A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS$^{\mathrm{RT}}$

Felipe Cerqueira Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Scheduling latency under Linux and its principal real-time variant, the PREEMPT_RT patch, are typically measured using cyclictest*, a tracing tool that treats the kernel as a black box and directly reports scheduling latency. LITMUS*$^{\mathrm{RT}}$*, a real-time extension of Linux focused on algorithmic improvements, is typically evaluated using* Feather-Trace*, a fined-grained tracing mechanism that produces a comprehensive overhead profile suitable for overhead-aware schedulability analysis. This difference in tracing tools and output has to date prevented a direct comparison. This paper reports on a port of* cyclictest *to LITMUS*$^{\mathrm{RT}}$ *and a case study comparing scheduling latency on a 16-core Intel platform. The main conclusions are:* ***(i)*** *LITMUS*$^{\mathrm{RT}}$ *introduces only minor overhead itself, but* ***(ii)*** *it also inherits mainline Linux's severe limitations in the presence of I/O-bound background tasks.*

1 Introduction

Real-time tasks are usually activated in response to external events (*e.g.*, when a sensor triggers) or by periodic timer expirations (*e.g.*, once every millisecond). At the kernel level, both types of activations require the following sequence of steps to be carried out:

1. the processor transfers control to an interrupt handler to react to the device or timer interrupt;
2. the interrupt handler identifies the task waiting for the event and resumes it, which causes the task to be added to the ready queue;
3. the scheduler is invoked to check whether the resumed real-time task should be scheduled immediately (and, if so, on which processor); and finally,
4. if the resumed real-time task has higher priority than the currently executing task, then it is dispatched, which requires a context switch.

In theory, the highest-priority real-time task should be scheduled *immediately* when its activating event occurs, but in practice, Step 1 is delayed if interrupts are temporarily masked by critical sections in the kernel,[1] Steps 2–4 are delayed by cache misses, contention for memory bandwidth, and (in multiprocessor systems) lock contention, Step 3 is further delayed if preemptions are temporarily disabled by critical sections in the kernel,[2] and Step 4 generally causes a TLB flush (on platforms without a tagged TLB), which causes additional delays. Thus, even for the highest-priority task, there is always a delay between the activating event and the instant when the task starts executing. This delay, called *scheduling latency*, affects the response times of all tasks and imposes a lower bound on the deadlines that can be supported by the system. Therefore, it is essential to consider scheduling latency when determining whether a system can provide the desired temporal guarantees. The focus of this paper is an empirical evaluation of scheduling latency in Linux and two of its real-time variants, PREEMPT_RT and LITMUS$^{\mathrm{RT}}$, using *cyclictest*, a latency benchmark.

PREEMPT_RT and *cyclictest*. The importance of scheduling latency has made it a standard metric for the evaluation of real-time operating systems in general, and Linux and its real-time extensions in particular. Concerning the latter, the PREEMPT_RT patch—the *de facto* standard real-time variant of Linux—specifically aims to improve scheduling latency by reducing the number and the length of critical sections in the kernel that mask interrupts or disable preemptions [21, 27]. The efficacy of these changes is commonly quantified using *cyclictest*, a scheduling latency benchmark originally created by Thomas Gleixner and currently maintained by Clark Williams [2].

A key feature of *cyclictest* is that it is easy to use, and as a result it has been widely adopted as the universal benchmark of real-time performance under Linux. For instance, it has been applied to evaluate different hardware and software platforms (*e.g.*, [15, 25, 26, 29]) as well as various aspects of the kernel (*e.g.*, [16, 22, 24, 31]); the *cyclictest* approach has even been extended to continuously monitor the scheduling latency of real applications in a production environment [20]. In short, low scheduling latency as reported by *cyclictest* can be considered the gold standard of real-time performance in the Linux real-time community.

LITMUS$^{\mathrm{RT}}$. In contrast to the PREEMPT_RT project, which has been primarily driven by industry concerns, the **Li**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems [1, 10, 13, 14] is a primarily algorithms-oriented real-time extension of Linux. While PREEMPT_RT aggressively optimizes scheduling latency, LITMUS$^{\mathrm{RT}}$ facilitates the implementation and evaluation of novel scheduling and locking policies, which it does by introducing a scheduler

[1]In Linux, via the *local_irq_disable()* interface.

[2]In Linux, via the *preempt_disable()* interface.

plugin interface. That is, the PREEMPT_RT patch reengineers core parts of the kernel to avoid delaying Steps 1 and 3, whereas LITMUS$^{\mathrm{RT}}$ primarily modularizes the scheduling logic that is invoked in Step 3, and leaves other aspects of the kernel unchanged.

LITMUS$^{\mathrm{RT}}$ has served its purpose well and has enabled a number of studies exploring the tradeoff between system overheads and analytical temporal guarantees across a diverse range of scheduling approaches (*e.g.*, see [5, 7, 14, 19, 23]; a full list is provided online [1]). The key component of LITMUS$^{\mathrm{RT}}$ that enables such studies is Feather-Trace [11], a light-weight event tracing toolkit for x86 platforms that is used to collect fine-grained measurements of kernel overheads. With Feather-Trace, it is possible to extract detailed overhead profiles, which can then be incorporated into overhead-aware schedulability analysis to formally validate timing requirements under consideration of system overheads (this process is discussed in detail in [10, Ch. 4]).

The overhead characteristics of LITMUS$^{\mathrm{RT}}$'s scheduling policies are well documented and have been evaluated in detail in prior work [5, 10]. However, because *cyclictest* cannot be used (without modifications) to evaluate LITMUS$^{\mathrm{RT}}$ (see Sec. 2), and because Feather-Trace and *cyclictest* produce fundamentally different outputs (see Sec. 4), to date it has unfortunately not been possible to directly compare LITMUS$^{\mathrm{RT}}$ with Linux and the PREEMPT_RT patch.

In this paper, we present a comprehensive experimental evaluation of scheduling latency under LITMUS$^{\mathrm{RT}}$ based on data obtained with a ported version of *cyclictest*. By comparing LITMUS$^{\mathrm{RT}}$ against stock Linux and PREEMPT_RT, we seek to address two questions concerning the real-time capabilities of the current LITMUS$^{\mathrm{RT}}$ implementation:

1. Does the LITMUS$^{\mathrm{RT}}$ scheduling framework introduce a significant overhead in terms of scheduling latency? And:
2. To what extent is LITMUS$^{\mathrm{RT}}$ affected by high scheduling latencies due to not (yet) incorporating the improvements of the PREEMPT_RT patch?

To answer the first question, we compared LITMUS$^{\mathrm{RT}}$ against the underlying Linux version; to answer the second question, we compared LITMUS$^{\mathrm{RT}}$ against the latest stable version of the PREEMPT_RT patch.

The rest of the paper is organized as follows. Sec. 2 reviews how *cyclictest* operates and describes a faithful port of *cyclictest* to LITMUS$^{\mathrm{RT}}$. In Sec. 3, we present our experimental setup and discuss our results. In Sec. 4, we compare *cyclictest* with Feather-Trace and remark on some of the advantages and limitations of the *cyclictest* benchmark. In Sec. 5, we conclude and mention future work directions.

2 Porting *cyclictest* to LITMUS$^{\mathrm{RT}}$

While *cyclictest* is a remarkably versatile utility, it cannot be applied to LITMUS$^{\mathrm{RT}}$ "out of the box", since LITMUS$^{\mathrm{RT}}$ provides its own, non-standard system call and userspace API, which must be used to configure a real-time task's scheduling parameters. In this section, we discuss the (few) changes that were required to port *cyclictest* to LITMUS$^{\mathrm{RT}}$.

