# Runtime verification of real-time applications using trace data and model requirements

Progress Report Meeting
May 13, 2015

Raphaël BEAMONTE  Michel DAGENAIS

Distributed Open Reliable Systems Analysis Lab
Computer and Software Engineering Department
École Polytechnique de Montréal

# Introduction

- Low-overhead tracing is available
- But trace analysis requires users to have kernel knowledge
- So what about automating the analysis ?
  - CAE suggested to verify applications' execution using specifications
  - Ericsson is working towards programming at model level
  - **Why couldn't we do both?**
- $\Rightarrow$ model-based constraints

# Table of Contents


1. Introduction
2. Model-based constraints approach
3. Case studies
4. Conclusion

# Model representation

## What do we need?

- A way to follow the workflow of the application
  - $\Rightarrow$ tracepoints (or more precisely the events generated)
- A way to define the “zones” to check
- A way to define constraints

# Model representation

## What would our models look like?

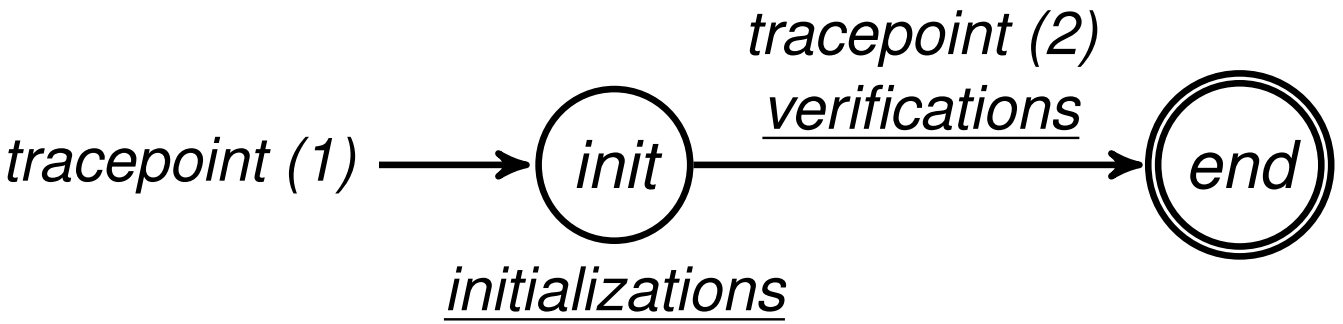



Figure: State machine representation that can be used to check metrics using traces

