Getting Started with

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

presented at
TuToR 2016 @ CPSWeek 2016
Vienna, Austria, April 11, 2016

Björn B. Brandenburg
Manohar Vanga
Mahircan Gül
Agenda

   The first decade of LITMUS<sub>RT</sub>

2. Major Features
   What sets LITMUS<sub>RT</sub> apart?

3. Key Concepts
   What you need to know to use LITMUS<sub>RT</sub>
What? Why? How?
The first decade of LITMUS\textsuperscript{RT}

— Part 1 —
What is LITMUS\textsuperscript{RT}?

A real-time extension of the Linux kernel.
What is LITMUSRT?

Linux kernel patch

+ 

user-space interface

+ 

tracing infrastructure
What is LITMUS\textsuperscript{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

- RT schedulers
- RT synchronization
  - [cache & GPU]
- C API
- device files
- scripts & tools
- overheads
- schedules
- kernel debug log

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  = exists & !is_running(entry->scheduled);
    out_of_time = exists & budget_exhausted(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
    TRACETASK(prev, "invoked gsnedf_schedule.\n"");
#endif

    if (exists)
        TRACETASK(prev,
            "blocks:%d out_of_time:%d np:%d sleep:%d preempt:%d "
            "state:%d sig:%d\n",
            blocks, out_of_time, np, sleep, preempt,
            prev->state, signal_pending(prev));

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

    samples: total=560090
    [IQR filter not applied]

G-EDF: measured scheduling overhead for 3 tasks per processor (host=ludwig)
    min=0.79us max=314.60us avg=54.83us median=40.15us stdev=45.83us

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
- 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.
Why is implementing “textbook” schedulers difficult?

Besides the usual kernel fun:
 restricted environment, special APIs, difficult to debug, …
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.
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 \( t \)…”
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.

Inform: what works well and what doesn’t?

**Practice**
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

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

Continuous maintained

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

Impact

► used in about 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, 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 cumbersome.

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 LITMUS$^RT$*
# Predictable Real-Time Schedulers

*Matching the literature!*

<table>
<thead>
<tr>
<th>Global EDF</th>
<th>Pfair (PD²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered EDF</td>
<td></td>
</tr>
<tr>
<td>Partitioned EDF</td>
<td></td>
</tr>
<tr>
<td>Partitioned Fixed-Priority (FP)</td>
<td></td>
</tr>
<tr>
<td>Partitioned Reservation-Based polling + table-driven</td>
<td></td>
</tr>
</tbody>
</table>

*maintained in mainline* **LITMUS**

<table>
<thead>
<tr>
<th>Global &amp; Clustered Adaptive EDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global FIFO</td>
</tr>
<tr>
<td>Global FP</td>
</tr>
<tr>
<td>RUN</td>
</tr>
<tr>
<td>slot shifting</td>
</tr>
<tr>
<td>QPS</td>
</tr>
<tr>
<td>Global Message-Passing EDF &amp; FP</td>
</tr>
<tr>
<td>Strong Laminar APA FP</td>
</tr>
<tr>
<td>EDF-HSB</td>
</tr>
<tr>
<td>EDF-WM</td>
</tr>
<tr>
<td>EDF-fm</td>
</tr>
<tr>
<td>EDF-C=D</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Sporadic Servers</td>
</tr>
<tr>
<td>CBS</td>
</tr>
<tr>
<td>CASH</td>
</tr>
<tr>
<td>soft-polling</td>
</tr>
<tr>
<td>slack sharing</td>
</tr>
<tr>
<td>external branches &amp; patches / paper-specific prototypes</td>
</tr>
</tbody>
</table>
Easily Compare Your Work

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\textsuperscript{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

maintained in mainline LITMUSRT

MC-IPC  MBWI
Global OMLP  RNLP
OMIP
Clustered OMLP

k-exclusion locks

external branches & patches / paper-specific prototypes
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 [72 tasks, 24 cores]

Partitioned EDF job release overhead [72 tasks, 24 cores]

Global EDF context-switch overhead [72 tasks, 24 cores]

