**sched_switch_in**	**running**
2	sched_set_state_sleepable	sleepable
3	**sched_need_resched**	**preemption_sleepable**
4	sched_entry	preemption_sleepable
5	**sched_switch_preempt**	**preemption_sleepable**
6	sched_waking	preemption_to_runnable
7	**sched_switch_in**	**running**
8	sched_entry	**not recognized!**



Code analysis

```
void schedule(void)
{
  sched_submit_work(){
        wq_worker_sleeping() {
              preempt_disable();
              wakeup(kworker_2);
              preempt_enable() {
                    should_resched() {
                          __schedule() {
                                context_switch
                          }
                    }
              }
        }
  }
  do {
        preempt_disable();
        __schedule();
        sched_preempt_enable_no_resched();
  } while (need_resched());
  sched_update_worker();
}
```

Coding optimization

```
void schedule(void)
{
  sched_submit_work(){
        wq_worker_sleeping() {
              preempt_disable();
              wakeup(kworker_2);
              preempt_enable() {
                    should_resched() {
                          __schedule() {
                                context_switch
                          }
                    }
              }
        }
  }
  do {
        preempt_disable();
        __schedule();
        sched_preempt_enable_no_resched();
  } while (need_resched());
  sched_update_worker();
}
```

Suggested patch:

```
--- a/kernel/sched/core.c
+++ b/kernel/sched/core.c
@@ -3545,9 +3545,16 @@ static inline void sched_submit_work(struct task_struct *tsk)
      /*
       * If a worker went to sleep, notify and ask workqueue whether
       * it wants to wake up a task to maintain concurrency.
+      *
+      * As this function is called inside the schedule() context,
+      * we disable preemption to avoid it calling schedule() again
+      * in the possible wakeup of a kworker.
       */
-     if (tsk->flags & PF_WQ_WORKER)
+     if (tsk->flags & PF_WQ_WORKER) {
+          preempt_disable();
           wq_worker_sleeping(tsk);
+          preempt_enable_no_resched();    /* Avoids calling the scheduler */
+     }
```

Agenda

- Introduction
- Related Work
- Background on automata theory
- Proposed approach
- Early results
- Conclusions and future work

Conclusions

- The understanding of Linux's task behavior is fundamental for the development of new real-time algorithms.
- It was possible to model the behavior of Linux's tasks using automata
  - The modular approach turns the modeling more intuitive.
  - It was possible to define logical statements from the automata.
    - E.g, sufficient/necessary conditions for scheduling calls.
- The automata was easily integrated with the tracing techniques used by developers.
- The integrated analysis of a trace and automata helped in the improvement of Linux.

Future work

- The modeling is an on-going work.
- Next steps:
  - Modeling IRQ/NMIs integrated with tasks.
  - Modeling locks.
  - Modeling multi-core.
- Create an in-kernel preempt model verifier in the same way that **lockdep** does for locks.
- Among other things.

Thanks!