# Model representation

## What would our models look like?

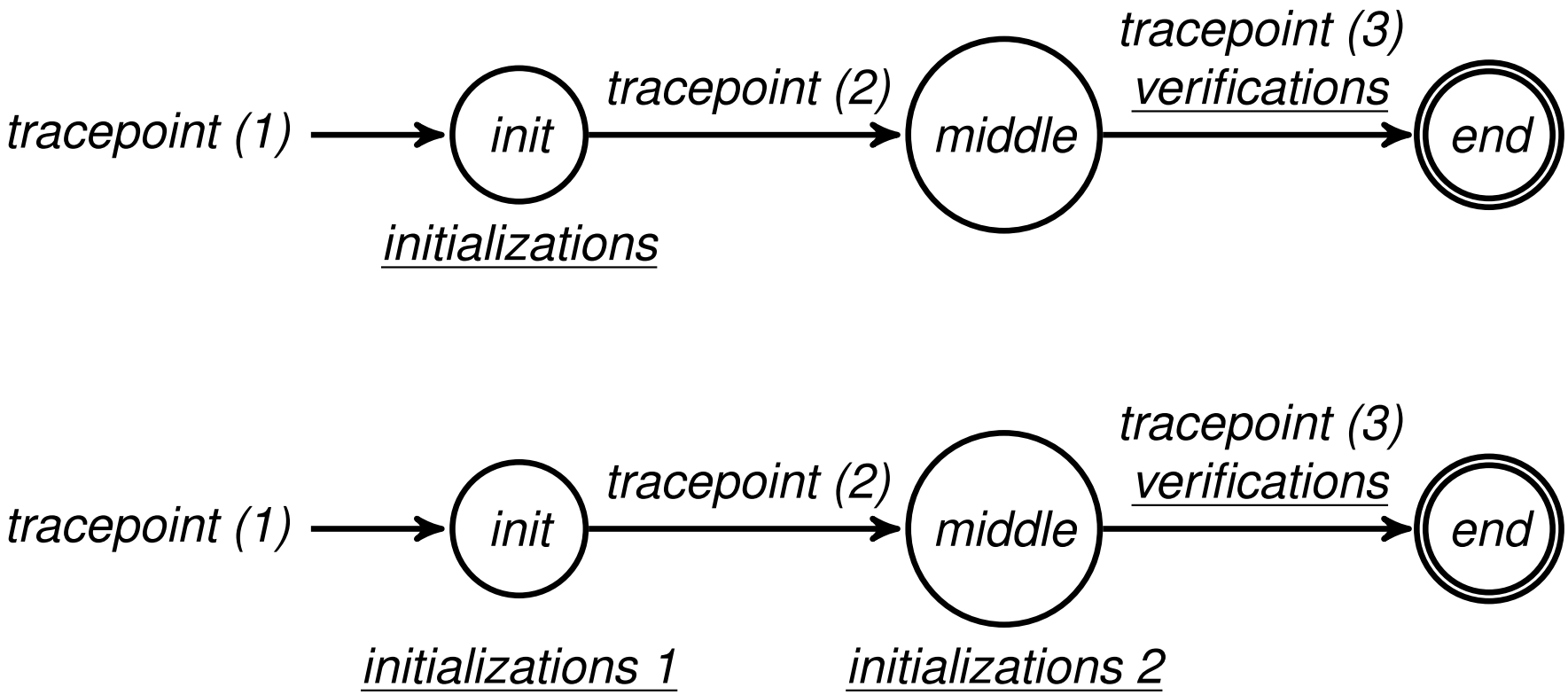



Figure: State machine representations with late verification of constraints and a transitional state

# Model representation

## What would our models look like?

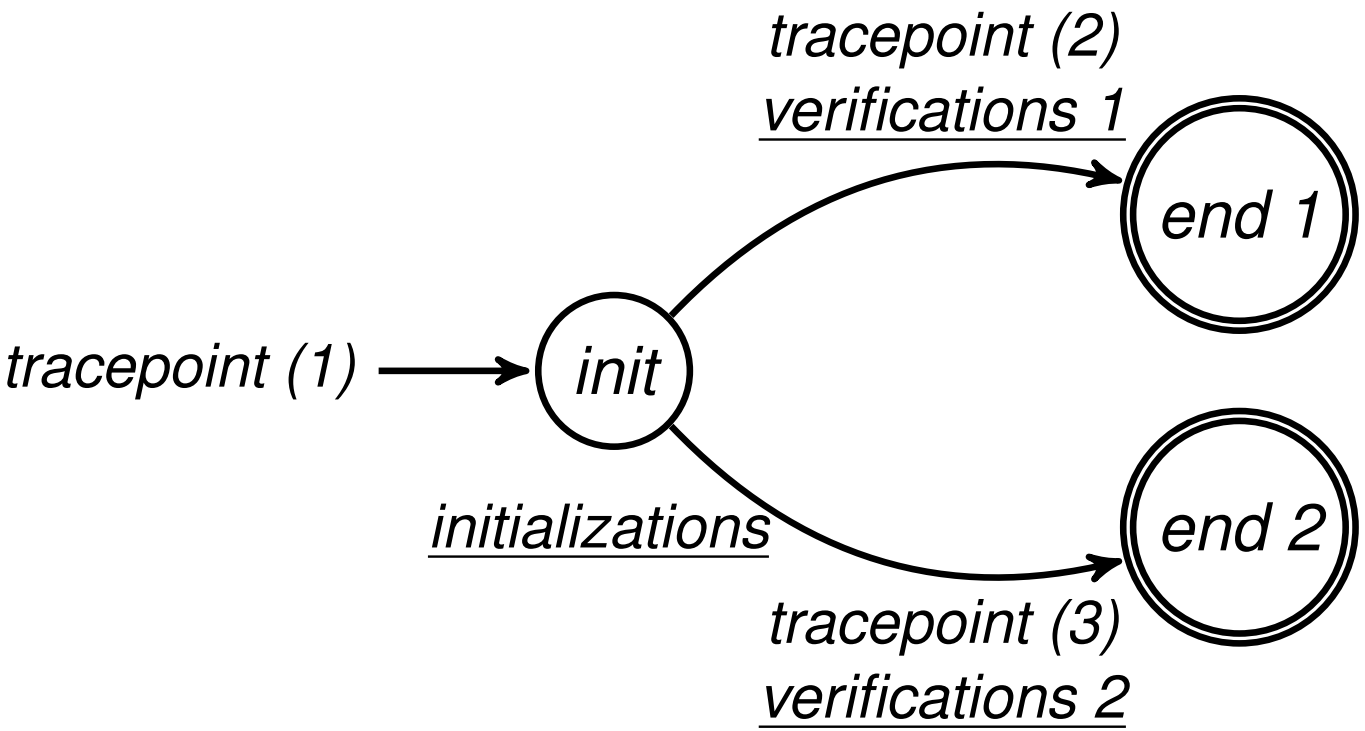



Figure: State machine representation with multiple next states for state “init”

