Real-Time Linux Response Time Measurement

Julien Desfossez
Michel Dagenais

May 2016
École Polytechnique de Montreal

Latency-tracker

- Kernel module to track down latency problems at run-time
- Simple API that can be called from anywhere in the kernel (tracepoints, kprobes, netfilter hooks, hardcoded in other module or the kernel tree source code)
- Keep track of entry/exit events and calls a callback if the delay between the two events is higher than a threshold

Usage

```
tracker = latency_tracker_create(threshold,
timeout, callback);

latency_tracker_event_in(tracker, key);
....
latency_tracker_event_out(tracker, key);
```

If the delay between the event_in and event_out for the same key is higher than "threshold", the callback function is called.

The timeout parameter allows to launch the callback if the event_out takes too long to arrive (off-CPU profiling).

Implemented Use-Cases

- Block layer latency
  - Delay between block request issue and complete
- Wake-up latency
  - Delay between sched_wakeup and sched_switch
- Network latency
- IRQ handler latency
- System call latency
  - Delay between the entry and exit of a system call
- Offcpu latency
  - How long a process has been scheduled out

Performance Optimizations

- Controlled memory allocation
- Lock-less per-cpu RCU free-list
- Out-of-context reallocation of memory if needed/enabled
- Kernel-ported lock-less userspace-rcu hashtable
- Custom call_rcu thread to avoid the variable side-effects of the built-in one
- **Numa-aware memory allocator**

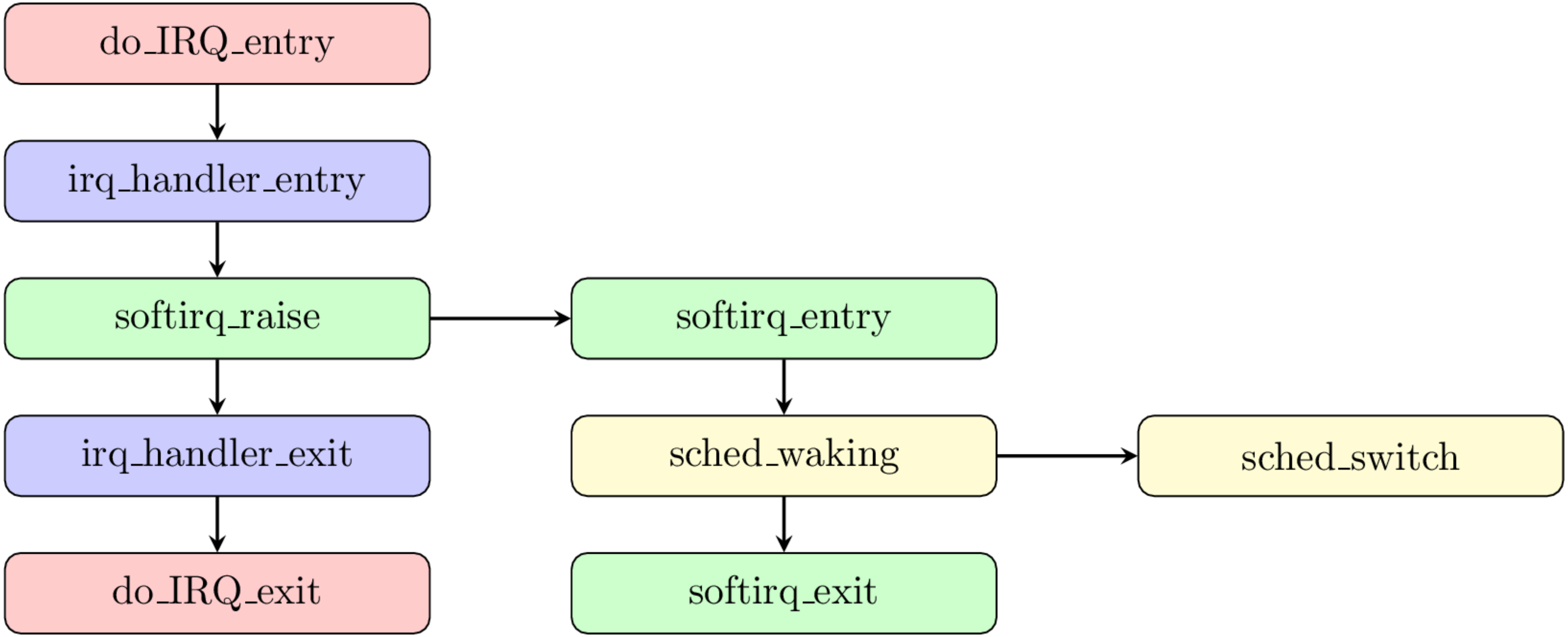
Measuring Response Time Latency

- Start tracking when the kernel receives the interrupt
- Compute the delay up to the moment when:
  - The target task gets scheduled in
  - The target task informs the kernel it finished its work
  - The target task goes back to waiting for the next interrupt
- Launch a user-defined action on high latency

Measuring Response Time Latency

- Work with the two main workloads:
  - periodic (timers)
  - aperiodic (hardware interrupts)