Global EDF job release overhead [72 tasks, 24 cores]

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

noise due to untimely interrupt

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

... → 1000

Core 27

end timestamp

IPI received before it was sent!? [→ overflows to extremely large outliers] → 990
Cycle Counter Skew Compensation

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

In **LITMUS**\textsuperscript{RT}, 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: st-draw

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

Note: this is *real* execution data from a 4-core machine, not a simulation! [*Color indicates CPU identity*].
"We traced the resulting schedules using LITMUS\textsuperscript{RT}'s \texttt{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 LITMUS^{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) 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}

\begin{verbatim}
# 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, 1036399, 0, -8966301, 0, 0, 1016535
29587, 9, 10000000, 1037287, 0, -8962713, 0, 0, 1015794
...
\end{verbatim}

How long did each job use the processor?
all tasks release their *first job* at a common time “zero.”
**Synchronous Task System Releases**

```c
int wait_for_ts_release(void);

→ task sleeps until synchronous release
```

```c
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"

\textit{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;  
[...]  
admmit_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
LITMUS\textsuperscript{RT}: Development Accelerator

Many common tasks have already been taken care of.

Explicit support for sporadic task model

\begin{itemize}
\item The kernel knows WCETs, periods, deadlines, phases etc.
\end{itemize}

Support for true global scheduling

\begin{itemize}
\item supports proper \textbf{pull-migrations}
  \begin{itemize}
  \item moving tasks among Linux’s per-processor runqueues
  \end{itemize}
\item Linux’s \texttt{SCHED\_FIFO} and \texttt{SCHED\_DEADLINE} global scheduling
  “emulation” is not 100\% correct (\texttt{races possible})
\end{itemize}

Low-overhead non-preemptive sections

\begin{itemize}
\item Non-preemptive spin locks \texttt{without system calls}.
\end{itemize}

Wait-free preemption state tracking

\begin{itemize}
\item “\texttt{Does this remote core need to be sent an IPI}?”
\item Simple API suppresses superfluous IPIs
\end{itemize}

Debug tracing with \texttt{TRACE()}

\begin{itemize}
\item Extensive support for \texttt{``printf() debugging''} \texttt{\rightarrow dump from Qemu}
\end{itemize}
Key Concepts

What you need to know to use LITMUS\textsuperscript{RT}

— Part 3 —
Scheduler Plugins

Linux scheduler classes:

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

pick_next_task() → active plugin

SCHED_LITMUS “class” invokes active plugin.

- LITMUSRT tasks have highest priority.
- SCHED_DEADLINE & SCHED_FIFO/RR:
  - best-effort from SCHED_LITMUS point of view

LITMUSRT plugins:

- Linux (dummy)
- PSN-EDF
- GSN-EDF
- C-EDF
- P-FP
- P-RES
Active plugin can be switched at runtime.

But only if no real-time tasks are present.

$\text{setsched PSN-EDF}$
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 LITMUSRT "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.

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

Most \text{LITMUS}^{\text{RT}} \text{ plugins ignore affinity masks.}

- In particular, all plugins in the mainline version do so.
  - \textit{Global is global; partitioned is partitioned}…

Recent out-of-tree developments

- Support for \textit{hierarchical} affinities [submitted to 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
➤ Though running on top of RT Xen seems to work now…
➤ 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.
What to expect in the hands-on session

Focus: using LITMUSRT as a development platform

- activating plugins
- running real-time tasks
- schedule tracing
- writing custom tasks
- (overhead tracing)

Out of scope: kernel hacking

- takes more than 90 minutes…
- Mastery of user-space is precursor to plugin development anyway.

If you have questions later, stop by our friendly mailinglist!

www.litmus-rt.org/tutor16

tutorial manual & slides available!
Focus: using LITMUS\textsuperscript{RT} as a development platform

- activating plugins
- running real-time tasks
- schedule tracing
- writing custom tasks
- (overhead tracing)

Out of scope: kernel hacking

- takes more than 90 minutes…
- Mastery of user-space is precursor to plugin development anyway.

If you have questions later, stop by our friendly mailinglist!