A Tour of LITMUS^{RT}

Björn B. Brandenburg, Mahircan Gül, and Manohar Vanga

LITMUS^{RT} is the ***Li**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems*. Its purpose is to provide a foundation for applied real-time systems research. To this end, LITMUS^{RT}

1. adds **predictable scheduling and synchronization policies** to the Linux kernel,
2. reduces the effort involved in **prototyping** new scheduling and synchronization polices in the kernel by offering a simplified scheduling policy plugin interface,
3. provides `liblitmus` a **userspace library** to expose the added kernel functionality to applications, and
4. comes with *Feather-Trace*, a **low-overhead tracing** framework.

This guide provides an introduction to the user-facing parts of LITMUS^{RT}, namely (3) and (4), and assumes that the kernel (with the LITMUS^{RT} modifications in place) has already been compiled and is working. How to modify the kernel itself or how to develop new policy plugins is beyond the scope of this guide.

This guide describes LITMUS^{RT} as of version 2016.1 (released in April 2016).

Booting

LITMUS^{RT} boots just like any other Linux system. In most cases, no modifications need to be made to the Linux distribution's init system.

Linux supports a large number of boot parameters that can be passed to the kernel as a kernel command line argument. Most of these parameters are irrelevant to LITMUS^{RT}; however, one of them, the `maxcpus` parameter, is particularly useful and worth highlighting.

The `maxcpus` parameter allows limiting the number of processors used by the kernel. This can be used to instruct the kernel to leave some of the available processor cores unused. For instance, this is especially useful when conducting scalability experiments where the overhead of a scheduler (or some other component) needs to be obtained for a range of core counts.

(Note: there also exists a static maximum supported processor count called `NR_CPUS` that is set as part of the kernel configuration at compile time. The `maxcpus` parameter cannot exceed `NR_CPUS`.)

Schedulers

As already mentioned, LITMUS^{RT} augments the Linux kernel with a number of predictable scheduling policies, which are realized as **scheduler plugins**.

At all times, the **active plugin** determines the current scheduling policy in a LITMUS^{RT} kernel. When the system boots up, it initially uses the dummy *Linux* plugin, which simply disables all LITMUS^{RT} functionality (which is not needed during bootup). Hence, before using any LITMUS^{RT} features, one of the available policy plugins must be activated with a **plugin switch** (as discussed in more detail below).

As of version 2016.1, the following real-time scheduling policies are built into the LITMUS^{RT} kernel:

- Linux : A dummy scheduling policy for disabling real-time functionality
- P-FP : Partitioned, *fixed-priority* scheduling
- PSN-EDF : Partitioned, dynamic-priority *earliest-deadline first* (EDF) scheduling
- GSN-EDF : Global EDF scheduling
- C-EDF : Clustered EDF scheduling, a hybrid of partitioned and global EDF scheduling
- PFAIR : Proportionate fair (Pfair) scheduling, based on the PD^2 algorithm
- P-RES : Reservation-based scheduling plugin, which supports a set of *partitioned* uniprocessor reservations:
  - periodic polling server
  - sporadic polling server
  - table-driven reservations

These policy plugins form two groups: whereas the **classic plugins** — P-FP, PSN-EDF, GSN-EDF, C-EDF, and PFAIR — are *process-based* (i.e., one real-time "task" = one single-threaded Linux process), the much more recent **reservation-based plugin** P-RES implements first-class reservations (i.e., one real-time "task" = any number of threads/processes).

The classic plugins have been a part of LITMUS^{RT} since the first version from 2006/2007 and have been used in virtually all prior studies based on LITMUS^{RT}. However, they impose certain limitations that may render them unsuitable for certain workloads (e.g., when a real-time process forks, the child process is *not* a real-time process to avoid overload). We expect future developments to focus increasingly on reservation-based plugins.

While P-FP, PSN-EDF, P-RES, and GSN-EDF are built into the kernel by default, C-EDF and PFAIR are optional and can be de-selected during kernel configuration. During boot-up, built-in policies are loaded by the kernel and can then be later activated (one at a time) by the root user.

At runtime, scheduler plugins are managed via a simple `/proc` interface that is exposed under `/proc/litmus`. All files under under `/proc/litmus` can be manually read or written for management purposes; however it is often easier and more convenient to use higher-level tools offered by `liblitmus`. In the following, we discuss a list of basic user-space commands related to plugin management, followed by an example.

All the commands mentioned in the following must be executed as root.

Listing All Available Schedulers

All loaded schedulers can be listed by running `cat /proc/litmus/plugins/loaded`.

For example:

```
root@litmus:~ cat /proc/litmus/plugins/loaded
PFAIR
P-FP
P-RES
PSN-EDF
GSN-EDF
Linux
```

Displaying the Active Scheduler Plugin

The currently active scheduler is reported by the `showsched` tool. Alternatively, it can be printed with `cat /proc/litmus/active_plugin`.

Example:

```
root@litmus:~ showsched
Linux
root@litmus:~ cat /proc/litmus/active_plugin
Linux
```

After boot-up, the active scheduler is set to `Linux` by default.

Changing the Active Scheduler Plugin

The active scheduler plugin can be selected with the `setsched` tool. Switching scheduling policies is permitted only when there are no active real-time tasks in the system.

To activate a specific policy, invoke `setsched` with the policy name as the first argument. For example, to activate the GSN-EDF plugin, run `setsched GSN-EDF`.

If `setsched` succeeds, then there is no output, as is typical for UNIX tools.

Example:

```
root@litmus:~ setsched GSN-EDF
root@litmus:~ showsched
GSN-EDF
root@litmus:~
```

To be presented with a text-based interface for selecting the active scheduler from the list of available plugins, run `setsched` without an argument.

There can be only one active policy at any time.

