LITMUS^{RT}: A Hands-On Primer

Manohar Vanga, Mahircan Gül, Björn Brandenburg

MPI-SWS, Kaiserslautern, Germany

Goals

- A whirlwind tour of LITMUS^{RT}
    - Working with scheduler plugins.
    - Running real-time tasks under global & partitioned schedulers
    - Synchronous release
    - Working with reservations
- Tracing and visualizing schedules
- Writing real-time tasks using `liblitmus`
- Overhead tracing with Feather-Trace

Goals

Slides for this tutorial available at

`http://litmus-rt.org/tutorial/tutorial-slides.pdf`

More extensive and detailed LITMUS^{RT} manual at

`http://litmus-rt.org/tutorial/manual.html`

Preliminaries

Before We Begin...

Setup your environment by following the instructions at `http://litmus-rt.org/tutorial/`

1. Install VirtualBox (`http://virtualbox.org`)
2. Download the LITMUS^{RT} playground image (`http://litmus-rt.org/tutorial/litmus-2016.1.qcow.tar.gz`)
3. Create a new VirtualBox VM using the LITMUS^{RT} image

Before We Begin...

This is a hands-on tutorial!

You should follow along by typing out commands highlighted in orange in a root shell.

```
$ echo "Hello World"
Hello World
```

Before We Begin...

Boot up into LITMUS^{RT} (default boot option in GRUB).

```
$ uname -a
Linux litmus 4.1.3+ #1 SMP Mon Apr 4 19:00:57
CEST 2016 x86_64 x86_64 x86_64 GNU/Linux
```

After booting up the VM,

1. Open up the terminal
2. Login as `root` (password: `litmus`)
3. Navigate to `/sandbox`

```
litmus@litmus:~$ sudo su
Password:
root@litmus:/home/litmus# cd /sandbox
root@litmus:/sandbox#
```

A Whirlwind Tour of LITMUS^{RT}

Available Schedulers

LITMUS^{RT} provides a whole bunch of schedulers out-of-the-box!

```
$ cat /proc/litmus/plugins/loaded
PFAIR
P-FP
P-RES
PSN-EDF
GSN-EDF
Linux
```

`showsched`: Display Current Scheduler

The current scheduler can be viewed using the `showsched` command.

```
$ showsched
Linux
```

The default scheduler after boot is the `Linux` scheduler (dummy LITMUS^{RT} scheduler that defers all scheduling decisions to Linux's CFS scheduler).

setsched: Set Scheduler

We can enable a new scheduler using the setsched command.

```
$ setsched GSN-EDF
$ showsched
GSN-EDF
```

After enabling a LITMUS^{RT} plugin, the CFS scheduler continues to co-exist (at a lower level in the Linux scheduler hierarchy) to run non-RT background workloads.

Real-Time Processes in LITMUS^{RT}

Bunch of ways to create real-time processes in LITMUS^{RT}

- **`rt_launch`**: utility to run an arbitrary process as a real-time process.
- **`rtspin`**: dummy spinning task for use in experiments.
- **`liblitmus`-based**: custom tasks can be written using the LITMUS^{RT} C API provided by `liblitmus`.

Up Next: `rt_launch` and `rtspin` under `GSN-EDF` and `P-FP`.

(Examples of using the `liblitmus` API at the end of the talk.)

rt_launch: Launching a Real-Time Process

rt_launch provides a simple way to run an arbitrary binary as a real-time process.

```
rt_launch WCET PERIOD -- PROGRAM ARGS
```

Hands-On Demo: run a real-time web server in one command!

```
$ rt_launch 50 100 -- /usr/sbin/lighttpd \
         -f /etc/lighttpd/lighttpd.conf
$ firefox 127.0.0.1
$ killall lighttpd
```

rt_launch: Built-In Help

Lots more functionality. See built-in help (-h)

```
$ rt_launch -h
```

- Specify a priority (highest=1, lowest=511)
- Assign a relative deadline
- Specify a phase
- Wait for synchronous release

And lots more!

rtspin: Dummy Spinning Task

rtspin provides a dummy, spinning task for testing purposes.

```
rtspin OPTIONS WCET PERIOD DURATION
```

Hands-On Demo: run a dummy task with 10ms WCET and 100ms period for 5 seconds.

```
$ rtspin -v 10 100 5
```

The -v option prints out per-job information:

```
rtspin/2082:2 @ 100.3752ms
  deadline: 120709165733ns (=120.71s)
  current time: 120.61s, slack: 99.59ms
  target ACET:  10.00ms (100.00% of WCET)
```

P-FP: Partitioned Fixed-Priority Scheduler

So far, we've been working with global scheduling (GSN-EDF).

