Getting Started with

**LITMUS**

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

---

July 2016

Max Planck Institute for Software Systems

Björn B. Brandenburg
Manohar Vanga
Mahircan Gül
Agenda

The first decade of LITMUS\textsuperscript{RT}

2. Major Features
What sets LITMUS\textsuperscript{RT} apart?

3. Key Concepts
What you need to know to get started
What? Why? How?
The first decade of LITMUS\textsuperscript{RT}

— Part 1 —
What is LITMUS<sup>RT</sup>?

A real-time extension of the Linux kernel.
What is LITMUS\textsuperscript{RT}?

Linux kernel patch

+ 

user-space interface

+ 

tracing infrastructure
What is LITMUS$^{\text{RT}}$?

Linux kernel patch

+ 

user-space interface

+ 

tracing infrastructure

- RT schedulers
- RT synchronization
- [cache & GPU]

- C API
- device files
- scripts & tools

- overheads
- schedules
- kernel debug log
What is LITMUS\textsuperscript{RT}?

Linux kernel patch

+ user-space interface

+ tracing infrastructure

\begin{itemize}
\item RT schedulers
\item RT synchronization [cache & GPU]
\item C API
\item device files
\item scripts & tools
\item overheads
\item schedules
\item kernel debug log
\end{itemize}

Releases

2007.1
2007.2
2007.3
2008.1
2008.2
2008.3
2010.1
2010.2
2011.1
2012.1
2012.2
2012.3
2013.1
2014.1
2014.2
2015.1
2016.1
Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.
Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.

**practical and realistic:**

- Efficiently…
  - enable apples-to-apples comparison with existing systems (esp. Linux)

- ...support real applications…
  - I/O, synchronization, legacy code

- ...on real multicore hardware…
  - Realistic overheads on commodity platforms.

- ...in a real OS.
  - Realistic implementation constraints and challenges.
“At any point in time, the system schedules the m highest-priority jobs, where a job’s current priority is given by…”

Going from this…
assumptions on the state of the current task since it may be called for a
number of reasons. The reasons include a scheduler_tick() determined that it
was necessary, because sys_exit_np() was called, because some Linux
* subsystem determined so, or even (in the worst case) because there is a bug
* hidden somewhere. Thus, we must take extreme care to determine what the
* current state is.
* The CPU could currently be scheduling a task (or not), be linked (or not).
* The following assertions for the scheduled task could hold:
* - !is_running(scheduled) // the job blocks
* - scheduled->timeslice == 0 // the job completed (forcefully)
* - get_rt_flag() == RT_F_SLEEP // the job completed (by syscall)
* - linked != scheduled // we need to reschedule (for any reason)
* - is_np(scheduled) // rescheduling must be delayed,
  sys_exit_np must be requested
* Any of these can occur together.
*/
static struct task_struct* gsnedf_schedule(struct task_struct* prev)
{
    cpu_entry_t* entry = &get_cpu_var(gsnedf_cpu_entries);
    int out_of_time, sleep, preempt, np, exists, blocks;
    struct task_struct* next = NULL;
#ifdef CONFIG_RELEASE_MASTER
/* Bail out early if we are the release master. *
 * The release master never schedules any real-time tasks. */
    if (unlikely(gsnedf.release_master == entry->cpu)) {
        sched_state_task_picked();
        return NULL;
    }
#endif
    raw_spin_lock(&gsnedf_lock);

    /* sanity checking */
    BUG_ON(entry->scheduled & entry->scheduled != prev);
    BUG_ON(entry->scheduled & !is_realtime(prev));
    BUG_ON(!is_realtime(prev) & !entry->scheduled);

    /* (0) Determine state */
    exists = entry->scheduled != NULL;
    blocks = entry->is_running(entry->scheduled);
    out_of_time = exists & !budget_enforced(entry->scheduled);
    &budget_exhausted(entry->scheduled);
    np = exists & !is_np(entry->scheduled);
    sleep = exists & get_rt_flag(entry->scheduled) == RT_F_SLEEP;
    preempt = entry->scheduled != entry->linked;

#ifdef WANT_ALL_SCHED_EVENTS
    TRACE_TASK(prev, "invoked gsnedf_schedule.
"");
#endif

    if (exists)
        TRACE_TASK(prev,
            "blocks:\d out_of_time:\d np:\d sleep:\d preempt:\d "
            "state:\d sigd:\d"
        ,
            blocks, out_of_time, np, sleep, preempt,
            prev->state, signal_pending(prev));

    if (entry->linked & preempt)
        TRACE_TASK(prev, "will be preempted by %s/dn",
                entry->linked->comm, entry->linked->pid);

    ... to this!