Since the validity of the measurements depends on a correct and unbiased application of *cyclictest*, we begin with a review of *cyclictest* in Sec. 2.1 and explain how its scheduling parameters are configured under stock Linux and PREEMPT_RT in Sec. 2.2. In Sec. 2.3, we review LITMUS$^{\mathrm{RT}}$'s task model and show how *cyclictest* was mapped to it. Finally, Sec. 2.4 briefly discusses a simple, but unexpected problem with the resolution of Linux's one-shot timers that we encountered during the experiments.

2.1 An Overview of *cyclictest*

The execution of *cyclictest* can be divided into three phases. During the initialization phase, the program creates a configurable number of threads (according to the specified parameters), which are then admitted as real-time tasks. The processor affinity mask is also set, which enables migration to be restricted. After that, each thread starts a periodic (*i.e.*, cyclic) execution phase, during which *cyclictest* executes the main measurement loop. An iteration (*i.e.*, one test cycle) starts when the thread's associated one-shot timer expires.[3] Once the thread resumes, a sample of scheduling latency is recorded as the difference between the current time and the instant when the timer should have fired. The timer is then rearmed to start a new iteration and the thread suspends. After a configurable duration, the thread is demoted to best-effort status and exits, and the recorded scheduling latency samples are written to disk.

The cyclic phase, during which the measurements are collected, uses only standard userspace libraries (for example, POSIX APIs to set up timers, synchronize threads, *etc.*) and does not rely on scheduler-specific functionality. Since LITMUS$^{\mathrm{RT}}$ maintains compatibility with most userspace APIs, only the code pertaining to task admission and exit must be adapted, *i.e.*, it suffices to replace *sched_setscheduler()* system calls with LITMUS$^{\mathrm{RT}}$'s library functions for task admission. Importantly, *cyclictest*'s core measurement loop does not have to be changed, which helps to avoid the inadvertent introduction of any bias.

The required modifications, which amount to only 18 lines of new code, are discussed in Sec. 2.3 below. To illustrate how LITMUS$^{\mathrm{RT}}$'s interface differs from stock Linux's interface, we first review how *cyclictest* is configured as a real-time task in Linux (either with or without the PREEMPT_RT patch).

2.2 Setting Task Parameters under Linux

In accordance with the POSIX standard, Linux implements a fixed-priority real-time scheduler (with 99 distinct priority levels). Tasks are dispatched in order of decreasing priority,

[3] *cyclictest* supports a large number of options and can be configured to use different timer APIs. We focus herein on the tool's default behavior.

and ties in priority are broken according to one of two policies: under the SCHED_RR policy, tasks of equal priority alternate using a simple round-robin scheme, and, under the SCHED_FIFO policy, multiple tasks with the same priority are simply scheduled in FIFO order with respect to the time at which they were enqueued into the ready queue. By default, tasks are allowed to migrate among all processors in Linux, but it is possible to restrict migrations using processor affinity masks, and *cyclictest* optionally does so.

Fig. 1 summarizes the relevant code from *cyclictest* that is used to admit a thread as a SCHED_FIFO real-time task under Linux. First, the thread defines the CPU affinity mask, which is assigned with *pthread_setaffinity_np()* (lines 6–13 in Fig. 1). To attain real-time priority, the thread calls the *sched_setscheduler()* system call with the desired scheduling policy and priority as parameters (lines 17-20). Finally, after the measurement phase, the thread transitions back to non-real-time status by reverting to the SCHED_OTHER best-effort policy (lines 24-25).

In the experiments discussed in Sec. 3, *cyclictest* was configured to spawn one thread per core and to use the SCHED_FIFO policy with the maximum priority of 99. Further, processor affinity masks were assigned to fix each measurement thread to a dedicated core. Since there is only one task per core, the choice of tie-breaking policy is irrelevant.

Executing the initialization sequence depicted in Fig. 1 under LITMUS$^{\mathrm{RT}}$ would *not* result in an error (LITMUS$^{\mathrm{RT}}$ does not disable SCHED_FIFO); however, it would also not achieve the desired effect because, for historical reasons, real-time tasks must use a different API (which also allows specifying more explicit and detailed parameters) to attain real-time status in LITMUS$^{\mathrm{RT}}$. We thus adapted *cyclictest* to use LITMUS$^{\mathrm{RT}}$'s API to create an analogous setup.

2.3 Setting Task Parameters under LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ implements the *sporadic task model* [28], in which real-time tasks are modeled as a sequence of recurrent jobs and defined by a tuple $T_i = (e_i, d_i, p_i)$, where e_i denotes the *worst-case execution time* (WCET) of a single job, d_i the *relative deadline*, and p_i the *minimum inter-arrival time* (or *period*). Under LITMUS$^{\mathrm{RT}}$'s event-driven scheduling policies, the parameter e_i is optional and used only for budget enforcement (if enabled). The parameter d_i, however, is required for scheduling plugins based on the *earliest-deadline first* (EDF) policy, and the parameter p_i is always required to correctly identify individual jobs. In LITMUS$^{\mathrm{RT}}$, all task parameters are expressed in nanosecond granularity since this is the granularity internally used by the kernel.

As mentioned in Sec. 2.1, each thread in *cyclictest* executes in a loop, alternating between resuming, measuring latency, and suspending. The wake-up timer is armed periodically, according to a configurable interval I (in microseconds) defined as a parameter.[4] *cyclictest*'s periodic pattern of execution exactly matches the assumptions underlying the sporadic task model and can thus be trivially expressed with parameters $d_i = p_i = I$. To avoid having to estimate the per-job (*i.e.*, per-iteration) execution cost of *cyclictest*, we set e_i to a dummy value of 1 ns and disable budget enforcement, so that each thread can always execute without being throttled by LITMUS$^{\mathrm{RT}}$.

```
 1  struct thread_param *par;
 2
 3  /* par contains the cyclictest configuration
 4   * and thread parameters */
 5
 6  if (par->cpu != -1) {
 7      CPU_ZERO(&mask);
 8      CPU_SET(par->cpu, &mask);
 9      thread = pthread_self();
10      if(pthread_setaffinity_np(thread,
11              sizeof(mask), &mask) == -1)
12          warn("Could not set CPU affinity");
13  }
14
15  /* ... */
16
17  memset(&schedp, 0, sizeof(schedp));
18  schedp.sched_priority = par->prio;
19  if (setscheduler(0, par->policy, &schedp))
20      fatal("Failed to set priority\n");
21
22  /* measurement phase */
23
24  schedp.sched_priority = 0;
25  sched_setscheduler(0, SCHED_OTHER, &schedp);
```

Figure 1: Task admission in Linux (original *cyclictest*).

The code necessary to realize admission of a *cyclictest* thread as a real-time task under LITMUS$^{\mathrm{RT}}$ is shown in Fig. 2. The first step is defining the task parameters (in particular, e_i, d_i, and p_i) in lines 3–8 and initializing the userspace interface with *init_rt_thread()*. Budget enforcement is disabled (line 7) and the maximum possible priority is assigned (line 8). LITMUS$^{\mathrm{RT}}$'s fixed-priority plugin currently supports 512 distinct priorities; the *priority* field is ignored by EDF-based plugins.

Next, if a *partitioned* scheduler is used, a scheduling approach where each task is statically assigned to a core, the task must specify its assigned processor in the *rt_task* structure (line 13) and perform this migration (line 14), which is accomplished by calling *be_migrate_to()*.[5] Otherwise, if a *global* scheduler is used, a scheduling approach where tasks may migrate freely, a processor assignment is not required (and ignored by the kernel if provided). The task parame-

[4]In fact, *cyclictest* allows two parameters: i, the timer interval of the first thread, and d, an increment which is added to the interval of each consecutive thread. For example, if $i = 1000$ and $d = 100$, *cyclictest* launches threads with intervals $I \in \{1000, 1100, 1200, \ldots\}$ μs. For simplicity, we assume a single interval I. By default, and as employed in our experiments, *cyclictest* uses $i = 1000\mu s$ and $d = 500\mu s$.