We now look at some specifics of working with partitioned schedulers.

```
$ setsched P-FP
```

`rtspin` Usage with P-FP

Under P-FP: Need to additionally specify a partition (-p)

Valid range from 0 to $m - 1$ (where m is the no. of processors).

Example: run a dummy task with 10ms WCET and 100ms period for 5 seconds *on processor 1* at the lowest priority.

```
$ rtspin -v -p 1  10 100 5
```

`rtspin` Usage with P-FP

Under P-FP: Need to additionally specify a priority (-q)

Valid range from 1 (highest) to 511 (lowest).

Example: run a dummy task with 10ms WCET and 100ms period for 5 seconds on processor 1 *with priority 1 (highest)*.

```
$ rtspin -v -p 1 -q 1  10 100 5
```

Synchronous Release

Synchronous Release with `release_ts`

Often, we want to perform a *synchronous release*: releasing all tasks at once.

We can make `rtspin` wait for a synchronous release to occur before starting (-w option).

```
    $ rtspin -v -w  -p 1 10 100 5 &
```

Note: The trailing & starts the process in the background and is useful for scripting the creation of multiple waiting tasks.

Synchronous Release with `release_ts`

Hands-On Demo: create 2 `rtspin` tasks on one processor 1 that wait for synchronous release.

```
$ rtspin -v -w -p 1 -q 1 5 50 5 &
$ rtspin -v -w -p 1 -q 2 10 100 5 &
```

We can view information on waiting tasks via `/proc/litmus/stats`.

```
$ cat /proc/litmus/stats
real-time tasks   = 2
ready for release = 2
```

The `release_ts` command releases all waiting tasks.

```
$ release_ts
Released 2 real-time tasks.
```

Tracing and Visualizing Schedules

Scheduler Tracing: Overview

Two tracing mechanisms: Feather-Trace and `sched_trace`

Feather-Trace: Generic tracing framework used for measuring scheduler overheads.

`sched_trace`: records which tasks are scheduled at what point, and corresponding job releases and deadlines. Useful for acquiring job statistics and visualizing schedules.

Demo: Tracing and Visualizing Schedules

Hands-On Demo: Record and visualize a scheduling trace, as well as retrieve job-level information.

Create a new working directory for this demo:

```
$ mkdir /sandbox/st-demo
$ cd /sandbox/st-demo
```

Recording Traces

To record the execution of a task system:

1. Start recording scheduling decisions with `st-trace-schedule`
2. Launch and initialize real-time tasks and wait for a synchronous release
3. Release tasks (with `release_ts`)
4. Stop `st-trace-schedule` when the benchmark has completed.

Switch to GSN-EDF for next example:

```
$ setsched GSN-EDF
```

Recording Traces: Hands-On Demo

Start recording scheduling traces.

```
$ st-trace-schedule my-trace
```

```
CPU 0: 2950 > schedule_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0.bin [0]
CPU 1: 2952 > schedule_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1.bin [0]
Press Enter to end tracing...
```

Recording Traces: Hands-On Demo

Open up a new tab **as `root`** and create some waiting `rtspin` tasks.

```
$ rtspin -w 10 100 5 &
[1] 3003
$ rtspin -w 20 50 5 &
[2] 3004
$ rtspin -w 5 30 5 &
[3] 3005
$ rtspin -w 5 20 5 &
[4] 3006
```

Now release them with `release_ts` and `wait` for them to finish:

```
$ release_ts
Released 4 real-time tasks.
$ wait
```

Recording Traces: Hands-On Demo

Stop recording traces by pressing ENTER on `st-trace-schedule`

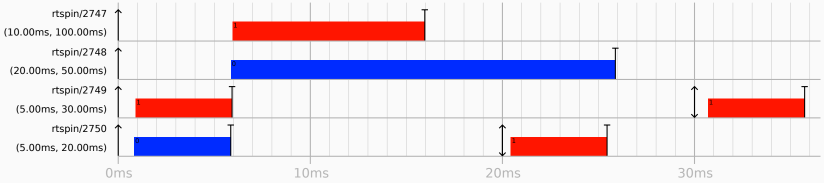
```
Ending Trace...
Disabling 10 events.
Disabling 10 events.
/dev/litmus/sched_trace1: 10584 bytes read.
/dev/litmus/sched_trace0: 10176 bytes read.
```