G-EDF: measured scheduling overhead for 3 tasks per processor (host=ludwig)
min=0.79us max=314.60us avg=54.63us median=40.15us stddev=45.83us
samples: total=560090
[IQR filter not applied]

Global EDF
scheduler overhead
for 72 task on 24 cores
Why You Should Be Using LITMUS<sup>RT</sup>

If you are doing kernel-level work anyway...
- Get a *head-start* — simplified kernel interfaces, debugging infrastructure, user-space interface, tracing infrastructure
- As a *baseline* — compare with schedulers in LITMUS<sup>RT</sup>

If you are developing real-time applications...
- Get a predictable execution environment with "textbook algorithms" matching the literature
- Isolate processes with *reservation-based scheduling*
- Understand *kernel overheads* with just a few commands!

If your primary focus is *theory and analysis*...
- To understand the impact of *overheads*.
- To *demonstrate practicality* of proposed approaches.
Theory vs. Practice

Why is implementing “textbook” schedulers difficult?

Besides the usual kernel fun:
restricted environment, special APIs, difficult to debug, …
Scheduler: a function that, at each point in time, maps elements from the set of ready jobs onto a set of \( m \) processors.
Scheduling in Theory

Scheduler: a function that, at each point in time, maps elements from the set of ready jobs onto a set of \( m \) processors.

Global policies based on global state
- E.g., “At any point in time, the \( m \) highest-priority…”

Sequential policies, assuming total order of events.
- E.g., “If a job arrives at time \( t \)…”
Scheduling in Theory

Practical scheduler: job assignment changes only in response to well-defined scheduling events (or at well-known points in time).
Scheduling in Practice

Ready Queue

CPU 1
event 1

CPU 2
event 2

CPU m
event m

... schedule()...

CPU 1

CPU 2

CPU m
Scheduling in Practice

Each processor schedules only itself locally.

- Multiprocessor schedulers are parallel algorithms.
- Concurrent, unpredictable scheduling events!
- New events occur while making decision!
- No globally consistent atomic snapshot for free!
Original Purpose of LITMUS RT

**Theory**

- Develop efficient implementations.

**Practice**

- Inform: what works well and what doesn’t?

---

MPI-SWS
## History — The first Ten Years

<table>
<thead>
<tr>
<th>Releases</th>
<th>Calandrino et al. (2006)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RTSS’06]</td>
<td>[not publicly released]</td>
</tr>
</tbody>
</table>

2007.1  
2007.2  
2007.3  
2008.1  
2008.2  
2008.3  
2010.1  
2010.2  
2011.1  
2012.1  
2012.2  
2012.3  
2013.1  
2014.1  
2014.2  
2015.1  
2016.1

---

Project initiated by **Jim Anderson** (UNC); first prototype implemented by John Calandrino, Hennadiy Leontyev, Aaron Block, and Uma Devi.

**Graciously supported over the years by:**
NSF, ARO, AFOSR, AFRL, and Intel, Sun, IBM, AT&T, and Northrop Grumman Corps.

**Thanks!**
History — The first Ten Years

Releases
[RTSS’06] Calandrino et al. (2006) [not publicly released]
2007.1
2007.2
2007.3
2008.1
2008.2
2008.3
2010.1
2010.2
2011.1
2012.1
2012.2
2012.3
2013.1
2014.1
2014.2
2015.1
2016.1

Continuously maintained
- reimplemented for 2007.1
- 17 major releases spanning 40 major kernel versions (Linux 2.6.20 — 4.1)

Impact
- used in 50+ papers, and 7 PhD & 3 MSc theses
- several hundred citations
- used in South & North America, Europe, and Asia
Goals and Non-Goals

Goal: Make life easier for real-time systems researchers
- LITMUS\textsuperscript{RT} always was, and remains, primarily a research vehicle
- encourage systems research by making it more approachable

Goal: Be sufficiently feature complete & stable to be practical
- no point in evaluating systems that can’t run real workloads

Non-Goal: POSIX compliance
- We provide our own APIs — POSIX is old and limiting.