Selecting the C-EDF and PFAIR Cluster Sizes

Under clustered scheduling, non-overlapping sets of processors are scheduled independently with a “global” policy. In LITMUS^{RT}, clusters are created based on cache topology.

The C-EDF scheduler builds clusters of processors *at activation time* based on the *current* value of the *cluster file* `/proc/litmus/plugins/C-EDF/cluster`. Valid values are “L1”, “L2”, “L3”, or “ALL”, where “ALL” indicates global scheduling (i.e., with “ALL” a single cluster containing all processor cores is built). On processors with private L1 caches, “L1” corresponds to partitioned scheduling (i.e., in this case, C-EDF builds one cluster for each processor).

The PFAIR plugin also supports clustered scheduling based on the corresponding cluster file `/proc/litmus/plugins/PFAIR/cluster`.

Note that clusters are configured when a plugin is activated, i.e, the cluster size cannot be changed while a plugin is active. To be effective, the desired cluster size has to be written to the cluster file *before* the scheduler switch.

If a change in cluster size is desired, first switch to the *Linux* plugin, update the cluster file, and then switch back to the clustered plugin. For example:

```
# switch to Linux plugin
setsched Linux
# configure L2-based clusters
echo L2 > /proc/litmus/plugins/C-EDF/cluster
# switch back to C-EDF
setsched C-EDF
```

Working with Reservations in P-RES

In contrast to other LITMUS^{RT} plugins, reservations are first-class entities in `P-RES` that exist independently of tasks. In particular, they must be created *before* any task can be launched, and they continue to exist even after all tasks have terminated. Multiple tasks or threads can be assigned to the same reservation. If a task forks, both the parent and the child remain in the same reservation.

Partitioned Reservations

To create new reservations, use the tool `resctl`. The tool has an online help message that can be triggered with `resctl -h`, which explains all options. In the following, we explain how to create partitioned, per-processor reservations.

Reservation Types

The current version of the `P-RES` plugin supports three reservation types:

```
polling-periodic (PP)
polling-sporadic (PS)
table-driven (TD)
```

Additional common reservations types (e.g., CBS, sporadic servers, etc.) have been developed in a separate branch and are expected to be released in a future version of LITMUS^{RT}.

The most simple reservation type is the polling reservation, which comes in two flavors: classic *periodic polling reservations* (PP), and more flexible *sporadic polling reservations* (SP). The latter is ideally suited for encapsulating sporadic and periodic real-time tasks, whereas the former is useful primarily if there is a need for fixed, known replenishment times.

In the following examples, we use sporadic polling reservations.

Creating a Reservation

Each reservation is identified by a **reservation ID (RID)**, which is simply a non-negative number (like a PID). However, there are two important differences between PIDs and RIDs:

1. In contrast to PIDs, RIDs are not automatically assigned. Rather, the desired RID must be specified when the reservation is created.
2. The same RID may be used on multiple processors (i.e., RIDs are not unique systemwide). Hence it is important to specify the processor both when creating reservations and when attaching processes to reservations.

Use the following invocation to create a new sporadic polling reservation with RID 123 on core 0:

```
resctl -n 123 -t polling-sporadic -c 0
```

Unless there is an error, no output is generated.

The above command created a polling reservation with a default budget of 10ms and a replenishment period of 100ms.

To specify other periods and/or budgets, use the `-p` and `-b` options, respectively.

For instance, use the following invocation to create a sporadic polling reservation with RID 1000 on core 1 with a budget of 25ms and a replenishment period of 50ms:

```
resctl -n 1000 -t polling-sporadic -c 1 -b 25 -p 50
```

By default, `resctl` creates EDF-scheduled polling reservations. To create fixed-priority reservations, simply set a numeric priority explicitly with the `-q` option. In LITMUS^{RT}, a lower numeric priority value corresponds to a higher priority, i.e., 1 is the highest priority in LITMUS^{RT} and 511 is (currently) the lowest possible priority. (The number of distinct priority levels available can be easily changed by recompiling the kernel.)

Deleting a Reservation

There is currently no mechanism to delete individual reservations. Once created, a reservation persists until the scheduler plugin is switched.

To delete *all* reservations, simply switch the active scheduler plugin to Linux and back again.

```
setsched Linux
setsched P-RES
```

Real-time Tasks

There are two ways of running real-time tasks in LITMUS^{RT}. The first is to use the `liblitmus` API to program a custom real-time application, and the second is to use the `rt_launch` tool for executing arbitrary Linux binaries as LITMUS^{RT} real-time tasks.

Common to both cases, there are a couple of parameters that should be set before starting a real-time task:

- **Worst-case execution time (WCET)** : The maximum execution time of any job of the task. In LITMUS^{RT}, scheduler plugins typically interpret this value as a *budget* (and not as a statement about the true maximum).
- **Period** : The minimum activation interval of the task. Scheduler plugins (typically) enforce this parameter.
- **Deadline** : The relative deadline of the task. If not specified, the period is used as the implicit deadline.
- **Offset** : The delay between a *task system release* (see the discussion of `release_ts` below) and the release

of the task's first job. This parameter defaults to zero and is rarely set.
- **Partition** : The partition (or cluster in the case of C-EDF or PFAIR) to which a task belongs. Unnecessary for global schedulers.

It is further possible to specify a task's "hardness" as its class; however, *this parameter is ignored* by all currently included schedulers and has no effect.

- **Class** : Tasks are classified into hard real-time (HRT), soft real-time (SRT) and best-effort (BE) in LITMUS^{RT} . If not specified otherwise, tasks default to being HRT tasks.

In the following, we provide a brief description of the tools in `liblitmus` related to running real-time tasks. For detailed usage information, run the mentioned commands without any parameters.