# Model representation

## What would our models look like?

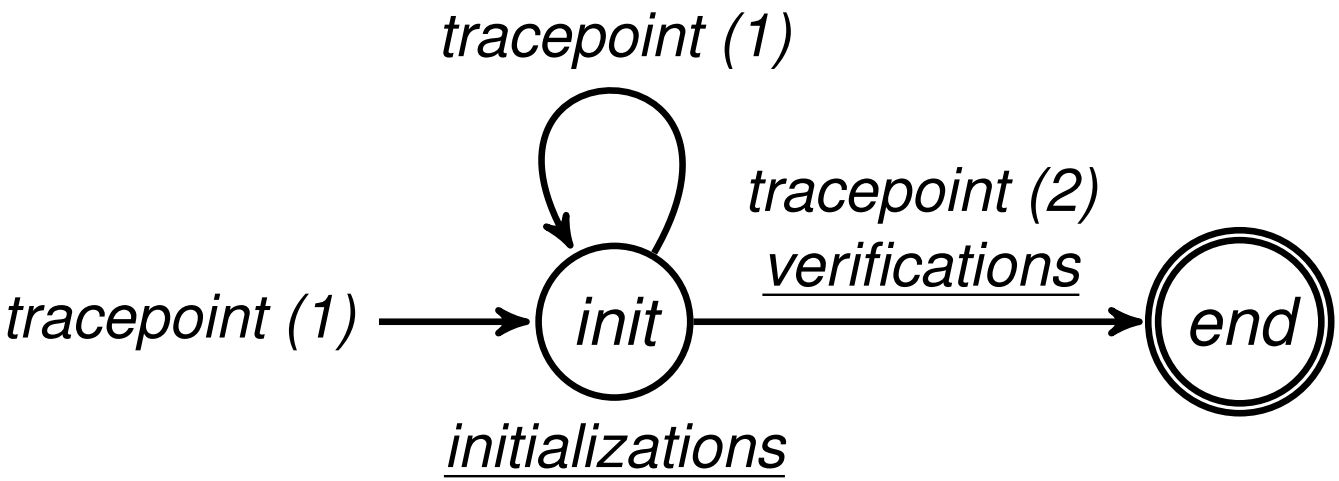
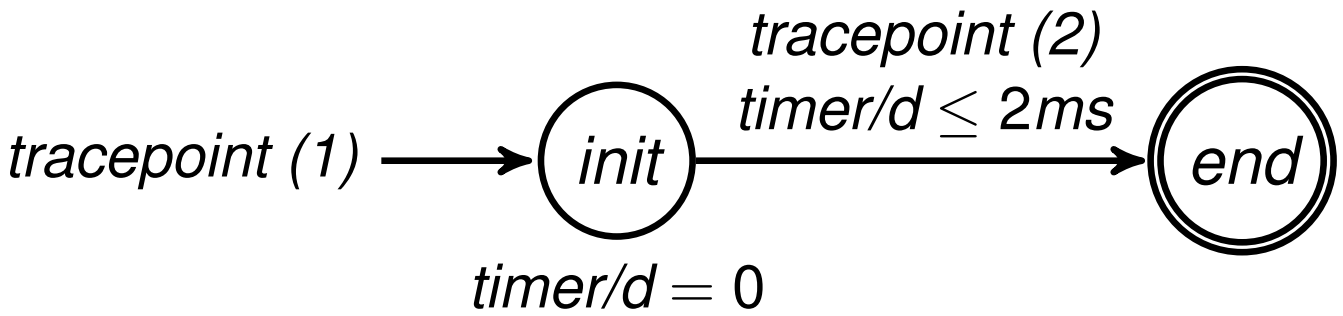



Figure: State machine representation using a loop to go over the initializations when reading an event generated by tracepoint (1)

# Sample constraints

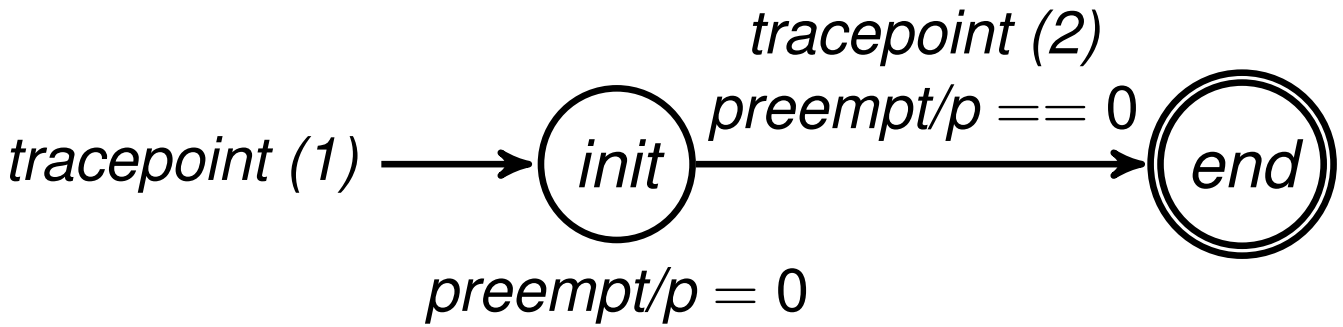
## Deadline constraint




- tracepoints *(1)* and *(2)* generate timestamped events
- timer/d $= 0$ saves the timestamp of the event from *(1)*
- timer/d $\leq 2ms$ verifies that the timestamp of the event from *(2)* is at most 2 ms after the one from *(1)*
- Needs only userspace traces