Interrupts Critical Path



Online Critical Tree

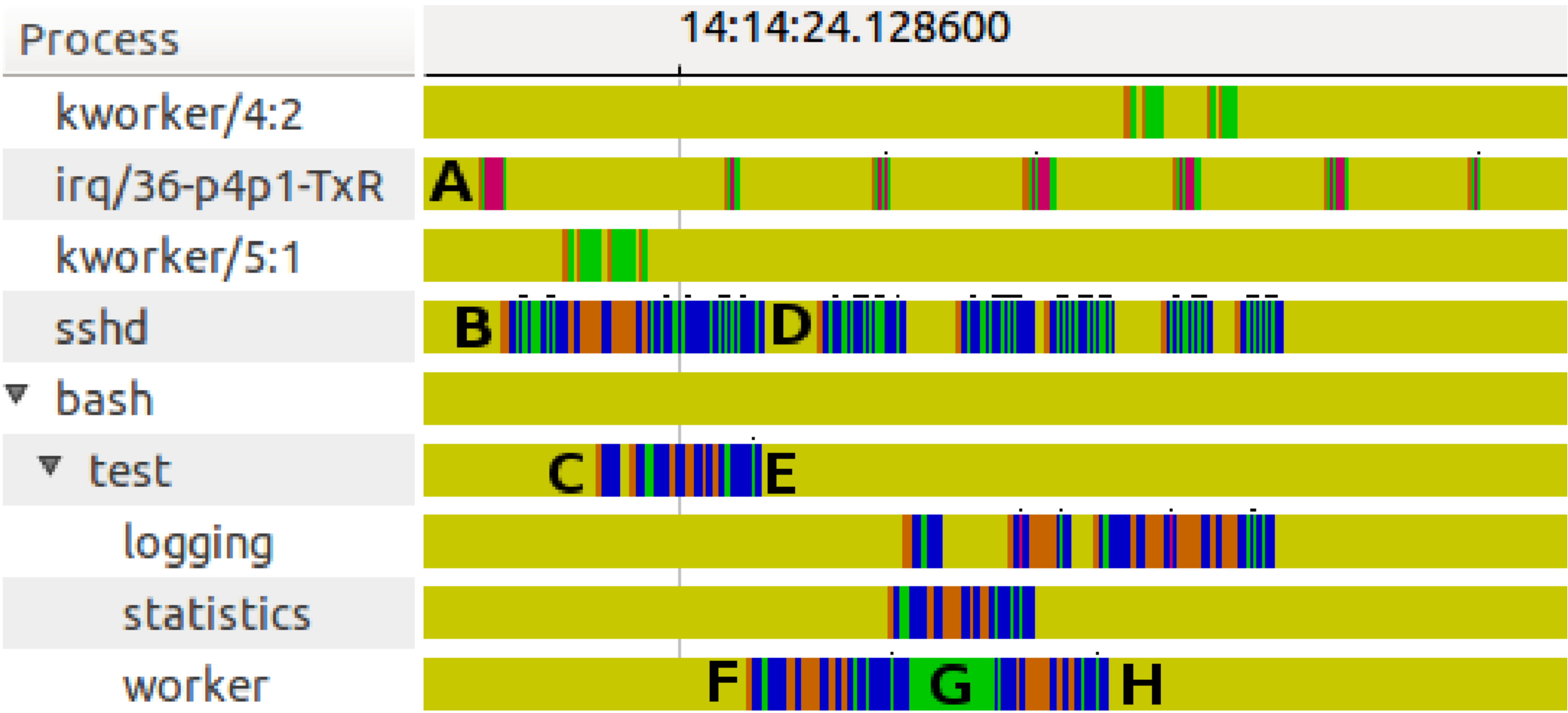
- Tracking an interrupt up to the point where a user-space task starts to run is usually a chain (no branches)
- But if we track an interrupt until the target task completes its work, there can be a lot of branches
- Each call to sched_waking or softirq_raise creates a new branch in the chain

Online Critical Tree

- We stop the tracking when one chain matches all the criteria
- We only know which one at the end
- So we need to track everything and cleanup as soon as possible to limit the overhead

Tracking in User-space

- Do not stop tracking when the target task is scheduled in or scheduled out
- More complex workloads:
  - Asynchronous
  - Active polling
  - Multi-process


Process
14:14:24.128600
kworker/4:2
irq/36-p4p1-TxR
kworker/5:1
sshd
bash
test
logging
statistics
worker
A
B
C
D
E
F
G
H

Demos

Overhead

Metric	Transition	No transition
Ratio of requests	0.6%	99.4 %
Average latency	1136.93 ns	259.13 ns
Standard deviation	278.71 ns	28.42 ns
Minimum latency	565 ns	237 ns
Maximum latency	3028 ns	1938 ns
Average instruction count	2024	756
Average L1 misses	38.78	3.04
Average LLC misses	3.66	0.003
Average TLB misses	0.12	0.002
Average branch misses	3.08	0.15

Total : 8μs for 7 transitions

Overhead

Test	Baseline	Tracker	Overhead
CPU	19.20s	19.20s	0.00%
Memory	32.33s	32.37s	0.30%
File Read/Write	9.04 s	9.50 s	5.10%
Network 1Gbps	942Mbps/s	942Mbps/s	0.00%
Network 10Gbps	8.02Gbps/s	7.70Gbps/s	3.89%
OLTP (MySQL)	2.27s	2.38s	4.84%

Install it

```
apt-get install git gcc make
linux-headers-generic

git clone
https://github.com/efficios/latenc
y-tracker.git

cd latency-tracker

make

insmod latency_tracker.ko

insmod latency_tracker_rt.ko
```

Questions ?