Simulating CPU-bound workloads with `rtspin`

The tool `rtspin` is a simple test task distributed with `liblitmus` that follows the first approach, i.e., it uses the API to programmatically "become" a real-time task when launched. The tool executes a simple, configurable spin loop that is useful for simulating CPU-bound workloads. It can be used for test and debugging purposes.

```
Usage:
    (1) rtspin [COMMON-OPTS] WCET PERIOD DURATION
    (2) rtspin [COMMON-OPTS] -f FILE [-o COLUMN] WCET PERIOD
    (3) rtspin -l
    (4) rtspin -B

Modes:
    (1) run as periodic task with given WCET and PERIOD
    (2) as (1), but load per-job execution times from CSV file
    (3) Run calibration loop (how accurately are target runtimes met?)
    (4) Run background, non-real-time cache-thrashing loop (w/ -m).

    -w                  wait for synchronous release
    -v                  verbose (prints PID)
    -p CPU              physical partition or cluster to assign to
    -r VCPU             reservation to attach to
    -d DEADLINE         relative deadline, implicit by default (in ms)
    -q PRIORITY         priority to use (ignored by EDF plugins, highest=1,
lowest=511)
    -c be|srt|hrt       task class (best-effort, soft real-time, hard real-time)
    -l                  run calibration loop
    -m RSS              set working set size (memory accessd by job)
    -e                  turn on budget enforcement,disabled by default
    wcet, period        reservation parameters (in ms)

    WCET and PERIOD are milliseconds, DURATION is seconds.
    CRITICAL SECTION LENGTH is in milliseconds.
    RESIDENT SET SIZE (RSS) is in number of pages
```

For example, the following command executes the task for 5 seconds on core 2 with a period of 10 ms and a WCET of 1.5 ms.

```
rtspin -p 2 1.5 10 5
```

For partitioned plugins (i.e., P-FP, PSN-EDF, P-RES), the partition number ranges from 0 to *m* - 1, where *m* is the number of CPUs.

For global schedulers (i.e., GSN-EDF, and PFAIR and C-EDF with the default cluster size "ALL"), the partition option (`-p`) is irrelevant and should be omitted.

Note that priority assignment is not carried out automatically under fixed-priority schedulers. To specify a task's priority, use the `-q` option, which accepts priorities in the range 1–511, where 1 is the highest-possible priority. If the `-q` option is omitted, `rtspin` defaults to the lowest-possible priority (511 in the current LITMUS^{RT} version).

When testing schedulers or learning to use LITMUS^{RT}, we recommend to run `rtspin` with the `-v` option, which provides some useful feedback on the execution of each job, including the task's PID and each job's absolute deadline.

Launching `rtspin` in a Reservation

The `rtspin` test application can be assigned to a pre-existing reservation with the `-r` option.

For example, to assign an `rtspin` process that runs for three seconds with period 100 and execution time 10 to reservation 1000 on core 1, launch `rtspin` like this:

```
 rtspin -p 1 -r 1000 10 100 3
```

Under reservation-based schedulers, task priorities are irrelevant (i.e., ignored). Instead, priorities need to be assigned to reservations, as mentioned in the section titled *Creating a reservation*.

Launching Arbitrary Binaries with `rt_launch`

Sometimes it is useful to run arbitrary tasks in LITMUS^{RT} real-time mode, either to run "legacy" code or for testing purposes. For this purpose, `liblitmus` provides a tool called `rt_launch` that allows executing arbitrary binaries as real-time tasks with arbitrary parameters. (In some sense, `rt_launch` can be thought of as the opposite of `nice`.)

The tool accepts mostly the same parameters as `rtspin`, but instead of a test duration, simply specify a binary that is to be executed. Any parameters not consumed by `rt_launch` will be passed as arguments to the to-be-launched binary.

```
Usage: rt_launch OPTIONS wcet period program [arg1 arg2 ...]
    -w                  wait for synchronous release
    -v                  verbose (prints PID)
    -p CPU              physical partition or cluster to assign to
    -r VCPU             virtual CPU or reservation to attach to (irrelevant to
most plugins)
    -R                  create sporadic reservation for task (with VCPU=PID)
    -d DEADLINE         relative deadline, implicit by default (in ms)
    -o OFFSET           offset (also known as phase), zero by default (in ms)
    -q PRIORITY         priority to use (ignored by EDF plugins, highest=1,
lowest=511)
    -c be|srt|hrt     task class (best-effort, soft real-time, hard real-time)
    -e                turn off budget enforcement (DANGEROUS: can result in
lockup)
    wcet, period      reservation parameters (in ms)
    program           path to the binary to be launched
```

For example, the below command launches the `find` utility on processor 1 as a "real-time task" with a period of 100ms and a budget of 10ms.

```
rt_launch -p 1 10 100 find /
```

By default, the launched binary will be subject to budget enforcement to prevent runaway processes from taking over the processor. For example, by launching `find` with a large period (say, 5 seconds), the effects of budget enforcement become easy to notice.

```
rt_launch -p 1 10 5000 find /
```

Note that it may be required to pass `--` (two dashes) after the last argument intended for `rt_launch` to stop it from processing options intended for the to-be-launched program. For example, suppose the goal is to print only files ending in `.c`. The following invocation will result in an error because `rt_launch` will try to interpret the option `-iname` and fail.

```
# Will fail; rt_launch does not understand -iname.
rt_launch -p 1 10 100 find / -iname '*.c'
```

Instead, use the following invocation to clearly separate options for `rt_launch` from those for `find`.

```
# Works as expected; rt_launch stops option processing at '--'.
rt_launch -p 1 10 100 -- find / -iname '*.c'
```

Budget enforcement can be turned off with the `-e` option, but this is dangerous: under a classic plugin, a CPU-bound task *will* monopolize the CPU and likely lock up the system. Under the newer reservation-based schedulers, the budget enforcement is provided by the reservations (and not on a per-task basis) and cannot be turned off.