# Sample constraints

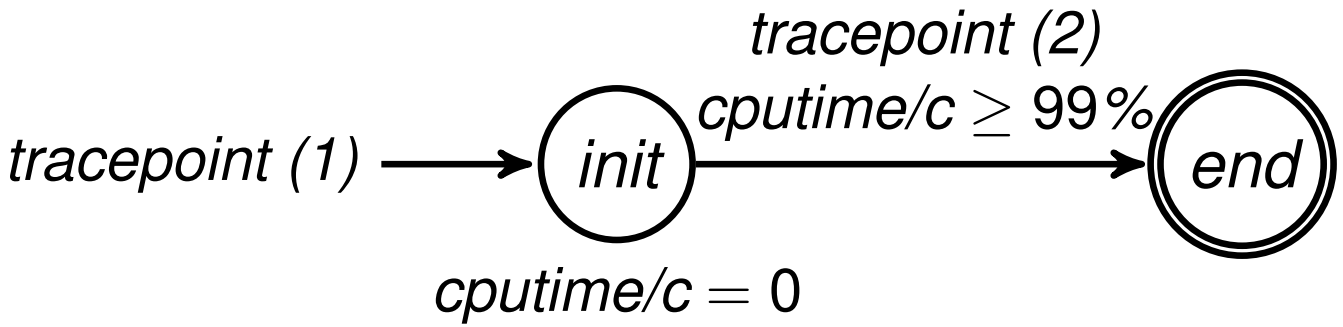
## Preemption constraint




- preempt/p = 0 initializes the counter to 0 at *(1)*
- preempt/p == 0 verifies that the counter is still 0 at *(2)* (i.e. counts the number of `sched_switch` during the period)
- Needs kernel and userspace traces
- Same idea: system calls constraint

# Sample constraints

## CPU usage constraint




- cputime/c $= 0$ sets the period start at *(1)*
- cputime/c $\geq$ 99% verifies that between *(1)* and *(2)* our process used at least 99 % of the CPU time
- Needs kernel and userspace traces
- Same idea: status constraint (wait-for-cpu, wait-blocked)

# Case studies

**. . . or how our approach could be used to detect some common problems**

1. Occasional missing of deadlines
2. Priority inversion
3. Unefficient synchronization method
4. Wait-blocked processes on multiprocessor activity
5. Wait-blocked processes while using external resources

# Case studies

## (1) Occasional missing of deadlines

### Problem

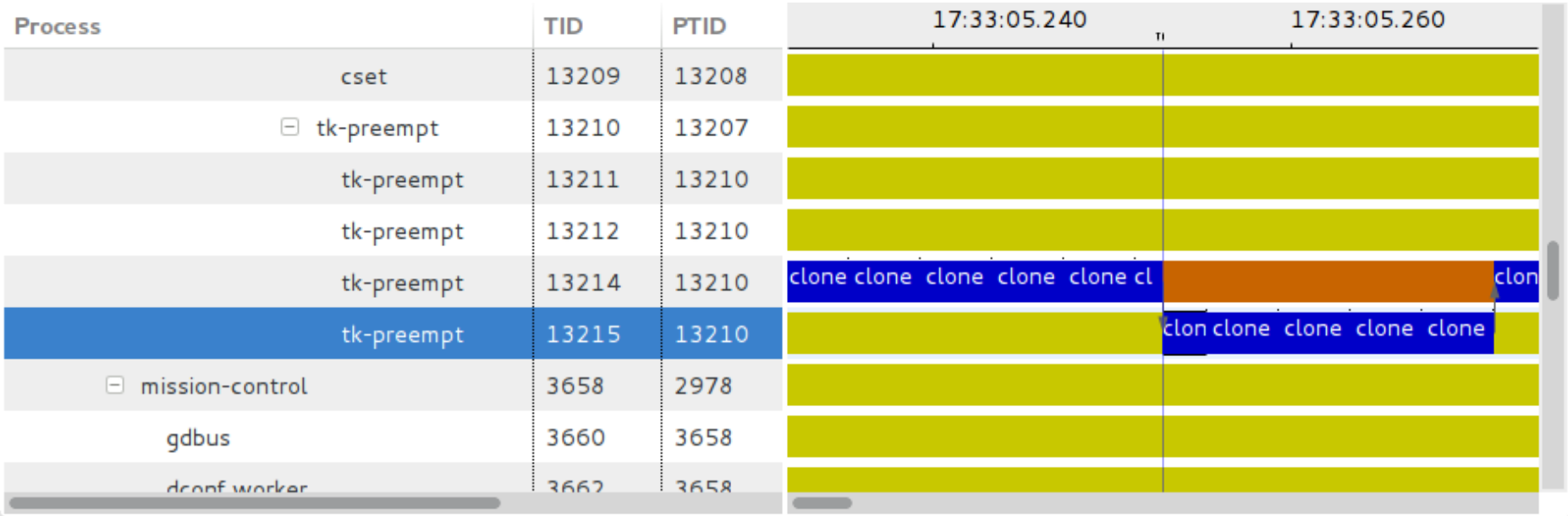
In a task that appears a lot of times, some deadlines are missed occasionnaly.