[5]The function *be_migrate_to()* is currently implemented as a wrapper around Linux's processor affinity mask API, but could be extended to incorporate LITMUS$^{\mathrm{RT}}$-specific functionality in the future. The "be_" prefix stems from the fact that it may be called only by best-effort tasks.

```
1   struct rt_task rtt; /* LITMUS^RT API */
2
3   init_rt_task_param(&rtt);
4   rtt.exec_cost = 1;
5   rtt.period = par->interval * 1000;
6   rtt.relative_deadline = par->interval * 1000;
7   rtt.budget_policy = NO_ENFORCEMENT;
8   rtt.priority = LITMUS_HIGHEST_PRIORITY;
9
10  init_rt_thread();
11
12  if (par->cpu != -1) {
13      rtt.cpu = par->cpu;
14      if (be_migrate_to(par->cpu) < 0)
15          fatal("Could not set CPU affinity");
16  }
17
18  if (set_rt_task_param(gettid(), &rtt) < 0)
19      fatal("Failed to set rt_param.");
20
21  if (task_mode(LITMUS_RT_TASK) != 0)
22      fatal("failed to change task mode.\n");
23
24  /* measurement phase */
25
26  task_mode(BACKGROUND_TASK);
```

Figure 2: Task admission in LITMUS$^{\mathrm{RT}}$ (modified *cyclictest*).

ters are stored in the process control block (and validated by the kernel) with the system call *set_rt_task_param()* in line 18. Finally, *task_mode(LITMUS_RT_TASK)* is called in line 21, which causes the thread to be admitted to the set of real-time tasks. After the measurement phase, *task_mode(BACKGROUND_TASK)* is used to give up real-time privileges and return to SCHED_OTHER status.

With these changes in place, *cyclictest* is provisioned under LITMUS$^{\mathrm{RT}}$ equivalently to the configuration executed under Linux (both with and without the PREEMPT_RT patch). This ensures a fair and unbiased comparison.

2.4 The Effect of Timer Resolution on *nanosleep()*

Despite our efforts to establish a level playing field, we initially observed unexpectedly large scheduling latencies under LITMUS$^{\mathrm{RT}}$ in comparison with SCHED_FIFO, even in an otherwise idle system. This was eventually tracked down to a systematic 50μs delay of timer interrupts, which was caused by the fact that Linux subjects *nanosleep()* system calls to timer coalescing to reduce the frequency of wake-ups. As this feature is undesirable for real-time tasks, it is circumvented for SCHED_FIFO and SCHED_RR tasks. A similar exception was introduced for LITMUS$^{\mathrm{RT}}$, which resolved the discrepancy in expected and observed latencies.

It should be noted that LITMUS$^{\mathrm{RT}}$ provides its own API for periodic job activations, and that this API has never been subject to timer coalescing, as it does not use the *nanosleep* functionality. The issue arose in our experiments only because we chose to not modify the way in which *cyclictest* triggers its periodic activations (since we did not want to change the actual measuring code in any way).

3 Experiments

We conducted experiments with *cyclictest* to evaluate the scheduling latency experienced by real-time tasks under LITMUS$^{\mathrm{RT}}$ in comparison with an unmodified Linux kernel. The results were further compared with latencies as observed under Linux with the PREEMPT_RT patch. Our testing environment consisted of a 16-core Intel Xeon X7550 2.0GHz platform with 1 TiB RAM. Features that lead to unpredictability such as hardware multithreading, frequency scaling, and deep sleep states were disabled for all kernels, along with every kernel configuration option associated with tracing or debugging. Background services such as *cron* were disabled to the extent possible, with the notable exception of the remote login server *sshd* for obvious reasons.

We used *cyclictest* to sample scheduling latency under six different kernel and scheduling policy combinations. Under LITMUS$^{\mathrm{RT}}$, which is currently still based on Linux 3.0, we focused our analysis on a subset of the event-driven scheduler plugins: the *partitioned EDF* plugin (PSN-EDF), the *global EDF* plugin (GSN-EDF), and the *partitioned fixed-priority* plugin (P-FP).[6] We did not evaluate LITMUS$^{\mathrm{RT}}$'s Pfair [4] plugin, which implements the PD^2 [3] scheduling policy, since PD^2 is a quantum-driven policy and hence not optimized to achieve low scheduling latency.[7]

We further evaluated SCHED_FIFO in three Linux kernels: in Linux 3.0 (the stock version, without any patches), Linux 3.8.13 (again, without patches), and Linux 3.8.13 with the PREEMPT_RT patch. Though we compare scheduling latencies of two different underlying versions of Linux, both considered versions exhibit a similar latency profile (for which we provide supporting data in Sec. 3.5), so our comparison of LITMUS$^{\mathrm{RT}}$ and PREEMPT_RT is valid despite the difference in base kernel versions.

For each scheduling policy and kernel, we varied the set of background tasks to assess scheduling latency in three scenarios: **(i)** a system with no background workload, **(ii)** a system with a cache-intensive, CPU-bound background workload, and **(iii)** a system with an interrupt-intensive, I/O-bound background workload. Of these three scenarios, scenario (i) is clearly the best-case scenario, whereas scenario (iii) puts severe stress onto the system. Scenario (ii) matches the background workload that has been used in prior LITMUS$^{\mathrm{RT}}$ studies (*e.g.*, see [5, 6, 10, 12]).

cyclictest was executed with standard SMP parameters (one thread per processor), with periods in the range of $I \in \{1000\mu s, 1500\mu s, 2000\mu s, \ldots\}$ and the *-m* flag enabled, which locks memory pages with *mlockall()* to prevent page faults. The result of each execution is a histogram of observed scheduling latencies, where the x-axis represents the

[6] The "S" and "N" in the plugin names PSN-EDF and GSN-EDF refer to support for predictable suspensions and non-preemptive sections; see [8, 10]. These algorithmic details are irrelevant in the context of this paper.

[7] Under a quantum-driven scheduler, worst-case scheduling latencies cannot be lower than the quantum length, which in the current version of LITMUS$^{\mathrm{RT}}$ is tied to Linux's scheduler tick and thus is at least $1ms$.

measured delay and the y-axis the absolute frequency of the corresponding value plotted on a log scale. Samples were grouped in buckets of size 1 μs. Each test ran for 20 minutes, generating around 5.85 million samples per configuration.

The outcome of the experiments is depicted in Figs. 3–7 and analyzed in the following sections. In Secs. 3.1–3.3, we first discuss the differences and similarities in scheduling latency incurred under LITMUS$^{\mathrm{RT}}$'s P-FP plugin, Linux 3.0, and Linux 3.8.13 (both with and without the PREEMPT_RT patch), and then in Sec. 3.4 we compare the results of the three considered LITMUS$^{\mathrm{RT}}$ scheduler plugins with each other. Finally, Sec. 3.5 compares Linux 3.0 and Linux 3.8.13.

3.1 Idle System

As a first step, we evaluated the latency of the system without background tasks under P-FP and PREEMPT_RT, both running on Linux 3.0, and Linux 3.8.13 with and without the PREEMPT_RT patch. An idle system represents a best-case scenario as non-timer-related interrupts are rare, because kernel code and data is likely to remain cached between scheduler activations, and since code segments within the kernel that disable interrupts have only few inputs to process.