Under fixed-priority scheduling, don't forget to set a priority with the `-q` option.

Launching Arbitrary Binaries in a Reservation

The `rtspin` options related to reservations (`-p` and `-r`) are also understood by the `rt_launch` utility. Similar to the prior reservation example, the following command executes `find` in reservation 1000 on core 1:

```
rt_launch -p 1 -r 1000 10 100 find /
```

Attaching an Already Running Task to a Reservation

It is also possible to assign an already running, non-real-time process or thread to a reservation. This can be accomplished with the `-a` (attach) option of `resctl`.

For example, to move the current shell into reservation 1000 on core 1, execute the following command:

```
resctl -a $$ -r 1000 -c 1
```

As a result, the shell will now be running as a real-time task, subject to the priority and budget replenishment of the given reservation.

Setting Up a (Synchronous) Task System Release with `release_ts`

In many cases, tasks should not commence running before *all* tasks have finished initialization. Furthermore, it can simplify a system's design and analysis if tasks are known to share a common "time zero".

This is especially true for periodic tasks—typical schedulability analysis for periodic tasks assumes either *synchronous* task systems, where all tasks release their first job at a common time zero, or *asynchronous* task systems, where each task releases its first job at a *known* offset (or *phase*) relative to a shared time zero.

For these reasons, LITMUS^{RT} has explicit support for setting up and triggering synchronous task system releases (which are basically a form of barrier synchronization).

The tools `rtspin` and `rt_launch` support this feature with the `-w` option, which means *wait for a (synchronous) task system release*. When set, tasks do not start executing immediately, but wait for a synchronous release point.

The synchronous task release can be triggered by running `release_ts`. The number of tasks waiting for release, along with the total number of active real-time tasks, can be listed with `cat /proc/litmus/stats`.

Example:

```
$ setsched GSN-EDF
# launch four tasks waiting for release
$ rtspin -w 10 100 4 &
$ rtspin -w 10 100 4 &
$ rtspin -w 10 100 4 &
$ rtspin -w 10 100 4 &
$ cat /proc/litmus/stats
real-time tasks   = 4
ready for release = 4
$ release_ts
Released 4 real-time tasks.
```

Note that it is possible to specify a number of tasks to wait for before triggering the synchronous release with the `-f` option of `release_ts`.

Overhead Tracing

We next discuss how to trace and process system overheads (such as context switch costs, scheduling costs, task wake-up latencies, etc.).

Recording Overheads with Feather-Trace

To record overheads, use the high-level wrapper script `ft-trace-overheads` in a system running a LITMUS^{RT} kernel that has been compiled with overhead tracing enabled in the kernel configuration (i.e., `CONFIG_SCHED_OVERHEAD_TRACE=y`).

Use the script as follows. First, activate the scheduler that you are interested in (e.g., `GSN-EDF`). Then simply run `ft-trace-overheads` (as root) with a given name to identify the experiment. While `ft-trace-overheads` is running, execute your benchmark to exercise the kernel. When the benchmark has completed, terminate `ft-trace-overheads` by pressing the enter key.

Example:

```
$ setsched GSN-EDF
$ ft-trace-overheads my-experiment
[II] Recording /dev/litmus/ft_cpu_trace0 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_cpu=0.bin
[II] Recording /dev/litmus/ft_cpu_trace1 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_cpu=1.bin
[II] Recording /dev/litmus/ft_cpu_trace2 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_cpu=2.bin
[II] Recording /dev/litmus/ft_cpu_trace3 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_cpu=3.bin
[II] Recording /dev/litmus/ft_msg_trace0 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_msg=0.bin
[II] Recording /dev/litmus/ft_msg_trace1 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_msg=1.bin
[II] Recording /dev/litmus/ft_msg_trace2 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_msg=2.bin
[II] Recording /dev/litmus/ft_msg_trace3 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=my-experiment_msg=3.bin
Press Enter to end tracing...
# [run your benchmark]
# [press Enter when done]
Ending Trace...
Disabling 4 events.
Disabling 4 events.
Disabling 4 events.
Disabling 4 events.
Disabling 18 events.
Disabling 18 events.
Disabling 18 events.
Disabling 18 events.
/dev/litmus/ft_msg_trace3: XXX bytes read.
/dev/litmus/ft_msg_trace0: XXX bytes read.
/dev/litmus/ft_msg_trace1: XXX bytes read.
/dev/litmus/ft_cpu_trace2: XXX bytes read.
/dev/litmus/ft_cpu_trace1: XXX bytes read.
/dev/litmus/ft_cpu_trace3: XXX bytes read.
/dev/litmus/ft_cpu_trace0: XXX bytes read.
/dev/litmus/ft_msg_trace2: XXX bytes read.
```

For performance reasons, Feather-Trace records overhead data into separate per-processor trace buffers, and treats core-local events and inter-processor interrupts (IPIs) differently. Correspondingly, `ft-trace-overheads` records two trace files for each core in the system.

1. The file `overheads…cpu=$CPU.bin` contains all overhead samples related to CPU-local events such as context switches.
2. The file `overheads…msg=$CPU.bin` contains overhead samples stemming from IPIs such as reschedule notifications related to the processor `$CPU`.

Key=Value Encoding

To aid with keeping track of relevant setup information, the tool encodes the system's host name and the currently active scheduler plugin in a simple `key=value` format in the filename.

We recommend to adopt the same encoding scheme in the experiment tags. For example, when running an experiment named "foo" with (say) 40 tasks and a total utilization of 75 percent, we recommend to launch `ft-trace-overheads` as `ft-trace-overheads foo_n=40_u=75`, as the additional parameters will be added transparently to the final trace file name.