### Analysis

What happened on the kernel side when the deadlines were missed ?

# Case studies

## (1) Occasional missing of deadlines




ust/uid/0/64-bit | kernel

```
[17:33:05.252828753] (+0.000000748) computer
    sched_switch: { cpu_id = 2 }, { vtid = 13214, vpid =
    13210 }, { prev_comm = "tk-preempt", prev_tid =
    13214, prev_prio = -2, prev_state = 0, next_comm = "
    tk-preempt", next_tid = 13215, next_prio = -21 }
```

hrtimer=18446612150016859360, now=8091604000280, function=18446744071579722464, context._vtid=13215, contex

# Case studies

## (1) Occasional missing of deadlines

### Problem

In a task that appears a lot of times, some deadlines are missed occasionnaly.

### Analysis

What happened on the kernel side when the deadlines were missed ?

### Constraints that would have helped in our example

Deadline ; Preemption ; CPU usage

# Case studies

## (2) Priority inversion

### Problem

A high priority process still ends up being preempted by a lower priority process.

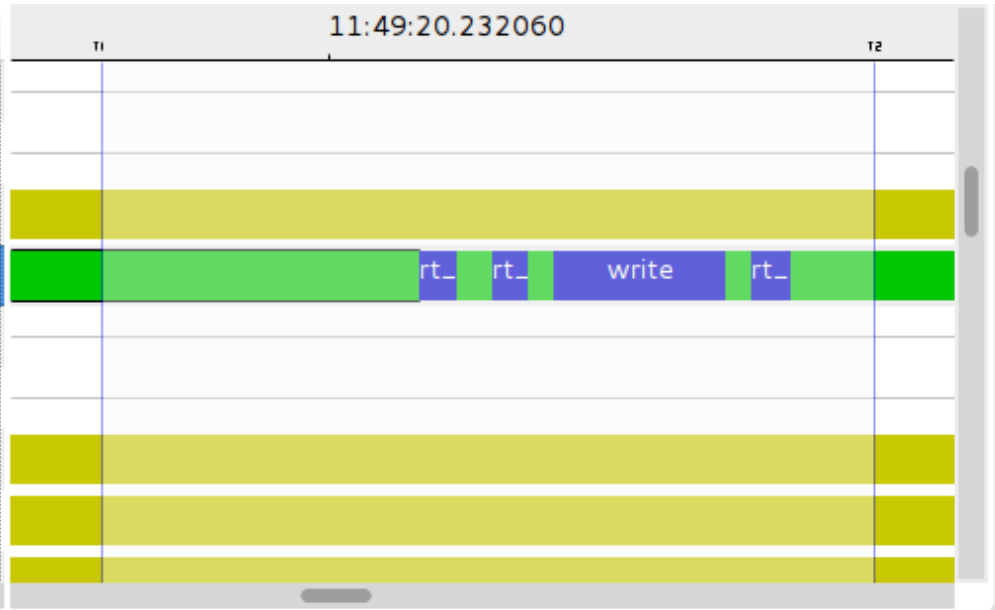
### Analysis

What happened that lead to that preemption?

# Case studies

## (2) Priority inversion

| | TID | PTID | Birth time |
|---|---|---|---|
| sudo | 6785 | 6736 | 11:49:04.711563346 |
| sh | 6786 | 6785 | 11:49:04.720466460 |
| sudo | 6787 | 6736 | 11:49:04.755070270 |
| npt | 6788 | 6787 | 11:49:04.763480843 |
| sudo | 6801 | 6736 | 11:49:51.725587947 |
| lttng | 6802 | 6801 | 11:49:51.857443514 |
| | 2857 | 1 | 11:49:04.688760858 |
| | 2874 | 1 | 11:49:04.688761987 |




kernel LTTng | kernel CAE | LTTng

```
hrtimer_cancel, hrtimer_expire_entry, softirq_raise,
rcu_utilization, softirq_raise, rcu_utilization,
hrtimer_expire_exit, hrtimer_start, sys_rt_sigpending,
exit_syscall, sys_rt_sigprocmask, exit_syscall,
sys_write, sched_wakeup, exit_syscall,
sys_rt_sigprocmask, exit_syscall
```

# Case studies

## (2) Priority inversion

### Problem

A high priority process still ends up being preempted by a lower priority process.

### Analysis

What happened that lead to that preemption?

### Constraints that would have helped in our example

CPU usage ; Preemption ; System calls

# Case studies

## (3) Unefficient synchronization method

### Problem

Some programs use `sleep` as a synchronization method.