As can be seen in Fig. 3, the maximum observed scheduling latency was below $20\mu s$ under each of the four schedulers (insets (a)-(d)), and even below $12\mu s$ under the PREEMPT_RT configuration. The maximum observed scheduling latency under stock Linux 3.8.13 is somewhat higher than under both LITMUS$^{\mathrm{RT}}$ and stock Linux 3.0. As high-latency samples occur only rarely, we ascribe this difference to random chance; with a longer sampling duration, latencies of this magnitude would likely be detected under LITMUS$^{\mathrm{RT}}$ and Linux 3.0, too. All three Linux variants exhibited comparable average and median latencies, close to $2.8\mu s$ and $2.6\mu s$, respectively. Scheduling latencies under LITMUS$^{\mathrm{RT}}$'s P-FP scheduler were slightly higher with a median and average of roughly $3.4\mu s$ and $3.1\mu s$, respectively.

Considering the overall shape of the histograms, all four schedulers exhibit similar trends. Slight differences are visible in PREEMPT_RT's histogram, which resembles the results for the other Linux versions in the initial 0-$5\mu s$ interval, but lacks higher latency samples. This suggests that PREEMPT_RT avoids outliers even in best-case scenarios.

Overall, as there are not many sources of latency in the absence of a background workload (such as the disabling of interrupts and contention at the hardware level), the observed scheduling latencies are suitably low under each tested kernel. While the LITMUS$^{\mathrm{RT}}$ patch does appear to increase latencies slightly on average, it does not substantially alter the underlying latency profile of Linux as other factors dominate. From this, we conclude that the LITMUS$^{\mathrm{RT}}$ framework does not inherently introduce undue complexity and overheads.

Next, we discuss the effects of increased processor and cache contention.

3.2 CPU-bound Background Workload

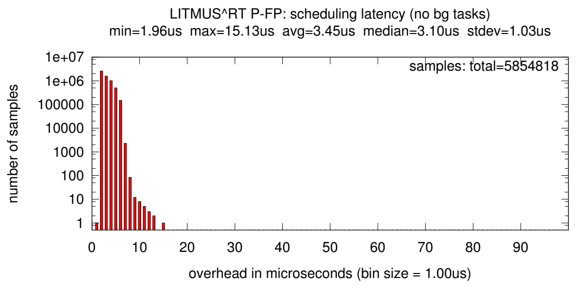
Kernel overheads in general, and thus also the scheduling latency experienced by real-time tasks, vary depending on the contention for limited hardware resources such as memory bandwidth and shared caches. A cache-intensive, CPU-bound background workload can thus be expected to result in worsened latencies, as cache misses and contention in the memory hierarchy are more likely to occur. To evaluate such a scenario, we executed *cyclictest* along with CPU-bound background tasks. For each processor, we instantiated a task consisting of a tight loop accessing random memory locations to generate cache misses and contention. Each such task was configured to have a working set of 20 MiB, to exceed the size of each processor's exclusive L2 cache, which has a capacity of 18 MiB. This causes significant cache pressure. Fig. 4 shows the recorded latencies for PSN-EDF, Linux 3.0, and Linux 3.8.13 (with and without PREEMPT_RT).

A pairwise comparison between the same policies in Fig. 3 and Fig. 4 illustrates how scheduling latencies are impacted by cache and memory issues. As expected, the average and maximum latencies under P-FP, Linux 3.0 and stock Linux 3.8.13, depicted in insets (a), (b), and (c), respectively, increased noticeably. While average and median scheduling latencies increased by only one to two microseconds in absolute terms, the increase in relative terms is significantly higher, exceeding 45 percent in the case of average latency under both LITMUS$^{\mathrm{RT}}$ and Linux 3.0. Most significant is the increase in observed maximum latency, which reached roughly $48\mu s$, $73\mu s$, and $65\mu s$ under LITMUS$^{\mathrm{RT}}$, Linux 3.0, and Linux 3.8.13, respectively. This shows that even a modest, compute-only background workload can significantly impact observable latencies in mainline Linux.

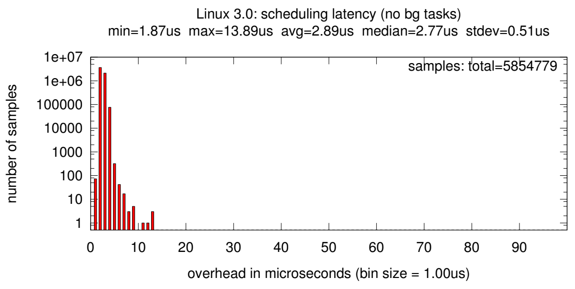
Interestingly, the advantages of PREEMPT_RT (inset (d)) become more apparent in this scenario: with the PREEMPT_RT patch, Linux was able to maintain low latencies despite the increased load, both in terms of average as well as maximum latency ($3.4\mu s$ and $17.42\mu s$, respectively). The corresponding stock kernel incurred significantly worse latencies (see the longer tail in Fig. 4(c)).

Comparing the distribution of samples, it can be seen that the observed scheduling latency under LITMUS$^{\mathrm{RT}}$'s P-FP plugin follows a slightly different pattern than either Linux 3.0 or Linux 3.8.13. In particular, LITMUS$^{\mathrm{RT}}$'s distribution appears to be "wider" and "heavier," with a less rapid decrease in the frequency of samples in the range of $1\mu s$–$40\mu s$. This explains LITMUS$^{\mathrm{RT}}$'s slightly higher average and median scheduling latencies, which are about $1\mu s$ higher than under either Linux 3.0 or Linux 3.8.13. However, note that LITMUS$^{\mathrm{RT}}$, Linux 3.0, and Linux 3.8.13 are all similarly subject to long tails, which indicates that the observed maximum latencies are caused by factors unrelated to LITMUS$^{\mathrm{RT}}$ (*i.e.*, they are caused by issues in the underlying Linux kernel, which the PREEMPT_RT patch addresses).

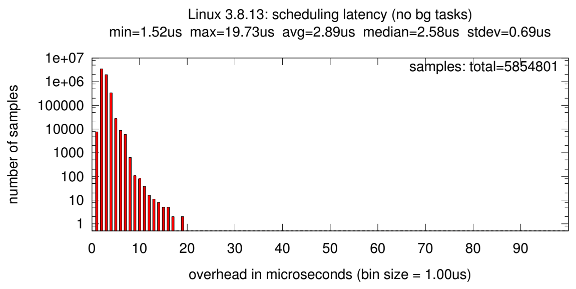
Nonetheless, the histograms reveal that, in the average case, LITMUS$^{\mathrm{RT}}$ adds measurable (but not excessive) ad-



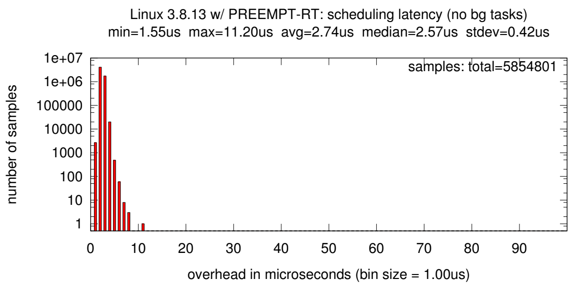
(a) LITMUS$^{\mathrm{RT}}$ with the P-FP scheduler plugin



(b) Linux 3.0



(c) Linux 3.8.13 without the PREEMPT_RT patch



(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 3: Histograms of observed scheduling latency in an otherwise idle system.

ditional overhead. We suspect two primary sources for this additional overhead. First, LITMUS$^{\mathrm{RT}}$'s scheduling path needs to acquire (at least) one additional spin lock compared to stock Linux, which is especially costly in the presence of high cache and memory-bandwidth contention. This additional spin lock acquisition stems from the fact that LITMUS$^{\mathrm{RT}}$'s scheduling state is not protected by Linux's runqueue locks; however, Linux's runqueue locks must still be acquired prior to invoking LITMUS$^{\mathrm{RT}}$'s scheduling framework. And second, the increased average-case overheads might be due to a lack of low-level optimizations in LITMUS$^{\mathrm{RT}}$ (in comparison with the mature codebase of Linux). Given that LITMUS$^{\mathrm{RT}}$ is primarily a research-oriented project focused on algorithmic real-time scheduling issues, a certain lack of low-level tuning is not surprising.