Example:

```
ft-trace-overheads foo_n=40_u=75
[II] Recording /dev/litmus/ft_cpu_trace0 -> overheads_host=rts5_scheduler=GSN-
EDF_trace=foo_n=40_u=75_cpu=0.bin
…
```

However, this convention is purely optional.

Automating `ft-trace-overheads`

It can be useful to terminate `ft-trace-overheads` from another script by sending a signal. For this purpose, provide the `-s` flag to `ft-trace-overheads`, which will make it terminate cleanly when it receives the `SIGUSR1` signal.

When recording overheads on a large platform, it can take a few seconds until all tracer processes have finished initialization. To ensure that all overheads are being recorded, the benchmark workload should not be executed until initialization is complete. To this end, it is guaranteed that the string "to end tracing…" does not appear in the script's output (on STDOUT) until initialization is complete on all cores.

What does Feather-Trace data look like?

Feather-Trace produces “raw” overhead files. Each file contains simple event records. Each event record consists of the following fields (→ see definition):

- an event ID (e.g., `SCHED_START`),
- the ID of the processor on which the event was recorded (0–255),
- a per-processor sequence number (32 bits),
- the PID of the affected or involved process (if applicable, zero otherwise),
- the type of the affected or involved process (best-effort, real-time, or unknown),
- a timestamp (48 bits),
- a flag that indicates whether any interrupts occurred since the last recorded event, and
- an approximate interrupt count (0–31, may overflow).

The timestamp is typically a raw cycle count (e.g., obtained with `rdtsc`). However, for certain events such as `RELEASE_LATENCY`, the kernel records the time value directly in nanoseconds.

Note: Feather-Trace records data in native endianness. When processing data files on a machine with a different endianness, endianness swapping is required prior to further processing (see `ftsort` below).

Event Pairs

Most Feather-Trace events come as pairs. For example, context-switch overheads are measured by first recording a `CXS_START` event prior to the context switch, and then a `CXS_END` event just after the context switch. The context-switch overhead is given by the difference of the two timestamps.

There are two event pairs related to scheduling: the pair `SCHED_START-SCHED_END` records the scheduling overhead prior to a context switch, and the pair `SCHED2_START-SCHE2D_END` records the scheduling overhead after a context switch (i.e, any clean-up).

To see which event records are available, simply record a trace with `ft-trace-overheads` and look through it with `ftdump` (see below), or have a look at the list of event IDs (→ see definitions).

Low-Level Tools

The feather-trace-tools repository provides three main low-level tools that operate on raw overhead trace files. These tools provide the basis for the higher-level tools discussed below.

1. `ftdump` prints a human-readable version of a trace file’s contents. This is useful primarily for manual inspection. Run as `ftdump <MY-TRACE-FILE>`.
2. `ftsort` sorts a Feather-Trace binary trace file by the recorded sequence numbers, which is useful to normalize traces prior to further processing in case events were stored out of order. Run as `ftsort <MY-TRACE-FILE>`. `ftsort` can also carry-out endianness swaps if needed. Run `ftsort -h` to see the available options.
3. `ft2csv` is used to extract overhead data from raw trace files. For example, to extract all context-switch overhead samples, run `ft2csv CXS <MY-TRACE-FILE>`. Run `ft2csv -h` to see the available options.

By default, `ft2csv` produces CSV data. It can also produce binary output compatible with NumPy's `float32` format, which allows for efficient processing of overhead data with NumPy's `numpy.memmap()` facility.

High-Level Tools

The feather-trace-tools repository provides a couple of scripts around `ftsort` and `ft2csv` that automate common post-processing steps. We recommend that novice users stick to these high-level scripts until they have acquired some familiarity with the LITMUS^{RT} tracing infrastructure.

Post-processing of (a large collection of) overhead files typically involves:

1. sorting all files with `ftsort`,
2. splitting out all recorded overhead samples from all trace files,
3. combining data from per-cpu trace files and from traces with different task counts, system utilizations, etc. into merged data files for further processing,
4. counting how many events of each type were recorded,
5. shuffling and truncating all sample files, and finally
6. extracting simple statistics such as the observed median, mean, and maximum values.

Note that step 4 is required to allow a statistically meaningful comparison of the sampled maximum overheads. (That is, to avoid sampling bias, do not compare the maxima of trace files containing a different number of samples.)

Corresponding to the above steps, the feather-trace-tools repository provides a number of scripts that automate these tasks.

Sorting Feather-Trace Files

The `ft-sort-traces` script simply runs `ftsort` on all trace files. Invoke as `ft-sort-traces <MY-TRACE-FILES>`. We recommended to keep a log of all post-processing steps with `tee`.

Example:

```
ft-sort-traces overheads_*.bin 2>&1 | tee -a overhead-processing.log
```

Sorting used to be an essential step, but in recent versions of LITMUS^{RT}, most traces do not contain any out-of-order samples.

Extracting Overhead Samples

The script `ft-extract-samples` extracts all samples from all provided files with `ft2csv`.

Example:

```
ft-extract-samples overheads_*.bin 2>&1 | tee -a overhead-processing.log
```

The underlying `ft2csv` tool automatically discards any samples that were disturbed by interrupts.

Combining Samples

The script `ft-combine-samples` combines several data files into a single data file for further processing. This script assumes that file names follow the specific key=value naming convention already mentioned above:

```
<basename>_key1=value1_key2=value2...keyN=valueN.float32
```

The script simply strips certain key=value pairs to concatenate files that have matching values for all parameters that were not stripped. For instance, to combine all trace data irrespective of task count, as specified by "*n=*", invoke as `ft-combine-samples -n <MY-DATA-FILES>`. The option `--std` combines files with different task counts (`_n=`), different utilizations (`_u=`), for all sequence numbers (`_seq=`), and for all CPU IDs (`_cpu=` and `_msg=`).