### Analysis

- How much time a process has spent as wait-blocked? What was he waiting for?
- Analyze of `apt` using the critical path
- Analyze of `MongoDB` using the call stack and critical path

# Case studies

## (3) Unefficient synchronization method

### Problem

Some programs use `sleep` as a synchronization method.

### Analysis

- How much time a process has spent as wait-blocked? What was he waiting for?
- Analyze of `apt` using the critical path
- Analyze of `MongoDB` using the call stack and critical path

### Constraints that would have helped in our example

Deadline (using mean delay of task, for `MongoDB` it could be around 1 s for instance)

# Case studies

## (4) Wait-blocked processes on multiprocessor activity

### Problem

Scalability inefficiency: more processors $\Rightarrow$ less performance

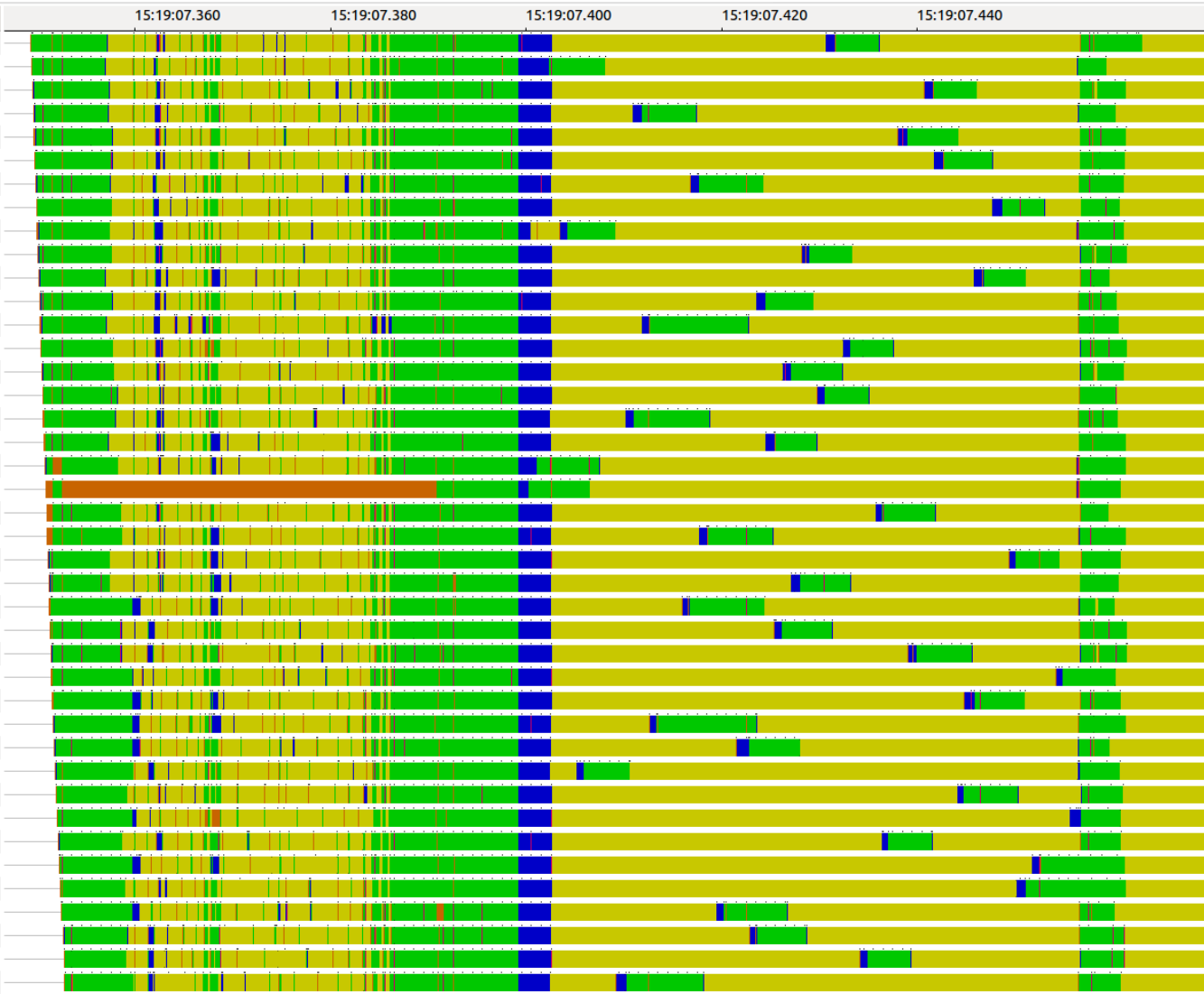
### Analysis

What happened between the last working scaling step and the first performance regression?