Non-Goal: API stability
- We rarely break interfaces, but do it without hesitation if needed.

Non-Goal: Upstream inclusion
- LITMUS\textsuperscript{RT} is neither intended nor suited to be merged into Linux.
Major Features

What sets LITMUS<sup>RT</sup> apart?

— Part 2 —
Partitioned vs. Clustered vs. Global
real-time multiprocessor scheduling approaches

partitioned scheduling

clustered scheduling

global scheduling
Predictable Real-Time Schedulers

*Matching the literature!*

- Global EDF
- Pfair (PD²)
- Clustered EDF
- Partitioned EDF
- Partitioned Fixed-Priority (FP)
- Partitioned Reservation-Based polling + table-driven

*maintained in mainline LITMUSRT*
Predictable Real-Time Schedulers

Matching the literature!

- Global EDF
- Pfair (PD²)
- Clustered EDF
- Partitioned EDF
- Partitioned Fixed-Priority (FP)
- Partitioned Reservation-Based polling + table-driven

maintained in mainline LITMUSR T

- Global & Clustered Adaptive EDF
- Global FIFO
- RUN
- slot shifting
- QPS
- MC²

- Global Message-Passing EDF & FP
- Strong Laminar APA FP
- EDF-HSB
- EDF-WM
- EDF-fm
- EDF-C=D

... Sporadic Servers

CBS
CASH
soft-polling
slack sharing

external branches & patches / paper-specific prototypes
Jump-Start Your Research

Bottom line:
➡ The scheduler that you need might already be available.

(Almost) never start from scratch:
➡ If you need to implement a new scheduler, there likely exists a good starting point (e.g., of similar structure).

Plenty of baselines:
➡ At the very least, LITMUS$RT$ can provide you with interesting baselines to compare against.
Predictable Locking Protocols

Matching the literature!

SRP  MPCP-VS
FMLP+  DPCP
PCP  DFLP
MPCP

non-preemptive spin locks

MC-IPC
MBWI
Global OMLP
OMIP
RNLP
Clustered OMLP
k-exclusion locks

maintained in mainline LITMUS^RT

external branches & patches / paper-specific prototypes

MPI-SWS
Lightweight Overhead Tracing

minimal static trace points

+ binary rewriting ($\text{jmp} \leftrightarrow \text{nop}$)

+ per-processor, wait-free buffers
Evaluate Your Workload with Realistic Overheads

**Partitioned EDF context-switch overhead**

- **Summary:**
  - **Host:** ludwig
  - **Tasks per Processor:** 3
  - **Data:**
    - **Minimum (min):** 0.63us
    - **Maximum (max):** 44.59us
    - **Average (avg):** 5.70us
    - **Median:** 5.39us
    - **Standard Deviation (stdev):** 2.39us
  - **Samples:**
    - **Total:** 560087
    - **Filtered:** 14 (0.00%)
  - **IQR:**
    - **Extent:** 12
    - **Threshold:** 46.30us

**Global EDF context-switch overhead**

- **Summary:**
  - **Host:** ludwig
  - **Tasks per Processor:** 3
  - **Data:**
    - **Minimum (min):** 0.62us
    - **Maximum (max):** 37.74us
    - **Average (avg):** 5.52us
    - **Median:** 5.31us
    - **Standard Deviation (stdev):** 2.10us
  - **Samples:**
    - **Total:** 560087
    - **Filtered:** 105 (0.02%)
  - **IQR:**
    - **Extent:** 12
    - **Threshold:** 37.80us

**Partitioned EDF job release overhead**

- **Summary:**
  - **Host:** ludwig
  - **Tasks per Processor:** 3
  - **Data:**
    - **Minimum (min):** 0.27us
    - **Maximum (max):** 23.54us
    - **Average (avg):** 5.48us
    - **Median:** 4.93us
    - **Standard Deviation (stdev):** 2.72us
  - **Samples:**
    - **Total:** 152059
    - **Filtered:** (IQR filter not applied)

**Global EDF job release overhead**

- **Summary:**
  - **Host:** ludwig
  - **Tasks per Processor:** 3
  - **Data:**
    - **Minimum (min):** 1.75us
    - **Maximum (max):** 291.17us
    - **Average (avg):** 62.05us
    - **Median:** 43.40us
    - **Standard Deviation (stdev):** 52.43us
  - **Samples:**
    - **Total:** 152059
    - **Filtered:** (IQR filter not applied)

*Note the scale!*
Automatic Interrupt Filtering

Overhead tracing, ideally:

- start timestamp
- measured activity
- end timestamp

With outliers:

- start timestamp
- ISR
- end timestamp

noise due to untimely interrupt
Automatic Interrupt Filtering

Overhead tracing, ideally:

- start timestamp
- measured activity
- end timestamp

With outliers:

- start timestamp
- ISR
- end timestamp

How to cope?
- can’t just turn off interrupts
- Used statistical filters…
  - …but which filter?
  - … what if there are true outliers?