Example:

```
ft-combine-samples --std overheads_*.float32 2>&1 | tee -a overhead-
processing.log
```

Counting Samples

The script `ft-count-samples` simply looks at all provided trace files and, for each overhead type, determines the minimum number of samples recorded. The output is formatted as a CSV file.

Example:

```
ft-count-samples  combined-overheads_*.float32 > counts.csv
```

Random Sample Selection

To allow for an unbiased comparison of the sample maxima, it is important to use the same number of samples for all compared traces. For example, to compare scheduling overhead under different schedulers, make sure you use the same number of samples for all schedulers. If the traces contain a different number of samples (which is very likely), then a subset must be selected prior to computing any statistics.

The approach chosen here is to randomly shuffle and then truncate (a copy of) the files containing the samples. This is automated by the script `ft-select-samples`.

Note: the first argument to `ft-select-samples` *must* be a CSV file produced by `ft-count-samples`.

Example:

```
ft-select-samples counts.csv combined-overheads_*.float32 2>&1 | tee -a
overhead-processing.log
```

The script does not modify the original sample files. Instead, it produces new files of uniform size containing the randomly selected samples. These files are given the extension `sf32` (= shuffled float32).

Computing Simple Statistics

The script `ft-compute-stats` processes `sf32` or `float32` files to extract the maximum, average, median, and minimum observed overheads, as well as the standard deviation and variance. The output is provided in CSV format for further processing (e.g., formatting with a spreadsheet application).

Note: Feather-Trace records most overheads in cycles. To convert to microseconds, one must provide the speed of the experimental platform, measured in the number of processor cycles per microsecond, with the `--cycles-per-usec` option. The speed can be inferred from the processor's spec sheet (e.g., a 2Ghz processor executes 2000 cycles per microsecond) or from `/proc/cpuinfo` (on x86 platforms). The LITMUS^{RT} user-space library liblitmus also contains a tool `cycles` that can help measure this value.

Example:

```
ft-compute-stats combined-overheads_*.sf32 > stats.csv
```

Complete Example

Suppose all overhead files collected with `ft-trace-overheads` are located in the directory `$DIR`. Overhead statistics can be extracted as follows.

```
# (1) Sort
ft-sort-traces overheads_*.bin 2>&1 | tee -a overhead-processing.log

# (2) Split
ft-extract-samples overheads_*.bin 2>&1 | tee -a overhead-processing.log

# (3) Combine
ft-combine-samples --std overheads_*.float32 2>&1 | tee -a overhead-
processing.log

# (4) Count available samples
ft-count-samples  combined-overheads_*.float32 > counts.csv

# (5) Shuffle & truncate
ft-select-samples counts.csv combined-overheads_*.float32 2>&1 | tee -a
overhead-processing.log

# (6) Compute statistics
ft-compute-stats combined-overheads_*.sf32 > stats.csv
```

Tracing a Schedule

Whereas Feather-Trace data records *how long* a scheduling decision or context switch takes, the `sched_trace` interface instead records *which* tasks are scheduled at what point and corresponding job releases and deadlines.

Recording a Schedule with `sched_trace`

The main high-level tool for recording scheduling decisions is the script `st-trace-schedule`.

To record the execution of a task system, follow the following rough outline:

1. start recording all scheduling decisions with `st-trace-schedule`;
2. launch and initialize all real-time tasks such that they wait for a *synchronous task system release* (see the `release_ts` utility in `liblitmus`);
3. release the task set with `release_ts`; and finally
4. stop `st-trace-schedule` when the benchmark has completed.

Example:

```
st-trace-schedule my-trace
CPU 0: 17102 > schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=0.bin
[0]
CPU 1: 17104 > schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=1.bin
[0]
CPU 2: 17106 > schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=2.bin
[0]
CPU 3: 17108 > schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=3.bin
[0]
Press Enter to end tracing...
# [launch tasks]
# [kick off experiment with release_ts]
# [press Enter when done]
Ending Trace...
Disabling 10 events.
Disabling 10 events.
Disabling 10 events.
Disabling 10 events.
/dev/litmus/sched_trace2: XXX bytes read.
/dev/litmus/sched_trace3: XXX bytes read.
/dev/litmus/sched_trace1: XXX bytes read.
/dev/litmus/sched_trace0: XXX bytes read.
```

As the output suggests, `st-trace-schedule` records one trace file per processor.

What does `sched_trace` data look like?

A scheduling event is recorded whenever

- a task is dispatched (switched to),
- a task is preempted (switched away),
- a task suspends (i.e., blocks), or
- a task resumes (i.e., wakes up).

Furthermore, the release time, deadline, and completion time of each job are recorded, as are each task's parameters and the name of its executable (i.e., the `comm` field in the Linux kernel's PCB `struct task_struct`). Finally, the time of a synchronous task system release (if any) is recorded as a reference of "time zero".

The tool `st-dump` may be used to print traces in a human-readable format for debugging purposes.

Drawing a Traced Schedule

The pycairo-based `st-draw` tool renders a trace as a PDF. By default, it will render the first one thousand milliseconds after *time zero*, which is either the first synchronous task system release (if any) or the time of the first event in the trace (otherwise).

Example:

```
st-draw schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=*.bin
# Will render the schedule as schedule_host=rts5_scheduler=GSN-EDF_trace=my-
trace.pdf
```

Invoke `st-draw -h` for a list of possible options. If the tool takes a long time to complete, run it in verbose mode (`-v`) and try to render a shorter schedule (`-l`).

Obtaining Job Statistics

The tool `st-job-stats` produces a CSV file with relevant per-job statistics for further processing with (for example) a spreadsheet application.