# Case studies

## (4) Wait-blocked processes on multiprocessor activity

| Process | TID | PTID | Birth time | Trace |
|---|---|---|---|---|
| io-test | 7659 | 7653 | 15:19:07.349273524 | io-test-large-chunk |
| io-test | 7660 | 7653 | 15:19:07.349362116 | io-test-large-chunk |
| io-test | 7661 | 7653 | 15:19:07.349466784 | io-test-large-chunk |
| io-test | 7662 | 7653 | 15:19:07.349567738 | io-test-large-chunk |
| io-test | 7663 | 7653 | 15:19:07.349653359 | io-test-large-chunk |
| io-test | 7664 | 7653 | 15:19:07.349730222 | io-test-large-chunk |
| io-test | 7665 | 7653 | 15:19:07.349803949 | io-test-large-chunk |
| io-test | 7666 | 7653 | 15:19:07.349883533 | io-test-large-chunk |
| io-test | 7667 | 7653 | 15:19:07.349955517 | io-test-large-chunk |
| io-test | 7668 | 7653 | 15:19:07.350034725 | io-test-large-chunk |
| io-test | 7669 | 7653 | 15:19:07.350114954 | io-test-large-chunk |
| io-test | 7670 | 7653 | 15:19:07.350198220 | io-test-large-chunk |
| io-test | 7671 | 7653 | 15:19:07.350270267 | io-test-large-chunk |
| io-test | 7672 | 7653 | 15:19:07.350343310 | io-test-large-chunk |
| io-test | 7673 | 7653 | 15:19:07.350416502 | io-test-large-chunk |
| io-test | 7674 | 7653 | 15:19:07.350511038 | io-test-large-chunk |
| io-test | 7675 | 7653 | 15:19:07.350596689 | io-test-large-chunk |
| io-test | 7676 | 7653 | 15:19:07.350672078 | io-test-large-chunk |
| io-test | 7677 | 7653 | 15:19:07.350753453 | io-test-large-chunk |
| io-test | 7678 | 7653 | 15:19:07.350837615 | io-test-large-chunk |
| io-test | 7679 | 7653 | 15:19:07.350911700 | io-test-large-chunk |
| io-test | 7680 | 7653 | 15:19:07.350986857 | io-test-large-chunk |
| io-test | 7681 | 7653 | 15:19:07.351058482 | io-test-large-chunk |
| io-test | 7682 | 7653 | 15:19:07.351130601 | io-test-large-chunk |
| io-test | 7683 | 7653 | 15:19:07.351204445 | io-test-large-chunk |
| io-test | 7684 | 7653 | 15:19:07.351278853 | io-test-large-chunk |
| io-test | 7685 | 7653 | 15:19:07.351348336 | io-test-large-chunk |
| io-test | 7686 | 7653 | 15:19:07.351418551 | io-test-large-chunk |
| io-test | 7687 | 7653 | 15:19:07.351527255 | io-test-large-chunk |
| io-test | 7688 | 7653 | 15:19:07.351617418 | io-test-large-chunk |
| io-test | 7689 | 7653 | 15:19:07.351720711 | io-test-large-chunk |
| io-test | 7690 | 7653 | 15:19:07.351805365 | io-test-large-chunk |
| io-test | 7691 | 7653 | 15:19:07.351882896 | io-test-large-chunk |
| io-test | 7692 | 7653 | 15:19:07.352032911 | io-test-large-chunk |
| io-test | 7693 | 7653 | 15:19:07.352123417 | io-test-large-chunk |
| io-test | 7694 | 7653 | 15:19:07.352213516 | io-test-large-chunk |
| io-test | 7695 | 7653 | 15:19:07.352303961 | io-test-large-chunk |
| io-test | 7696 | 7653 | 15:19:07.352391671 | io-test-large-chunk |
| io-test | 7697 | 7653 | 15:19:07.352637963 | io-test-large-chunk |
| io-test | 7698 | 7653 | 15:19:07.352722317 | io-test-large-chunk |
| io-test | 7699 | 7653 | 15:19:07.352781222 | io-test-large-chunk |



# Case studies

## (4) Wait-blocked processes on multiprocessor activity

### Problem

Scalability inefficiency: more processors $\Rightarrow$ less performance

### Analysis

What happened between the last working scaling step and the first performance regression?

### Constraints that would have helped in our example

CPU usage ; Process status (wait-blocked)