Visualizing Schedules with `st-draw`

`st-draw` allows to easily visualize schedules:

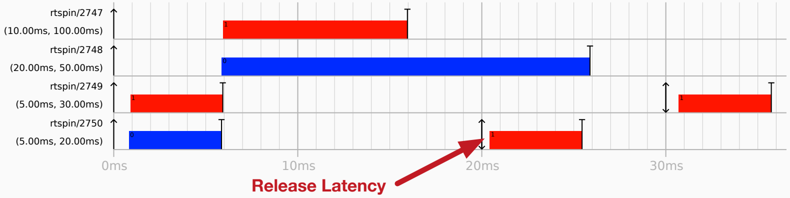
```
$ st-draw *.bin
$ evince *.pdf
```

Life saver when it comes to debugging! See `st-draw -h` for more command line options.



Release Latency in Virtual Machines

Caution: Timing within virtual machines is inaccurate due to the overhead of virtualization. This can result in large **release latency** (˜2ms in the example below).



Release latency is orders of magnitude lower on real hardware.

Job Statistics with `st-job-stats`

`st-job-stats` allows to easily obtain job statistics from a scheduling trace.

```
$ st-job-stats *my-trace*.bin | head
```

```
# Task, Job, Period, Response, DL Miss?, Lateness, Tardiness, Forced?, ACET
# task NAME=rtspin PID=3783 COST=10000000 PERIOD=100000000 CPU=0
3783, 2, 100000000,    21238, 0, -99978762, 0, 0,     2642
3783, 3, 100000000, 11318100, 0, -88681900, 0, 0, 10022417
3783, 4, 100000000, 20907624, 0, -79092376, 0, 0, 10009508
3783, 5, 100000000, 11308376, 0, -88691624, 0, 0, 10043864
3783, 6, 100000000, 20336977, 0, -79663023, 0, 0,  9999738
```

Lots of other useful data available:

- Response time of each job
- Flag specifying if the job missed a deadline
- Job lateness, tardiness
- Actual execution time of job

Working with Reservations

P-RES Plugin: Reservations in LITMUS^{RT}

P-RES is a reservation-based scheduling plugin in LITMUS^{RT}.

```
$ setsched P-RES
```

Supports a set of partitioned uniprocessor reservations of the following types:

- periodic polling server
- sporadic polling server
- table-driven reservations

P-RES support EDF, FP, as well as table-driven scheduling (time partitioning).

P-RES Plugin: Reservations in LITMUS^{RT}

Basic workflow for working with reservations:

1. Create a reservation on a specific core
2. Start a real-time task attached to a reservation

P-RES Plugin: Reservations in LITMUS^{RT}

Each reservation has a reservation ID (RID) in P-RES

- Must be explicitly assigned when creating reservations.
- Must be unique per core.

Important to specify the processor both when creating reservations and when attaching processes to reservations.

Creating a Reservation with `resctl`

The `resctl` command can be used to create reservations.

Hands-On Demo: Create a new sporadic polling reservation with RID 123 on core 1.

```
$ resctl -n 123 -t polling-sporadic -c 1
```

By default, budget is 10ms with a replenishment period of 100ms.

Hands-On Demo: Create a sporadic polling reservation with RID 234 on core 1 with a budget of 25ms and a replenishment period of 50ms

```
$ resctl -n 234 -t polling-sporadic -c 1 -b 25 -p 50
```

Creating a Reservation with `resctl`

The `resctl` command has many more options. See built-in help.

```
$ resctl -h
```

- Assigning priorities to reservations
- Specify a relative deadline
- Specify a phase

And more!

Assign `rtspin` Tasks to Reservations

Tasks are not assigned priorities directly. Instead priorities assigned to reservations (using the `-q` option with `resctl`)

Tasks are just attached to reservations at creation time.