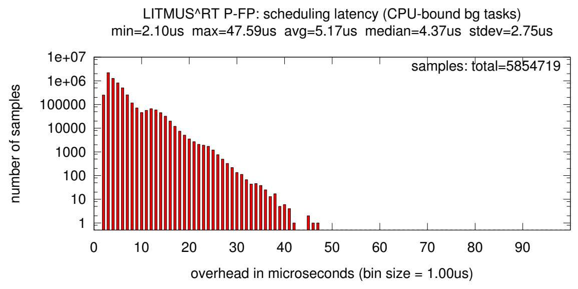
As was already briefly mentioned, the CPU-bound background workload matches the setup that has been used in prior LITMUS$^{\mathrm{RT}}$-based studies (*e.g.*, [5, 6, 10, 12]). As is apparent when comparing Fig. 3(a) with Fig. 4(a), our data confirms that the CPU-bound workload generates sufficient memory and cache pressure to magnify kernel overheads. Conversely, conducting overhead experiments *without* a cache-intensive background workload does not yield an accurate picture of kernel overheads. Next, we discuss the impact of interrupt-intensive background workloads.

3.3 I/O-bound Background Workload

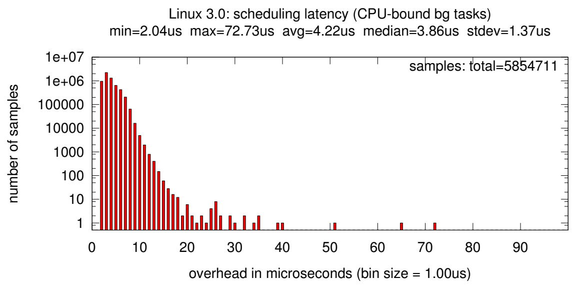
Interrupts are challenging from a latency point of view since interrupt handlers typically disable interrupts temporarily and may carry out significant processing, which both directly affects scheduling latency. It should be noted that Linux has long supported *split interrupt handling* (*e.g.*, see [9]), wherein interrupt handlers are split into a (short) *top half* and a (typically longer) *bottom half*, and only the top half is executed in the (hard) interrupt context, and the bottom half is queued for later processing. However, in stock Linux, bottom halves still effectively have "higher priority" than regular real-time tasks, in the sense that the execution of bottom halves is not under control of the regular SCHED_FIFO process scheduler[8] and thus may negatively affect scheduling latencies. Further, bottom halves may still disable interrupts and preemptions for prolonged times.

Considerable effort has been invested by the developers of the PREEMPT_RT patch to address these very issues. This is accomplished by forcing bottom half processing to take place in kernel threads (which can be scheduled such that they do not delay high-priority real-time tasks), and by identifying and breaking up code segments that disable interrupts and preemptions for prolonged durations. In contrast, since LITMUS$^{\mathrm{RT}}$ is currently based on stock Linux, and since the focus of LITMUS$^{\mathrm{RT}}$ is the exploration and evaluation of new scheduling policies (and *not* the reengineering of the underlying Linux kernel), no such improvements are present in LITMUS$^{\mathrm{RT}}$. A key motivation for our experiments was to determine to which extent LITMUS$^{\mathrm{RT}}$ is penalized by the

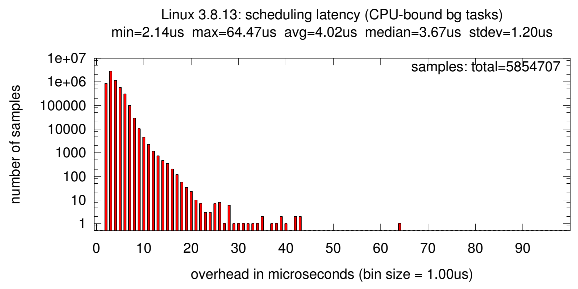
[8]Bottom halves are processed by so-called *softirqs*, which in stock Linux are invoked from interrupt and exception return paths.



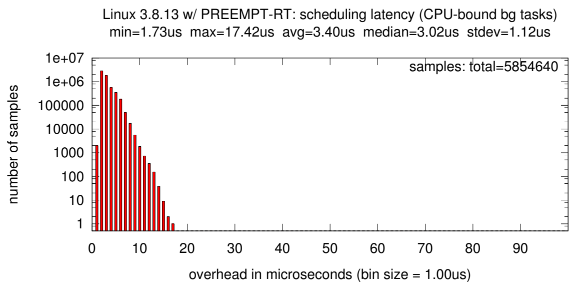
(a) LITMUS$^{\mathrm{RT}}$ with the P-FP scheduler plugin



(b) Linux 3.0



(c) Linux 3.8.13 without the PREEMPT_RT patch



(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 4: Histograms of observed scheduling latency in the presence of a CPU-bound background workload.

absence of such improvements.

We explored the impact of interrupt-intensive workloads on scheduling latency with I/O-bound background tasks that generate a large number of interrupts, system calls, and scheduler invocations. To simulate such workloads, we used a combination of the following three tools.

1. *hackbench*, a standard stress test for the Linux scheduler [30]. Under the employed default settings, it creates 400 processes that exchange tokens via (local) sockets, thus causing frequent system calls and scheduler invocations (due to blocking reads).

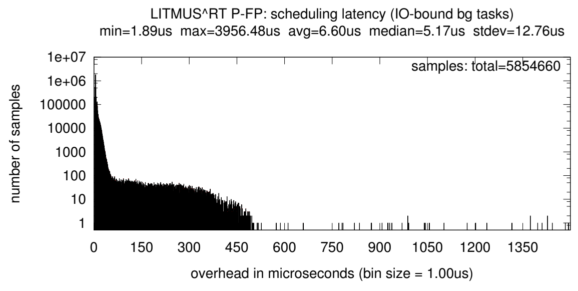
2. *Bonnie++*, a standard file system and hard disk stress test [17]. *Bonnie++* tests file creation, random and sequential file access, and file system metadata access. We configured *Bonnie++* to use *direct I/O*, which circumvents Linux's page cache (and thus results in increased disk activity). This workload results in a large number of system calls, disk interrupts, and scheduler invocations (due to blocking reads and writes).

3. *wget*, a common utility to download files via HTTP. We used *wget* to download a 600 MiB file from a web server on the local network in a loop. The downloaded file was immediately discarded by writing it to */dev/null* to avoid stalling on disk I/O. One such download-and-discard loop was launched for each of the 16 cores. This workload generates a large number of network interrupts as Ethernet packets are received at the maximum rate sustained by the network (with 1 Gib links in our lab).