Example:

```
st-job-stats schedule_host=rts5_scheduler=GSN-EDF_trace=my-trace_cpu=*.bin
# Task,   Job,     Period,   Response, DL Miss?,   Lateness,  Tardiness,
Forced?,       ACET
# task NAME=rtspin PID=17406 COST=590000 PERIOD=113000000 CPU=254
 17406,     3,  113000000,   17128309,        0,  -95871691,          0,
0,     388179
 17406,     4,  113000000,   12138793,        0, -100861207,          0,
0,     382776
 17406,     5,  113000000,    7137743,        0, -105862257,          0,
0,     382334
 17406,     6,  113000000,    2236774,        0, -110763226,          0,
0,     382352
 17406,     7,  113000000,     561701,        0, -112438299,          0,
0,     559208
```

```
 17406,     8,  113000000,     384752,         0, -112615248,          0,
0,     382539
 17406,     9,  113000000,     565317,         0, -112434683,          0,
0,     561602
 17406,    10,  113000000,     379963,         0, -112620037,          0,
0,     377526
[...]
```

There is one row for each job. The columns record:

1. the task's PID,
2. the job sequence number,
3. the task's period,
4. the job's response time,
5. a flag indicating whether the deadline was missed,
6. the job's lateness (i.e., response time minus relative deadline),
7. the job's tardiness (i.e., max(0, lateness)),
8. a flag indicating whether the LITMUS^{RT} budget enforcement mechanism inserted an artificial job completion, and finally
9. the actual execution time (ACET) of the job.

Note that the *Forced?* flag is always zero for proper reservation-based schedulers (e.g., under `P-RES`). Forced job completion is an artefact of the LITMUS^{RT} legacy budget enforcement mechanism under process-based schedulers (such as `GSN-EDF` or `P-FP`).

Writing Real-Time Tasks From Scratch

The tools used so far — primarily, `rtspin` and `rt_launch` — are useful for illustrative purposes and to exercise scheduler plugins. However, to actually deploy real real-time workloads on LITMUS^{RT}, closer integration with the system is required. To this end, we discuss in the following how to write real-time tasks that use the LITMUS^{RT} API "from scratch."

Note that the following is intended as a brief tutorial to provide a starting point. Giving a comprehensive survey of the entire LITMUS^{RT} API and user-space environment is beyond the scope of this guide.

We begin with the most basic case, a simple periodic task.

A Minimal Periodic Task

First, we require a couple of standard system headers for I/O and related tasks.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

The main header to include for functionality specific to LITMUS^{RT} is `litmus.h`, which is provided by `liblitmus`.

```
#include <litmus.h>
```

We now define the task parameters. Note that this may be determined dynamically at runtime (e.g., via passed parameters, as done by `rtspin` and `rt_launch`). The main thing to note is that the LITMUS^{RT} API expects times to be specified in nanoseconds. The `ms2ns()` helper function provided by liblitmus comes in handy in specifying these parameters more legibly in millisecond granularity.

```
#define PERIOD      ms2ns(1000)
#define DEADLINE    ms2ns(1000)
#define EXEC_COST   ms2ns(50)
```

Before going further, it is worth emphasizing that virtually all LITMUS^{RT} APIs can fail at runtime. It is therefore good practice — and strongly recommended — to carefully check all return values. To this end, we define a simple macro for ensuring that syscalls do not fail.

```
#define CALL( exp ) do { \
        int ret; \
        ret = exp; \
        if (ret != 0) \
            fprintf(stderr, "%s failed: %m\n", #exp); \
        else \
            fprintf(stderr, "%s ok.\n", #exp); \
    } while(0)
```

The periodic task that we implement is very simple. It increments a counter and then signals exit after ten jobs have been processed.

```
int i = 0;

int job(void)
{
    i++;
    if (i >= 10)
        return 1;
    return 0;
}
```

Next, we define the main program and loop that defines the period task. The variable `params` of type `struct rt_task` will hold all information related to this task relevant to the kernel.

```
int main()
{
    int do_exit;
    struct rt_task params;
```

First, we call `init_litmus()`, which initializes the data structures within `liblitmus` and is mandatory in order to correctly use `liblitmus`.

```
    CALL(init_litmus());
```

We now set up the parameters for this task by filling in the `struct rt_task`.

```
    init_rt_task_param(&params);
    params.exec_cost = EXEC_COST;
    params.period = PERIOD;
    params.relative_deadline = DEADLINE;
```

Note that we first call `init_rt_task_param()` to populate the struct with default values. It is recommended to always first call `init_rt_task_param()` because the layout of `struct rt_task` changes from version to version (e.g., sometimes newer LITMUS^{RT} kernels require new fields to be added). Using `init_rt_task_param()` ensures that all fields have sensible defaults values and thus aids with portability.

After the `struct rt_task` has been populated, we invoke the system call `set_rt_task_param()` with the thread ID of the current process to communicate the desired configuration to the kernel.

```
    CALL(set_rt_task_param(gettid(), &params));
```

In LITMUS^{RT}, processes start out as background processes and need to be “transitioned” into real-time mode. This is done via the `task_mode()` function in `liblitmus`. Note that `set_rt_task_param()` simply communicates the configuration to the kernel; it does not yet transition the task into real-time mode, which must be accomplished explicitly with the `task_mode()` API.

```
    CALL(task_mode(LITMUS_RT_TASK));
```

At this point, the process is running as a LITMUS^{RT} real-time task.

However, we do not yet want the task to commence the actual real-time computation, for the following reason. In a real system, there is most likely more than one task, and various tasks may depend on one another in complex ways. To ensure that the system is in a consistent state and ready for execution, the real-time task should wait at this point until all tasks of the task system have finished initialization. For this purpose, LITMUS^{RT} provides an API for signaling (synchronous) task system releases, as already mentioned before in the context of `rtspin` and `rt_launch` (recall the `-w` flag).