Hands-On Demo: create `rtspin` task with 10ms WCET and 100ms period for 5 seconds on core 1, and attach it to previously created reservation (RID 234)

```
$ rtspin -v -p 1 -r 234  10 100 5
```

Overloading Reservations with a Large Budget

Setting a task budget higher than the available reservation budget results in job tardiness.

Hands-On Demo: create `rtspin` task with 30ms WCET and 50ms period for 5 seconds on core 1, and attach it to previously created reservation (RID 234, with budget 25ms and period 50ms).

```
$ rtspin -v -p 1 -r 234  30 50 5
```

Jobs are tardy as is indicated by the negative slack in the output:

```
...
rtspin/2908:86 @ 5006.4201ms
    deadline: 3312509325918ns (=3312.51s)
    current time: 3313.22s, slack: -706.44ms
    target ACET:  30.00ms (100.00% of WCET)
...
```

Overloading Reservations with Short Periods

Setting a task period lower than the reservation period results in job tardiness.

Hands-On Demo: create `rtspin` task with 25ms WCET and 40ms period for 5 seconds on core 1, and attach it to previously created reservation (RID 234, with budget 25ms and period 50ms).

```
$ rtspin -v -p 1 -r 234  25 40 5
```

Jobs are tardy as is indicated by the negative slack in the output:

```
...
rtspin/2909:104 @ 5006.4731ms
    deadline: 3573540213389ns (=3573.54s)
    current time: 3574.39s, slack: -846.54ms
    target ACET:  25.00ms (100.00% of WCET)
...
```

Table-Driven Reservations

Table-Driven Reservations

Under P-RES, reservations can be scheduled via a periodically-repeating, statically-defined scheduling table (a la **ARINC 653** time-partitioned scheduling).

The workflow remains the same as for the other reservation types:

1. Create one or more table-driven reservations using `resctl` (now additionally specified with a static schedule).
2. Attach one or more tasks to each table-driven reservation.

Specifying Static Schedules for Reservations

The static scheduler for a table-driven reservation is specified using two parameters:

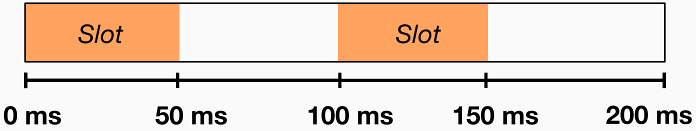
- **Major cycle (M):** Period of the scheduling table (*i.e.*, at runtime, the schedule repeats every M milliseconds).
- **Scheduling Slots:** A sequence of *non-overlapping* scheduling intervals relative to the start of the major cycle.

Specifying Static Schedules Using `resctl`

Example: Create a table-driven reservation on core 1 with ID 100 having a major cycle of 200ms and scheduled every alternate 50ms.

```
$ resctl -n 100 -c 1 -t table-driven  \
-m 200 '[0, 50)' '[100, 150)'
```

The above results in the following scheduling table:



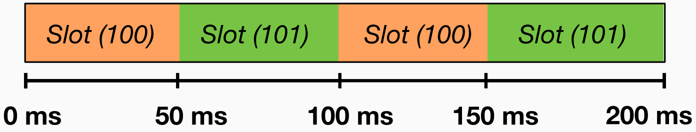
Note: `resctl` will throw an error if specified slots are not disjoint.

Specifying Static Schedules Using `resctl`

Example: Create two table-driven reservation on core 1 having a major cycle of 200ms and scheduled alternately every 50ms.

```
$ resctl -n 100  -c 1 -t table-driven \
-m 200 '[0, 50)' '[100, 150)'
$ resctl -n 101  -c 1 -t table-driven \
-m 200 '[50, 100)' '[150, 200)'
```

The above results in the following scheduling table:



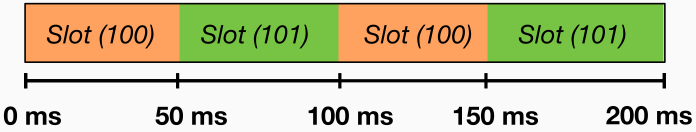
Caution: `resctl` will **not** throw an error if slots across *multiple reservations* overlap!

Specifying Static Schedules Using `resctl`

Caution: Reservations can be created with *different major cycles*, but care must be taken to ensure that slots do not overlap (up to the hyperperiod):

```
$ resctl -n 100 -c 1 -t table-driven \
-m 200  '[0, 50)' '[100, 150)'
$ resctl -n 101 -c 1 -t table-driven \
-m 100  '[50, 100)'
```

The above results in the same scheduling table as before but **takes up less space in memory**.