Since LITMUS\textsuperscript{RT} 2012.2:
- ISRs increment counter
- timestamps include counter snapshots & flag
- interrupted samples discarded automatically
Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI):

Core 1: start timestamp

... IPI

Core 27: end timestamp
Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI), with non-aligned clock sources:

Core 1

start timestamp

... 

Core 27

dot-dashed arrow to 1000

IPI

IPI received before it was sent!? 
[→ overflows to extremely large outliers]

dot-dashed arrow to 990

end timestamp
Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI), with non-aligned clock sources:

Core 1

start timestamp

Core 27

end timestamp

IPI

IPI received before it was sent!? 
[→ overflows to extremely large outliers]

In LITMUSRT, simply run `ftcat -c` to measure and automatically compensate for unaligned clock sources.
Lightweight Schedule Tracing

- task parameters
  - context switches & blocking
  - job releases & deadlines & completions

Built on top of: feather trace
Schedule Visualization: \texttt{st-draw}

Ever wondered what a Pfair schedule looks like in reality?
Schedule Visualization: \texttt{st-draw}

Ever wondered what a Pfair schedule looks like in reality?
Easy! Just record the schedule with \texttt{sched_trace} and run \texttt{st-draw}!

Note: this is \textit{real} execution data from a 4-core machine, \textit{not} a simulation! [Color indicates CPU identity].
Easy Access to Workload Statistics

“We traced the resulting schedules using LITMUSRT’s sched_trace facility and recorded the response times of more than 45,000,000 individual jobs.”

[—, “A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications”, ECRTS’13]
Easy Access to Workload Statistics

“We traced the resulting schedules using LITMUSRT sched_trace facility and recorded the response times of more than 45,000,000 individual jobs.”

[—, “A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications”, ECRTS’13]

(1) **st-trace-schedule** my-ecrts13-experiments-OMIP

[...run workload...]

(2) **st-job-stats** *my-ecrts13-experiments-OMIP*.bin

| Task, Job, Period, Response, DL Miss?, Lateness, Tardiness, Forced?, ACET |
|---------------------------------------------|------------------|------------------|------------------|------------------|
| task NAME=rtspin PID=29587 COST=1000000 PERIOD=1000000 CPU=0 |
| 29587, 2, 10000000, 1884, 0, -9998116, 0, 0, 1191 |
| 29587, 3, 10000000, 1019692, 0, -8980308, 0, 0, 1017922 |
| 29587, 4, 10000000, 1089789, 0, -8910211, 0, 0, 1030550 |
| 29587, 5, 10000000, 1034513, 0, -8965487, 0, 0, 1016656 |
| 29587, 6, 10000000, 1032825, 0, -8967175, 0, 0, 1016096 |
| 29587, 7, 10000000, 1037301, 0, -8962699, 0, 0, 1016078 |
| 29587, 8, 10000000, 1033699, 0, -8966301, 0, 0, 1016535 |
| 29587, 9, 10000000, 1037287, 0, -8962713, 0, 0, 1015794 |

...
Easy Access to Workload Statistics

“We traced the resulting schedules using LITMUS\textsuperscript{RT} sched\_trace facility and recorded the response times of more than 45,000,000 individual jobs.”

[—, “A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications”, ECRTS’13]

(1) \texttt{st-trace-schedule my-ecrts13-experiments-OMIP} \\
[...run workload...]