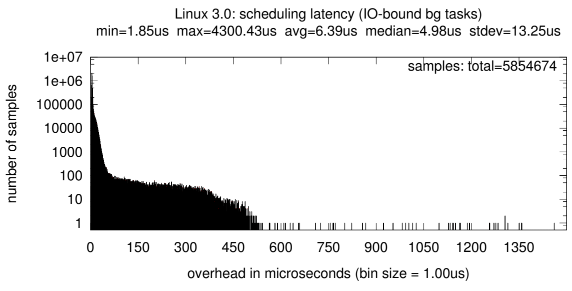
In combination, these three workloads cause considerable stress for the entire kernel, and can be expected to frequently trigger code paths that inherently have to disable interrupts and preemptions. While the three tools may not reflect any particular real-world application, the chosen combination of stress sources is useful to consider because it approaches a worst-case scenario with regard to background activity. The resulting distributions of observed scheduling latency under P-FP, Linux 3.0, and Linux 3.8.13 with and without the PREEMPT_RT patch are depicted in Fig. 5.

Scheduling latencies are severely affected by the I/O-bound background workload under LITMUS$^{\mathrm{RT}}$, Linux 3.0, and stock Linux 3.8.13 alike. The corresponding histograms, shown in insets (a)–(c) of Fig. 5, respectively, exhibit a long, dense tail. Note that the x-axis in Fig. 5 uses a different scale than Fig. 3 and 4: scheduling latencies in excess of $5ms$ were observed in this scenario, two orders of magnitude worse than in the previous ones. Scheduling latencies in this range clearly limit these kernels to hosting applications that are not particularly latency-sensitive.

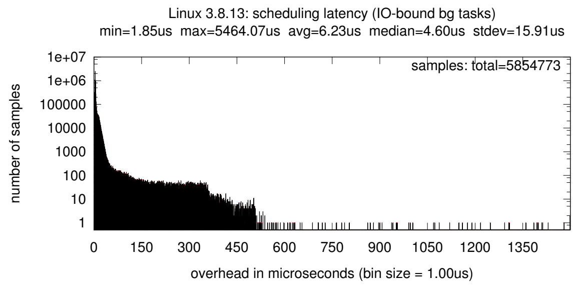
In contrast, Linux 3.8.13 with the PREEMPT_RT patch maintained much lower scheduling latencies, in the order of tens of microseconds, despite the stress placed upon the system, which can be seen in Fig. 5(d). Nonetheless, the maximum observed scheduling latency did increase to $44\mu s$, which shows that, even with the PREEMPT_RT patch, non-negligible latencies arise given harsh workloads. However, this maximum was still significantly lower than the maximum latency previously observed under Linux 3.8.13



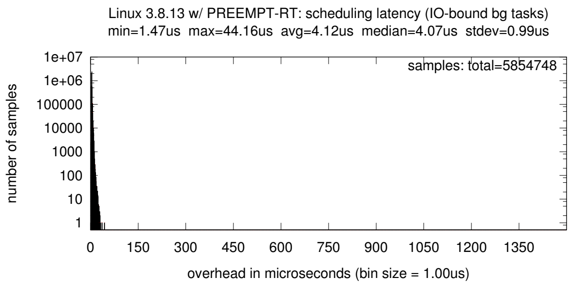
(a) LITMUS$^{\text{RT}}$ with the P-FP scheduler plugin



(b) Linux 3.0



(c) Linux 3.8.13 without the PREEMPT_RT patch



(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 5: Histograms of observed scheduling latency in the presence of an I/O-bound background workload.

without the PREEMPT_RT patch in the presence of only CPU-bound workloads, which is apparent when comparing Fig. 4(c) with Fig. 5(d). Remarkably, the average and median scheduling latency under PREEMPT_RT worsened by less than $0.7\mu s$ with the introduction of the I/O-bound workload.

Finally, we also ran two variations of the I/O-bound workload with varying degrees of disk activity. First, we disabled *bonnie++* altogether, which brought down the maximum observed latencies under Linux 3.0, Linux 3.8.13 (without the PREEMPT_RT patch), and LITMUS$^{\text{RT}}$ to around $550\mu s$, which is still too high for practical purposes, but shows that the extreme outliers are caused by disk-related code. And second, we tried launching an instance of *bonnie++* on each core, which brought the disk I/O subsystem to its knees and caused latency spikes in the range of 80–200 *milliseconds* (!) under the three non-PREEMPT_RT kernels. Remarkably, the maximum observed scheduling latency under PREEMPT_RT remained below $50\mu s$ even in this case.

Overall, our experiment asserts the importance of PREEMPT_RT in turning Linux into a viable real-time platform. Given the huge differences in maximum observed latency, LITMUS$^{\text{RT}}$ would be substantially improved if it incorporated PREEMPT_RT. Though this will require considerable engineering effort (both patches modify in part the same code regions), there are no fundamental obstacles to rebasing LITMUS$^{\text{RT}}$ on top of the PREEMPT_RT patch.

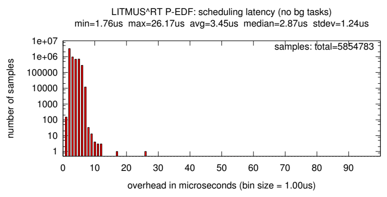
3.4 Scheduling Latency of LITMUS$^{\text{RT}}$ Plugins

In the preceding sections, we have focused on LITMUS$^{\text{RT}}$'s P-FP plugin, since it implements the same scheduling policy as SCHED_FIFO (albeit with a larger number of priorities and support for additional real-time locking protocols) and thus allows for the most direct comparison. We also investigated how scheduling latency varies among the three evaluated LITMUS$^{\text{RT}}$ scheduler plugins. Fig. 6 compares the P-FP, PSN-EDF and GSN-EDF plugins in LITMUS$^{\text{RT}}$, under each of the three considered background workloads.

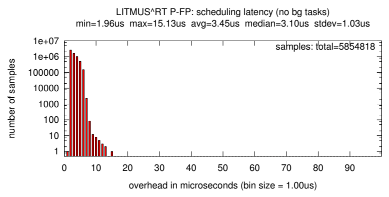
Comparing insets (g), (h), and (i), it is apparent that the three plugins are equally subject to high scheduling latencies (approaching $4ms$) in the case of the I/O-bound background workload. This is not surprising, since the long tail of high scheduling latencies is caused by the design of the underlying Linux kernel, and thus independent of the choice of plugin.

Further, comparing Fig. 6(a) with Fig. 6(b), and Fig. 6(d) with Fig. 6(e), it is apparent that the PSN-EDF and P-FP plugins yield near-identical scheduling latency distributions, despite the difference in implemented scheduling policy. This, however, is expected since the tests run only one real-time task per processor; the real-time scheduler is hence not stressed and the cost of the scheduling operation is so small compared to other sources of latency that any differences between fixed-priority and EDF scheduling disappear in the noise. Differences emerge only for higher task counts [10].