Attaching Tasks to Table-Driven Reservations

- **Multiple tasks** may be assigned to each table-driven reservation.
- When scheduled, a table-driven reservation selects the next process to be dispatched from its ready queue **via round-robin**.
- A table-driven reservation with no ready tasks **yields the processor to background tasks** when scheduled.

Creating Table-driven Reservations

Hands-On Demo: Create three table-driven reservations (on core 1) with major cycles of 200ms and non-overlapping slots:

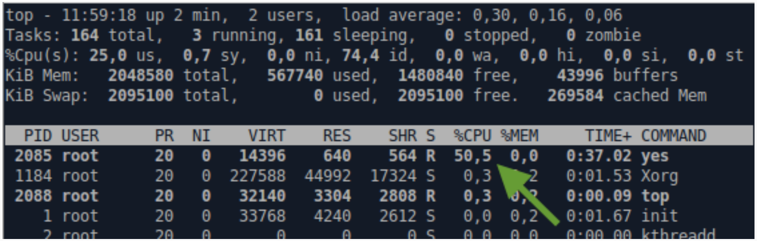
```
$ resctl -n 100 -c 1 -t table-driven  \
-m 200 '[0, 50)' '[100, 150)'

$ resctl -n 101 -c 1 -t table-driven  \
-m 200 '[50, 100)'

$ resctl -n 102 -c 1 -t table-driven  \
-m 200 '[150, 200)'
```

Creating Table-Driven Reservations

Hands-On Demo: Attach a process into our previously created reservation (ID 100) on core 1.

```
$ yes > /dev/null &
$ resctl -a `pidof yes` -r 100 -c 1
```

Running top on a new tab shows that the CPU usage of yes is capped at 50%.



```
top - 11:59:18 up 2 min,  2 users,  load average: 0,30, 0,16, 0,06
Tasks: 164 total,   3 running, 161 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25,0 us,  0,7 sy,  0,0 ni, 74,4 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   2048580 total,   567740 used,  1480840 free,    43996 buffers
KiB Swap:  2095100 total,        0 used,  2095100 free.   269584 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2085 root      20   0   14396    640    564 R  50,5  0,0   0:37.02 yes
 1184 root      20   0  227588  44992  17324 S   0,3  2,2   0:01.53 Xorg
 2088 root      20   0   32140   3304   2808 R   0,3  0,2   0:00.09 top
    1 root      20   0   33768   4240   2612 S   0,0  0,2   0:01.67 init
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.00 kthreadd
```

Coordinating Task Activations in Table-Driven Reservations

Coordinating Task Activations: Can ensure that a periodic task assigned to a table-driven reservation always wakes up at the beginning of each scheduling slot.

- The LITMUS^{RT} kernel's notion of time is `CLOCK_MONOTONIC`.
- Can use `clock_nanosleep()` to time wake-ups precisely to the start of time slots.

Deleting Reservations

Currently, there is no way to delete individual reservations.

Easy way to delete all reservations: switch plugin to Linux and all reservations are destroyed.

```
$ setsched Linux
# All reservations destroyed
```

Using `liblitmus`

liblitmus in Two Examples

liblitmus provides a C language API for interacting with LITMUS^{RT} in order to build custom real-time tasks.

We demonstrate its usage by explaining two simple examples:

- A periodic task (example_periodic.c)
- An event-driven task (example_event.c)

Available in the /opt/tutorial/ folder (along with a Makefile and README explaining how to use them).

Periodic Task with `liblitmus`

The `liblitmus` API is available via the `litmus.h` header.

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

Periodic Task with `liblitmus`

LITMUS$^{\text{RT}}$ calls may fail at runtime and error checking is highly recommended. We define a simple macro to help with this.

```
#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)
```

Periodic Task with `liblitmus`

Our periodic task simply increments a global counter to 10 before signaling an exit condition.

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

Periodic Task with `liblitmus`

In our `main()` function, the `param` variable 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;
```

Periodic Task with `liblitmus`

We must always start by calling `init_litmus()` in order to initialize `liblitmus` correctly.

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

Periodic Task with liblitmus

We now fill up the task parameters in the params variable.