(2) \texttt{st-job-stats *my-ecrts13-experiments-OMIP*.bin}

<table>
<thead>
<tr>
<th>Task, Job, Period,</th>
<th>Response, DL Miss?, Lateness, Tardiness, Forced?, ACET</th>
</tr>
</thead>
<tbody>
<tr>
<td>29587, 2, 10000000, 1884, 0, -9998116, 0, 0, 1191</td>
<td></td>
</tr>
<tr>
<td>29587, 3, 10000000, 1019692, 0, -8980308, 0, 0, 1017922</td>
<td></td>
</tr>
<tr>
<td>29587, 4, 10000000, 1089789, 0, -8910211, 0, 0, 1030550</td>
<td></td>
</tr>
<tr>
<td>29587, 5, 10000000, 1034513, 0, -8965487, 0, 0, 1016656</td>
<td></td>
</tr>
<tr>
<td>29587, 6, 10000000, 1032825, 0, -8967175, 0, 0, 1016096</td>
<td></td>
</tr>
<tr>
<td>29587, 7, 10000000, 1037301, 0, -8962699, 0, 0, 1016078</td>
<td></td>
</tr>
<tr>
<td>29587, 8, 10000000, 1033699, 0, -8966301, 0, 0, 1016535</td>
<td></td>
</tr>
<tr>
<td>29587, 9, 10000000, 1037287, 0, -8962713, 0, 0, 1015794</td>
<td></td>
</tr>
</tbody>
</table>

How long did each job use the processor?
Synchronous Task System Releases

all tasks release their first job at a common time “zero.”
Synchronous Task System Releases

int wait_for_ts_release(void);

→ task sleeps until synchronous release

int release_ts(lt_t *delay);

→ trigger synchronous release in <delay> nanoseconds
Asynchronous Releases with Phase/Offset

*LITMUS*\textsuperscript{RT} also supports non-zero phase/offset.

⇒ release of first job occurs with some *known* offset after task system release.

release of *first job* is staggered w.r.t. time “zero”

⇒ *can use schedulability tests for asynchronous periodic tasks*
Easier Starting Point for New Schedulers

simplified scheduler plugin interface

```c
struct sched_plugin {
    [..]
    schedule_t    schedule;
    finish_switch_t finish_switch;
    [..]
    admit_task_t   admit_task;
    fork_task_t    fork_task;
    task_new_t     task_new;
    task_wake_up_t task_wake_up;
    task_block_t   task_block;
    task_exit_t    task_exit;
    task_cleanup_t task_cleanup;
    [..]
}
```

simplified interface  
+  
richer task model  
+  
plenty of working code to steal from
Many More Features...

Support for true global scheduling
- supports proper pull-migrations
  - moving tasks among Linux’s per-processor runqueues
- Linux’s SCHED_FIFO and SCHED_DEADLINE global scheduling “emulation” is not 100% correct (races possible)

Low-overhead non-preemptive sections
- Non-preemptive spin locks without system calls.

Wait-free preemption state tracking
- “Does this remote core need to be sent an IPI?”
- Simple API suppresses superfluous IPIs

Debug tracing with TRACE()
- Extensive support for “printf() debugging” → dump from Qemu
LITMUS\textsuperscript{RT}

Predictable execution platform and research accelerator.

Apply \textit{schedulability} analysis

+ under consideration of \textit{overheads}

+ \textit{jump-start} your development
Key Concepts

What you need to know to get started

— Part 3 —
Scheduler Plugins

Linux scheduler classes:

- SCHED_LITMUS
- SCHED_DEADLINE
- SCHED_FIFO/RR
- SCHED_OTHER (CFS)
- SCHED_IDLE

LITMUS\textsuperscript{RT} plugins:

- Linux (dummy)
- PSN-EDF
- GSN-EDF
- C-EDF
- P-FP
- P-RES

SCHED\_LITMUS “class” invokes active plugin.

- LITMUS\textsuperscript{RT} tasks have highest priority.
- SCHED\__DEADLINE & SCHED\_FIFO/RR:
  - best-effort from SCHED\_LITMUS point of view
Plugin Switch

Active plugin can be switched at runtime.

- But only if no real-time tasks are present.