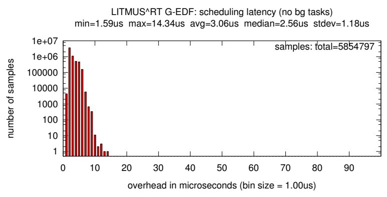
However, looking at Fig. 6(f) and Fig. 6(i), it is apparent that the scheduling latency is noticeably higher under GSN-



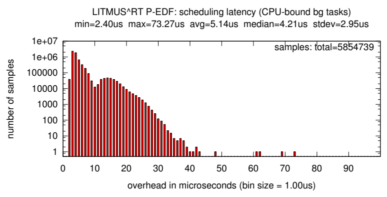
(a) PSN-EDF (idle)



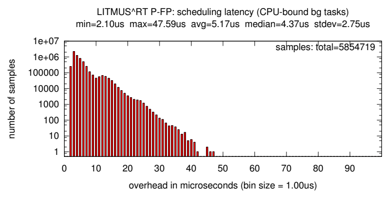
(b) P-FP (idle)



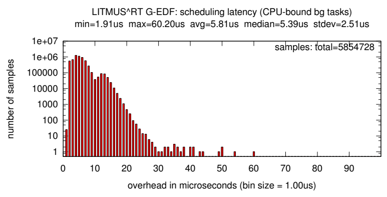
(c) GSN-EDF (idle)



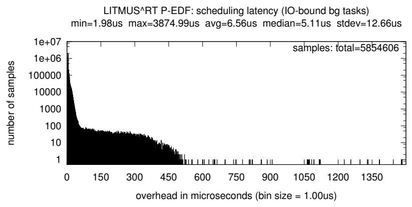
(d) PSN-EDF (CPU-bound)



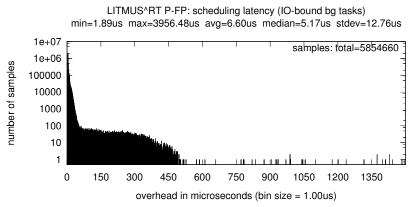
(e) P-FP (CPU-bound)



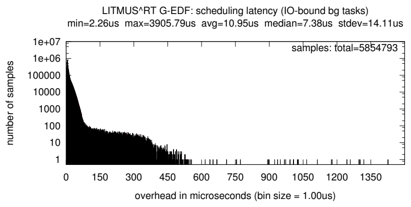
(f) GSN-EDF (CPU-bound)



(g) PSN-EDF (I/O-bound)



(h) P-FP (I/O-bound)



(i) GSN-EDF (I/O-bound)

Figure 6: Histograms of observed scheduling latency under LITMUS$^{\mathrm{RT}}$ with the PSN-EDF, P-FP, and GSN-EDF plugins, under each of the three considered background workloads.

EDF in the average case, which is due to its more complex implementation. Issues such as contention caused by coarse-grained locking, extra bookkeeping, and cache-coherence delays when accessing shared structures increase both the median and average observed scheduling latencies.

While this shows that LITMUS$^{\mathrm{RT}}$'s implementation of global scheduling incurs higher overheads, there is little reason to employ global scheduling when the number of tasks does not exceed the number of available cores (which is the case in the considered *cyclictest* setup). If the number of tasks actually exceeds the number of available cores—that is, if the scheduling problem is not entirely trivial—then other factors such as the impact of interference from higher-priority tasks or a need for bounded tardiness [18] can make minor differences in scheduling latency a secondary concern, with only little impact on overall temporal correctness.

3.5 Linux 3.0 vs. Linux 3.8

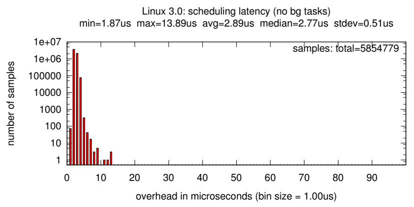
In this paper, we compared the latency of LITMUS$^{\mathrm{RT}}$ and Linux with the PREEMPT_RT patch using the latest versions of each patch, which are based on Linux 3.0 and Linux 3.8.13, respectively. As already discussed in the preceding sections, to verify that comparing the two patches is valid despite the difference in the underlying kernel version, we also measured the scheduling latencies exhibited by the two underlying (unpatched) Linux versions. For ease of comparison, the results are repeated in Fig. 7.

A comparison of inset (a)-(c) with insets (d)-(f) shows that, though the observed maxima vary (for example, from $13.89\mu s$ to $19.73\mu s$ in the scenario without background tasks), the shapes of the distributions are largely similar. Further, there are no substantial differences in the average and median latencies of the two kernel versions. This indicates that no significant improvements concerning latency and preemptivity have been incorporated since Linux 3.0. Therefore, a direct comparison between the LITMUS$^{\mathrm{RT}}$ patch and the PREEMPT_RT patch is valid.

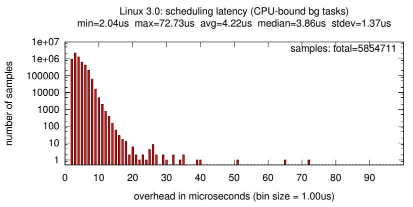
This concludes the discussion of our experimental results. Next, we briefly discuss how the presented *cyclictest* experiments differ from the overhead and latency tracing typically used to evaluate LITMUS$^{\mathrm{RT}}$.

4 Limitations of *cyclictest*

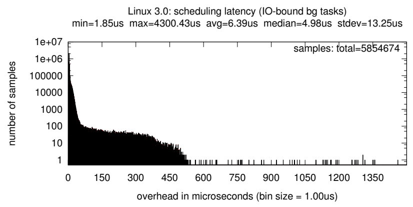
As discussed in Sec. 1, LITMUS$^{\mathrm{RT}}$ is normally evaluated using Feather-Trace, not *cyclictest*. While *cyclictest* is a very useful tool to assess and compare different kernel versions (*e.g.*, it can be used to test whether a proposed patch has a negative impact on scheduling latency), it also has some limitations if used as the sole metric for estimating a system's



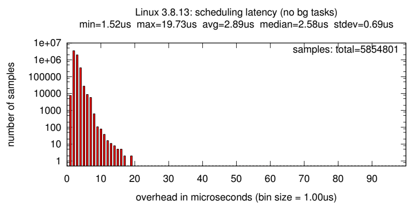
(a) Linux 3.0 (idle)



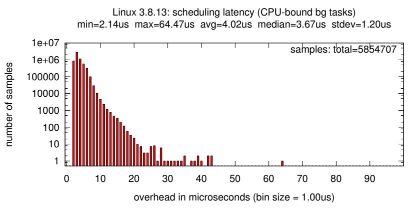
(b) Linux 3.0 (CPU-bound)



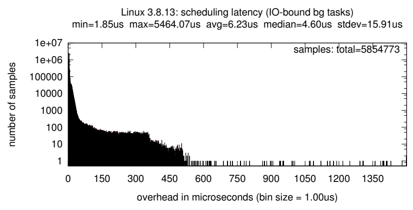
(c) Linux 3.0 (I/O-bound)



(d) Linux 3.8.13 (idle)



(e) Linux 3.8.13 (CPU-bound)



(f) Linux 3.8.13 (I/O-bound)

Figure 7: Histograms of observed scheduling latency under Linux 3.0 and 3.8.13, under each of the three considered background workloads.

capability to provide temporal guarantees.

The primary advantage of *cyclictest* is that it provides an easy-to-interpret metric that reflects various sources of unpredictability as a single, opaque measure. That is, it treats the kernel and the underlying hardware as a black box and reports the actual cumulative impact of system overheads and hardware capabilities on real-time tasks. For application developers, this is convenient as it requires neither post-tracing analysis nor a detailed understanding of the kernel.

In contrast, Feather-Trace yields a large number of (non-human-readable) event timestamps that require matching, filtering, post-processing, and a statistical evaluation. The resulting overhead profile is primarily intended for integration into schedulability analysis and is less suitable to direct interpretation. However, while *cyclictest* is arguably more convenient, LITMUS$^{\mathrm{RT}}$'s Feather-Trace approach provides a more complete picture since it yields the data required to assess the impact of kernel overheads on tasks other than the highest-priority task, as we explain next.

The main feature of Feather-Trace is that it integrates many tracepoints in the kernel, which can be used to collect fine-grained overheads. By measuring and considering the various sources of delay *individually*, a detailed *analysis* of the worst-case cumulative delay can be carried out.