```
#define PERIOD       ms2ns(1000)
#define DEADLINE     ms2ns(1000)
#define EXEC_COST    ms2ns(50)
...
init_rt_task_param(&params);
params.exec_cost = EXEC_COST;
params.period = PERIOD;
params.relative_deadline = DEADLINE;
```

Now we simply communicate these to the kernel:

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

Periodic Task with `liblitmus`

Processes begin as background processes in LITMUS^{RT}. We need to "transition" them to real-time tasks using the `task_mode()` function.

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

The process in now real-time. However, we might wish to wait for a synchronous release signal. This is achieved with the following line:

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

Periodic Task with `liblitmus`

We now write the main loop.

```
do {
    sleep_next_period();
    do_exit = job();
} while(!do_exit);
```

The key is the `sleep_next_period()` function call which ensures that the job function is invoked only once per period.

Our job function returns the exit condition for the loop (in our simple example, this is signaled by returning 1 when the counter reaches 10)

Periodic Task with `liblitmus`

Once the loop is complete, we transition back to background mode and exit.

```
    CALL(task_mode(LITMUS_RT_TASK));
    return 0;
  }
```

Event-Driven Task with `liblitmus`

Almost identical to the periodic example with some minor changes.

The main difference is that we do not call `sleep_next_period()` in the loop.

```
...
do {
  do_exit = job();
} while(!do_exit);
...
```

Event-Driven Task with `liblitmus`

Instead, the task simply blocks on the file descriptor from which it receives input events (STDIN in this case).

We do this by calling `read()` at the beginning of each job.

```
int job(void) {
  int ret;
  char buffer[80];

  ret = read(STDIN_FILENO, buffer, sizeof(buffer));
  buffer[ret] = '\0';

  /* Strip the trailing newline */
  if (buffer[ret - 1] == '\n')
    buffer[ret - 1] = '\0';
```

Event-Driven Task with `liblitmus`

When an event is triggered, read() unblocks 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;
  }
}
```

Linking Against `liblitmus`

Included in the folder is a minimal `Makefile` to link the above two examples against `liblitmus`.

We simply include `config.makefile` at the top of our `Makefile` to link against `liblitmus`. (The `LIBLITMUS` environment variable holds the path to `liblitmus`).

```
include ${LIBLITMUS}/inc/config.makefile
```

At the end of the `Makefile`, we simply include `depend.makefile` to allow dependency tracking.

```
include ${LIBLITMUS}/inc/depend.makefile
```

Tracing with Feather-Trace

Feather-Trace: Overview

Feather-Trace allows us to trace and process various system overheads.

Generic framework that allows adding arbitrary trace points in the kernel statically at compile time (not covered in this tutorial).

`ft_trace`: Overview

What traces are available under Feather-Trace?

- Scheduling overhead
- Post-scheduling overhead
- Context switch overhead
- Task release latency
- Synchronization overheads
- Re-schedule IPI overhead

Demo: Overhead Tracing

Hands-On Demo: Record scheduler traces and retrieve various overhead statistics.

Create a new working directory for this demo:

```
$ mkdir /sandbox/ft-demo
$ cd /sandbox/ft-demo
```

FeatherTrace: Tracing with `ft-trace-overheads`

1. Start recording traces.

```
$ ft-trace-overheads my-trace
```

```
[II] Recording /dev/litmus/ft_cpu_trace0 -> overheads_host=litmus_scheduler=GSN-EDF_trace=my-
[II] Recording /dev/litmus/ft_cpu_trace1 -> overheads_host=litmus_scheduler=GSN-EDF_trace=my-
[II] Recording /dev/litmus/ft_msg_trace0 -> overheads_host=litmus_scheduler=GSN-EDF_trace=my-
[II] Recording /dev/litmus/ft_msg_trace1 -> overheads_host=litmus_scheduler=GSN-EDF_trace=my-
Press Enter to end tracing...
```

FeatherTrace: Tracing with `ft-trace-overheads`

Launch and release some tasks:

```
$ rtspin -w 10 100 5 &
[1] 4063
$ rtspin -w 10 100 5 &
[2] 4064
$ rtspin -w 10 100 5 &
[3] 4065
$ rtspin -w 10 100 5 &
[4] 4066
$ release_ts
Released 4 real-time tasks.
```