# Case studies

## (5) Wait-blocked processes while using external resources

### Problem

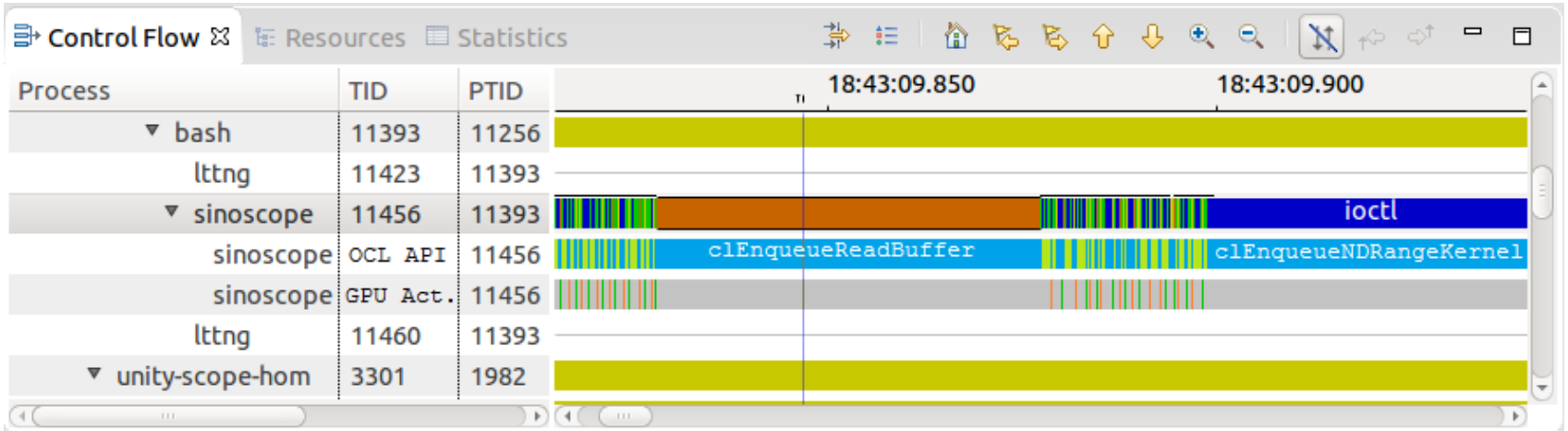
When using external resources such as GPU, bottlenecks can appear if the CPU-GPU work is not optimized

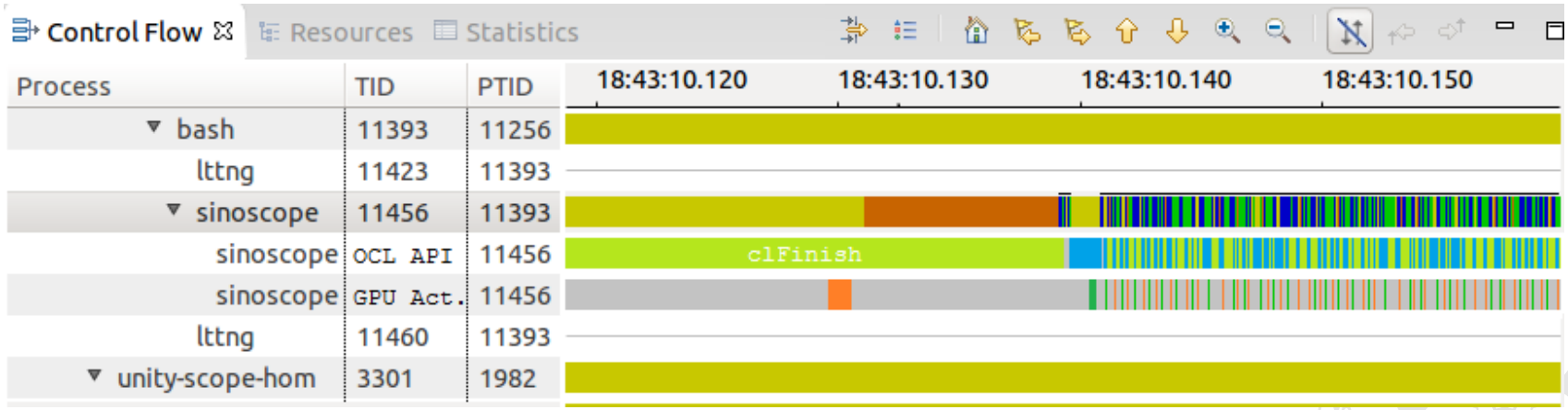
### Analysis

Was it caused by the CPU? By the GPU? What was the CPU process waiting for?

# Case studies

## (5) Wait-blocked processes while using external resources



# Case studies

## (5) Wait-blocked processes while using external resources

### Problem

When using external resources such as GPU, bottlenecks can appear if the CPU-GPU work is not optimized

### Analysis

Was it caused by the CPU? By the GPU? What was the CPU process waiting for?

### Constraints that would have helped in our example

CPU: preemption or Status (wait-blocked) $\Rightarrow$ always scheduled
GPU: Status (wait-blocked) if we know the GPU task duration

# Conclusion

- New approach using constraints to automatically detect problems using traces
- Overview of high performance cases where tracing was useful to identify an unexpected behavior
- Automatic identification of those behaviors using our model approach
- Range of problems we can detect is larger than what the literature provides thanks to kernel tracing
- Future work:
  - Identify the origins of the detected problems
  - Propose simple solutions for those problems (e.g. higher priority for a preempted process)

# Thank you.
# Any question?

raphael.beamonte@polymtl.ca

Slides:
www.dorsal.polymtl.ca/~rbeamonte/dorsal-pm-may2015.pdf