For example, for a task other than the highest-priority task, the cumulative delay incurred depends on the worst-case scheduling latency *and* the delays due to preemptions by higher-priority tasks, which in turn depends on context-switching overheads, scheduling overheads in the presence of potentially many ready tasks, and so on. With Feather-Trace in LITMUS$^{\mathrm{RT}}$, it is possible to measure all these aspects individually, and then account for them during schedulability analysis (see [10, Ch. 3] for a comprehensive introduction to overhead accounting), such that the observed worst-case overheads are fully reflected in the derived temporal guarantees for *all* tasks (and not just the highest-priority task).

As another example, consider how tasks are resumed under partitioned schedulers such as the P-FP plugin (or SCHED_FIFO with appropriate processor affinity masks). If a real-time task resumes on a remote processor (*i.e.*, any processor other than its assigned partition), an *inter-processor interrupt* (IPI) must be sent to its assigned processor to trigger the scheduler. IPIs are of course not delivered and processed instantaneously in a real system and thus affect scheduling latency *if they arise*. When scheduling *cyclictest* on hardware platforms with processor-local timers (such as local APIC timers in modern x86 systems), however, such IPIs are not required because the interrupt signaling the expiry of *cyclictest*'s one-shot timer is handled locally. If we simply execute *cyclictest* under PSN-EDF, P-FP, or SCHED_FIFO with appropriate processor affinity masks to determine "the worst-case latency," it will never trace the impact of such IPIs, even though an actual real-time application that is triggered by interrupts from devices other than timers (*e.g.*, such as a sensor) would actually be subject to IPI delays. In contrast, in the methodology used to evaluate LITMUS$^{\mathrm{RT}}$ (see [10, Ch. 4]), Feather-Trace is used to measure IPI latencies, which are then correctly accounted for in the schedulability analysis to reflect the worst-case task-activation delay.

In summary, it is impossible to derive how real-time tasks other than the highest-priority task are affected by overheads from *cyclictest*-based experiments, because overhead-aware schedulability analysis is fundamentally required to make temporal guarantees for all tasks. Such an analysis is made possible by Feather-Trace's ability to extract specific overheads. While obtaining measurements in a fine-grained manner is more involved than simply running *cyclictest*, Feather-Trace's fine-grained measurement approach provides a flexibility that is not achievable with coarse-grained approaches such as *cyclictest*. This, of course, does not diminish *cyclictest*'s value as a quick assessment and debugging aid, but it should not be mistaken to provide a general measure of a system's "real-time capability"; it can only show the lack of such capability under certain circumstances—for instance, by exposing scheduling latencies in excess of $5ms$ in the presence of I/O-bound background tasks.

5 Conclusion and Future Work

We presented an empirical evaluation of scheduling latency under LITMUS$^{\mathrm{RT}}$ using *cyclictest*. We ported *cyclictest* to LITMUS$^{\mathrm{RT}}$'s native API and collected samples of scheduling latency under several of its event-driven scheduler plugins, in three system configurations (an idle system, a system with CPU-bound background tasks, and a system with I/O-bound background tasks). For the purpose of comparison, we repeated the same measurements under Linux 3.0, Linux 3.8.13, and Linux 3.8.13 with the PREEMPT_RT patch using the original, unmodified *cyclictest* version.

The results obtained from an idle system and in the presence of CPU-bound background tasks showed that while LITMUS$^{\mathrm{RT}}$ introduces some additional overheads, the difference is minor in absolute terms and manifests only in the average and median scheduling latencies. Importantly, LITMUS$^{\mathrm{RT}}$ was not observed to affect the maximum scheduling latencies negatively, which is due to the fact that other factors in mainline Linux have a much larger impact on worst-case delays. We conclude from these observations that LITMUS$^{\mathrm{RT}}$ does not impose inherently impractical overheads. Further, we believe that the observed minor increase in average and median scheduling latency is not fundamental, but caused by a lack of low-level optimizations that could be rectified with additional engineering effort.

However, our data also documents that LITMUS$^{\mathrm{RT}}$ inherits mainline Linux's weaknesses in the presence of I/O-bound background tasks. Again, LITMUS$^{\mathrm{RT}}$ did not increase the observed maximum scheduling latency, but the latency profile of the underlying Linux 3.0 kernel renders it unfit for serious (hard) real-time applications. Further, our experiments confirmed that this is still the case with the more recent mainline Linux version 3.8.13. It would thus be highly desirable to combine LITMUS$^{\mathrm{RT}}$'s algorithmic improvements with the increased responsiveness under load achieved by the PREEMPT_RT patch, which remains as future work.

References

[1] The LITMUS$^{\mathrm{RT}}$ project. `http://www.litmus-rt.org`.

[2] Real-time linux wiki. cyclictest - RTwiki. `https://rt.wiki.kernel.org/index.php/Cyclictest`.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of synchronous periodic tasks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85. IEEE, 2001.

[4] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[5] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proc. of the 31st Real-Time Systems Symposium*, pages 14–24, 2010.

[6] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, pages 125–135, 2011.

[7] A. Block. *Adaptive multiprocessor real-time systems*. PhD thesis, University of North Carolina at Chapel Hill, 2008.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

[9] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly & Associates Inc, third edition, 2005.

[10] B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[11] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. of the Workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 61–70, 2007.

[12] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proc. of the 12th Intl. Conference on Principles of Distributed Systems*, pages 105–124, 2008.

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: a status report. *9th Real-Time Linux Workshop*, 2007.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

[15] G. Chanteperdrix and R. Cochran. The ARM fast context switch extension for Linux. *Real Time Linux Workshop*, 2009.

[16] R. Cochran, C. Marinescu, and C. Riesch. Synchronizing the Linux system time to a PTP hardware clock. In *Proc. of the 2011 Intl. IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, pages 87–92, 2011.

[17] R. Coker. *bonnie++ — program to test hard drive performance*. Linux manual page.

[18] U. Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, Chapel Hill, NC, USA, 2006.

[19] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[20] C. Emde. Long-term monitoring of apparent latency in PREEMPT_RT Linux real-time systems. *12th Real-Time Linux Workshop*, 2010.

[21] L. Fu and R. Schwebel. Real-time linux wiki. RT PREEMPT HOWTO. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`.

[22] L. Henriques. Threaded IRQs on Linux PREEMPT-RT. In *Proc. of the 5th Intl. Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2009.

[23] C. Kenna, J. Herman, B. Brandenburg, A. Mills, and J. Anderson. Soft real-time on multiprocessors: are analysis-based schedulers really worth it? In *Proc. of the 32nd Real-Time Systems Symposium*, pages 93–103, 2011.

[24] J. Kiszka. Towards Linux as a real-time hypervisor. In *Proc. of the 11th Real-Time Linux Workshop*, pages 205–214, 2009.

[25] K. Koolwal. Investigating latency effects of the linux real-time preemption patches (PREEMPT_RT) on AMD's GEODE LX platform. In *Proc. of the 11th Real-Time Linux Workshop*, pages 131–146, 2009.

[26] A. Lackorzynski, J. Danisevskis, J. Nordholz, and M. Peter. Real-time performance of L4Linux. In *Proc. of the 13th Real-Time Linux Workshop*, pages 117–124, 2011.

[27] P. McKenney. A realtime preemption overview. 2005. LWN. http://lwn.net/Articles/146861/.

[28] A. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, 1983.

[29] M. Traut. Real-time CORBA performance on Linux-RT PREEMPT. *9th Real-Time Linux Workshop*, 2007.

[30] C. Williams and D. Sommerseth. *hackbench — scheduler benchmark/stress test*. Linux manual page.

[31] B. Zuo, K. Chen, A. Liang, H. Guan, J. Zhang, R. Ma, and H. Yang. Performance tuning towards a KVM-based low latency virtualization system. In *Proc. of the 2nd Internation Conference on Information Engineering and Computer Science*, pages 1–4. IEEE, 2010.