FeatherTrace: Tracing with `ft-trace-overheads`

Stop recording: Press ENTER on `ft-trace-schedule`

```
Ending Trace...
Disabling 18 events.
Disabling 4 events.
Disabling 4 events.
Disabling 18 events.
/dev/litmus/ft_cpu_trace1: 359664 bytes read.
/dev/litmus/ft_msg_trace0: 400 bytes read.
/dev/litmus/ft_msg_trace1: 1248 bytes read.
/dev/litmus/ft_cpu_trace0: 500752 bytes read.
```

FeatherTrace: Tracing with `ft-trace-overheads`

The result is a bunch of binary-format files that basically contain a huge array of individual packed samples:

```
# ls *.bin
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0.bin
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1.bin
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_msg=0.bin
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_msg=1.bin
```

Post-Processing FeatherTrace Records

Post-processing in FeatherTrace is a bit more involved, but easy with the scripts available with LITMUS$^{\text{RT}}$.

- Sorting traces
- Extracting overhead samples from trace files
- Extracting simple statistics (*e.g.*, observed median, mean, and maximum values).

We will walk through these step-by-step.

Post-Processing: Sorting Traces

Sorts all records by sequence number (may be out-of-order).

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

* We recommend using `tee` as shown above to log all operations.

```
...
[2/4] Sorting overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1.bin
Total           : 22479
Holes           :     0
Reordered       :     0
Non-monotonic   :     0
Seq. constraint :     0
Implausible     :     0
Size            :  0.34 Mb
Time            :  0.00 s
Throughput      : 79.03 Mb/s
...
```

Post-Processing: Extract Samples

`ft-extract-samples` extracts all samples from overhead files.

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

Note: `ft-extract-samples` automatically discards samples that were disturbed by interrupts (these samples have a flag set).

Post-Processing: Extract Samples

Output: `NumPy`-compatible files (ending in `.float32`) containing an array of samples for each type of trace. Allows faster processing of data (compared to the CSV format) via memory mapping.

```
# ls *.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0_overhead=CXS.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0_overhead=RELEASE.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0_overhead=RELEASE-LATENCY.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0_overhead=SCHED.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=0_overhead=SCHED2.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_msg=0_overhead=SEND-RESCHED.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1_overhead=CXS.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1_overhead=RELEASE.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1_overhead=RELEASE-LATENCY.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1_overhead=SCHED.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_cpu=1_overhead=SCHED2.float32
overheads_host=litmus_scheduler=GSN-EDF_trace=my-trace_msg=1_overhead=SEND-RESCHED.float32
```

Post-Processing: Compute Statistics

Now we can simply compute statistics for each

```
$ ft-compute-stats overheads_*.float32 > stats.csv
```

```
# Plugin, #cores, Overhead,   Unit, #tasks, #samples,          max,  99.9th perc,...
 GSN-EDF,      *,      CXS, cycles,      *,      106,  10950.00000,  10895.92500,...
 GSN-EDF,      *,  RELEASE, cycles,      *,     1403, 274350.00000, 203814.84200,...
 GSN-EDF,      *,   SCHED2, cycles,      *,      108,    620.00000,    617.32500,...
 GSN-EDF,      *,    SCHED, cycles,      *,      108,  29892.00000,  29813.78300,...
 ...
```

Post-Processing: Comparing Results

To compare results from two or more experiments, we additionally need to do the following steps.

- Merging data files for further processing.
- Counting how many events of each type were recorded.
- Shuffling and truncating all sample files (un-biasing).

Post-Processing: Combine Traces From All Cores

`ft-combine-samples` combines files based on key-value naming convention.

The `--std` option combines files with different task counts, utilizations, for all sequence numbers and CPU IDs.

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

Post-Processing: Counting Samples, Un-biasing Data

For each overhead type, **ft-count-samples** determines the minimum number of samples recorded.

Counts used to un-bias data using **ft-select-samples**.

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

We now un-bias data by randomly selecting the same number of samples for all compared traces. (shuffle + truncate)

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

Where to go from here?

Further Resources

Project homepage

`https://litmus-rt.org`

Mailing list:

`https://wiki.litmus-rt.org/litmus/Mailinglist`

Design and implementation:

`http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf`

Manual:

`http://litmus-rt.org/tutorial/manual.html`