Linux scheduler classes:

- SCHED_LITMUS
- SCHED_DEADLINE
- SCHED_FIFO/RR
- SCHED_OTHER (CFS)
- SCHED_IDLE

Active plugin:

$ setsched PSN-EDF

LITMUS$RT plugins:

- Linux (dummy)
- PSN-EDF
- GSN-EDF
- C-EDF
- P-FP
- P-RES

MPI-SWS
Classic Process-Based Plugins

Plugin manages real-time tasks **directly**
- one sporadic task (scheduling theory)
  \[= exactly\] one process in Linux

$\text{rt\_launch} \ -p \ 1 \ 10 \ 100 \ <\text{my-app-binary}>$

**WCET**
Reservation-Based Scheduling (RBS)

Plugin schedules reservations
(= process containers)
  ➞ one sporadic task in analysis
    = one reservation (→ 0..n processes)
  ➞ second-level dispatcher in each reservation

SCHED_LITMUS active plugin

$ resctl -n 123 -c 1 -b 10 -p 100
$ rt_launch -p 1 -r 123 <my-app-component-1>
$ rt_launch -p 1 -r 123 <my-app-component-2>
[…]

arbitrary ID
core
budget
period

P-RES
RBS: Motivation 1/2

**Temporal** Decomposition: Sequential, Recurrent Tasks

- sequential tasks: basic element of RT scheduling theory

**VS**

**Logical** Decomposition: Software Modularization

- split complex applications into many *logical* modules or components to manage spheres of responsibility

Example: one real-time “task” may consist of multiple processes

*middleware process + redundant, isolated application threads + database process*
Irregular execution patterns

- e.g., http server triggered by unpredictable incoming requests
- with RBS, can safely encapsulate arbitrary activation patterns

“Legacy” and complex software

- Cannot require existing, complex software (e.g., ROS) to adopt and comply with LITMUS\textsuperscript{RT} API
- with RBS, can transparently encapsulate any Linux process
  - Even kernel threads (e.g., interrupt threads)!

Worst-case execution time is a fantasy

- Most practical systems must live with imperfect measurements and cannot always provision on a (measured) worst-case basis
- with RBS, can manage overruns predictably and actively exploit slack
Three Main Repositories

Linux kernel patch
→ litmus-rt

+ 

user-space interface
→ liblitmus

+ 

tracing infrastructure
→ feather-trace-tools
liblitmus: The User-Space Interface

C API (task model + system calls)

+ user-space tools

→ setsched, showsched, release_ts, rt_launch, rtspin
/proc/litmus/* and /dev/litmus/*

/proc/litmus/*
➡ Used to export information about the plugins and existing real-time tasks.
➡ Read- and writable files.
➡ Typically managed by higher-level wrapper scripts.

/dev/litmus/*
➡ Special device files based on custom character device drivers.
➡ Primarily, export **trace data** (use only with *ftcat*):
  ‣ *ft_cpu_traceX* — core-local overheads of CPU X
  ‣ *ft_msg_traceX* — IPIs related to CPU X
  ‣ *sched_traceX* — scheduling events on CPU X
➡ *log* — debug trace (use with regular *cat*)
Control Page: /dev/litmus/ctrl

A (private) per-process page mapped by each real-time task

- Shared memory segment between kernel and task.
- Purpose: **low-overhead communication channel**
- interrupt count
- *preemption-disabled* and *preemption-needed* flags
- current deadline, etc.

**Second purpose, as of 2016.1**

- implements LITMUS\textsuperscript{RT} “system calls” as ioctl() operations
- improves portability and reduces maintenance overhead

**Transparent use**

- liblitmus takes care of everything
(Lack of) Processor Affinities

In Linux, each process has a processor affinity mask.

\[ X^{th} \text{ bit set} \Rightarrow \text{process may execute on core } X \]

Most \textbf{LITMUS}RT plugins ignore affinity masks.

- In particular, all plugins in the mainline version do so.
  - *Global is global; partitioned is partitioned*...

Recent out-of-tree developments

- Support for \textit{hierarchical} affinities [ECRTS’16]
Things That Are Not Supported

With limited resources, we cannot possibly support & test all Linux features.

Architectures other than x86 and ARM

➡ Though not difficult to add support if someone cares…

Running on top of a hypervisor

➡ It works (→ hands-on session), but it’s not “officially” supported.
➡ You can use LITMUS\textsuperscript{RT} as a real-time hypervisor by encapsulating \texttt{kvm} in a reservation.

CPU Hotplug

➡ Not supported by existing plugins.

Processor Frequency Scaling

➡ Plugins “work,” but oblivious to speed changes.

Integration with PREEMPT\_RT

➡ For historic reasons, the two patches are incompatible
➡ Rebasing on top of PREEMPT\_RT has been on the wish list for some time…
Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.

Connect theory and practice.  
Don’t reinvent the wheel.

Use **LITMUS**\textsuperscript{RT} as a baseline.