To await the release of the entire task system, which signals that the initialization phase has finished, we simply call the LITMUS^{RT} API `wait_for_ts_release()`.

```
CALL(wait_for_ts_release());
```

This call returns only after the user has run `release_ts` (or some other process that calls the `release_ts()` API).

What follows is the main loop of the task. We expect the job function to return whether the loop should exit. (In our simple example, this is signaled by returning 1 when the counter reaches 10.) The key is the `sleep_next_period()` function, which ensures that the job function is invoked only once per period.

```
    // Do real-time stuff
    do {
        sleep_next_period();
        do_exit = job();
        printf("%d\n", do_exit);
    } while(!do_exit);
```

In a real application, `do_exit` could (for example) be set by a signal handler that is invoked when the system is shutting down the real-time task.

Once we are done, we simply transition back to non-real-time mode using again the `task_mode()` function and then exit from `main()`.

```
    CALL(task_mode(BACKGROUND_TASK));

    return 0;
}
```

This basic skeleton achieves periodic real-time execution under global schedulers.

Partitioned Schedulers

Under partitioned or clustered scheduling, the task needs to carry out two additional steps as part of its initialization. Let `PARTITION` denote the ID of the partition/cluster on which the task should be executed.

1. Migrate to the correct partition/cluster with `CALL(be_migrate_to_domain(PARTITION))` prior to transitioning into real-time mode.
2. Fill in the field `cpu` of `struct rt_task` as follows:
   `param.cpu = domain_to_first_cpu(PARTITION);`

The call to `be_migrate_to_domain()` ensures that the process is running on (one of) the processor(s) that form(s) the partition/cluster after the call returns. The `be_` prefix indicates that the function is available only to non-real-time tasks, i.e., it must be called before transitioning the process into real-time mode.

The call to `domain_to_first_cpu()` translates a logical domain or cluster ID into a physical core number. This translation layer is useful because, on some systems, physical core IDs are not stable across reboots, so it is more convenient to say “this task belongs to the 3rd cluster as defined by shared L2 caches” rather than identifying which physical cores actually share an L2 cache.

A Minimal Event-Driven Task

From the point of view of the LITMUS^{RT} API, the structure and initialization of an event-driven task is identical to that of a periodic task. The only major difference to a periodic task is simply the main loop of the task.

In the interest of brevity, we do not report the common parts and focus on the main loop instead in the following.

Event-Driven Main Loop

For simplicity, we use `STDIN` as our “input event channel” (i.e., as the file descriptor from which we receive new events). In a real application, this could (for example) be a device file (e.g., `/dev/my-sensor`) or a UDP, raw Ethernet, or CAN socket.

The main difference to the periodic task example above is that we do *not* call `sleep_next_period()`.

```
 ...
 // Do real-time stuff
 do {
    do_exit = job();
 } while(!do_exit);
 ...
```

Instead, the task simply blocks on the file descriptor from which it receives input events (`STDIN` in this case). To this end, we call `read()` on the standard input file descriptor at the beginning of each job.

```
int job(void) {
    char buffer[80];
    CALL(read(STDIN_FILENO, buffer, sizeof(buffer)));
    buffer[79] = 0;
```

When an event is triggered, the `read()` unblocks (i.e., the process is resumed) and the “event” is made available to the job, which prints the message unless it receives the word ‘exit’.

```
    if (strcmp(buffer, "exit") == 0)
        return 1;
    else {
        printf("%s\n", buffer);
        return 0;
    }
```

```
}
```

Note that even though there is no explicit call to `sleep_next_period()`, the classic scheduler plugins still police budgets and enforce minimum inter-arrival times for new jobs (as specified by the parameter `period` in `struct rt_task`).

Reservation-based plugins such as P-RES also enforce budgets and inter-arrival times (or, really, minimum replenishment time separation), but do so based on the reservation parameters as configured with `resctl` (and ignore task parameters).

A Faux Event Source

For a quick way to demonstrate event-driven activation, we create a named FIFO using the `mkfifo` command and give that as the standard input to our event-driven task. As a result, it will block when trying to read from `STDIN`.

Suppose `example_event` is the binary of our event-driven real-time task. To manually trigger events, we work with two terminal windows from now on.

```
Terminal 1:
===========
mkfifo input
./example_event < input
```

Events can be triggered by redirecting `STDOUT` to the FIFO. However, due to the way named FIFOs work in Linux, we need a long-running process that keeps the FIFO open. Here we just call sleep for a long time and redirect its output to the named FIFO to keep it open.

```
Terminal 2:
===========
sleep 1000 > input &
# The preceding command is needed to make sure
# the FIFO is not closed by the echo processes invoked below.
[1] 7009
echo "hello" > input
echo "world" > input
echo "exit" > input
```

It should be apparent that the example task running in the first terminal is indeed triggered by the events generated in the second terminal. By giving the task running in the first terminal a large period and a small budget (e.g., several seconds and one millisecond, respectively), the effects of budget enforcement and minimum inter-arrival-time separation can be easily observed by manually triggering many events with short inter-arrival times.

Skeleton Real-Time Tasks: `base_task` and `base_mt_task`

To provide an easy starting point for the development of custom real-time tasks, `liblitmus` comes with two example skeletons of periodic tasks that are known to compile and work.

The file `bin/base_task.c` contains an example sequential, single-threaded real-time task, which we recommend as a basis for developing real-time tasks using the `liblitmus` API.

The template in the file `bin/base_mt_task.c` is slightly more involved and shows how to build a multithreaded real-time process, where each thread corresponds to a sporadic task.