PRACTISE: a framework for PeRformance Analysis and Testing of real-time multIcore SchEdulers for the Linux kernel *

Fabio Falzoi, Juri Lelli, Giuseppe Lipari
{name.surname}@sssup.it - Scuola Superiore Sant'Anna - ITALY
Technical Paper

Abstract

The implementation of a multi-core real-time scheduler is a difficult task, due to the many problems that the developer encounters in debugging and testing parallel code inside the kernel. Also, it is often very difficult to compare the performance of alternative implementations, due to the many different parameters that influence the behaviour of the code.

In this paper, we present PRACTISE, a tool for developing, debugging, testing and analyse real-time scheduling data structures in user space. Unlike other similar tools, PRACTISE executes code in parallel, allowing to test and analyse the performance of the code in a realistic multiprocessor scenario.

After describing the organisation of the code, we show how easy it is to port the developed code in the Linux kernel. Also, we compare the performance of two different schedulers, SCHED_DEADLINE and SCHED_FIFO in the kernel and in the tool. Although PRACTISE cannot substitute performance analysis in the kernel, we show that, under certain conditions, PRACTISE can also be used as tool for early performance estimation.

1. Introduction

The wide diffusion of multi-core architectures in personal computing, servers and embedded systems, has revived the interest in multiprocessor scheduling, especially in the field of real-time applications. In fact, real-time scheduling on multi-core and multiprocessor systems is still an open research field both from the point of view of the theory and for the technical difficulties in implementing an efficient scheduling algorithm in the kernel.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme n.248465 "S(o)OS – Service-oriented Operating Systems."

Regarding the second problem, let us discuss a (non exhaustive) list of problems that the prospective developer of a new scheduler must be faced with. The task scheduler is a fundamental part of the operating system kernel: a buggy scheduler will soon crash the system, usually at random and unexpected points. The major difficulty in testing and debugging a new scheduling algorithm derives from the fact that, when the system crashes, it is difficult to reconstruct the situation (i.e. the sequence of events and states) that led to the crash. The developer has to carefully analyse system logs and traces (for example using one of the tools described in Section 3), and reconstruct the state to understand what went wrong. More importantly, it is often impossible to impose a precise sequence of events: crashes can rarely be reproduced deterministically. Hence, it is practically impossible to run a sequence of test-cases.

This problem is exacerbated in multi-core architectures where the scheduler service routines run in parallel on the different processors, and make use of shared data structures that are accessed in parallel. In these cases, it is necessary to ensure that the data structures remain consistent under every possible interleaving of the service functions. A simple solution is to protect the shared data structure with locks. However, a single big lock reduces parallelism and performance does not scale; fine-grain locks may cause deadlock situations, without improving scalability; and lock-free algorithms are difficult to implement and prove correct. As a consequence, many important and interesting scheduling algorithms proposed in the research literature fail to be implemented on popular operating systems like Linux due to the difficulty of the task.

One reasonable approach would be to develop, debug, test and analyse the algorithms in user space. Once the main algorithm is sufficiently tested using user-space debugging and testing techniques, the same algorithm can be ported in the kernel. However, if no specific methodology is followed, the code must be written twice, increasing the possibility of introducing bugs in one of the two versions. Also, if one is unsure of

which algorithm, data structure or locking strategy is more appropriate, the number of versions to implement, test, analyse by hand may become very large.

Hence, we decided to tackle the "user-space approach" by proposing a simple framework to facilitate the development, testing and performance evaluation of scheduling algorithms in user space, and minimise the effort of porting the same algorithms in kernel spaces.

1.1. Contributions of this work

In this paper, we propose PRACTISE (PeRformance Analysis and TestIng of real-time multicore SchEdulers) for the Linux kernel: it is a framework for developing, testing and debugging scheduling algorithms in user space before implementing them in the Linux kernel, that alleviates at least part of the problems discussed above. In addition, PRACTISE allows to compare different implementations by providing early estimations of their relative performance. In this way, the most appropriate data structures and scheduler structure can be chosen and evaluated in user-space. Compared to other similar tools, like LinSched, the proposed framework allows true parallelism thus permitting a full test in a realistic scenario (a short comparison between is done in Section 2)

The main features of PRACTISE are:

- Rapid prototyping of scheduling data structures in user space;
- Effective, quick and extensive testing of the data structures though consistency tests;
- Real multi-core parallelism using multi-threading;
- Relative performance estimation between different algorithms and data structures in user space;
- Possibility to specify application load though probabilistic distributions of events, and statistical analysis;
- Ease of porting to the kernel or to other scheduling simulators.

PRACTISE is available as open source software, and a development version is available for download[1].

The rest of the paper is organised as follows. In Section 2 we provide a survey of existing testing and debugging tools for the Linux kernel; in Section 3 we describe the architecture of the tool and the main implementation choices that we followed; in Section 4 we evaluate the tool by reporting the performance as measured by the tools compared to the performance of the same algorithms in the kernel; finally, in Section 5 we discuss conclusion and future work.

2. State of the art

Several tools exist, as open-source software, that are geared towards, or can be used as effective means to implement, debug and analyse real-time scheduling algorithms for multiprocessor systems. Each one tackles the intrinsic toughness of this field from different angles, generally focusing on one single aspect of the problem.

A valuable tool during the development process of a scheduling algorithm would be the one that allows fast prototyping and easy debugging. Originally developed by the Real Time Systems Group at University of North Carolina at Chapel Hill, and currently maintained by P. Turner from Google, **LinSched** [3][2] lets developers modify the behaviour of the Linux scheduler and test these changes in user-space. One of the major strength points of this tool is that it introduces very few modifications in the kernel sources. The developer can thus write kernel code and, once satisfied by tests, it has kernel ready patches at hand. Furthermore, debugging is facilitated by the fact that LinSched runs as a single thread user-space program, that can hence be debugged with common user-space tools like GDB[3]. Even if single-threading is useful for debugging purposes, it can be a notable drawback when focusing on the analysis of behaviour assuming a high degree of concurrency. LinSched can indeed verify locking, but it cannot precisely model multi-core contention.

LITMUS^{RT}[1] has a completely different focus. The *LITMUS^{RT}* patch, developed by the Real Time Systems Group at University of North Carolina at Chapel Hill, is a (soft) real-time extension of the Linux kernel that allows fast prototyping and evaluation of real-time (multiprocessor) scheduling algorithms on real hardware. The *LITMUS^{RT}* testbed provides an experimental platform that real-time system researchers can use to simplify the development process of scheduling and synchronisation algorithms (compared to modifying a stock Linux kernel). Another nice feature of this testbed is an integrated tracing infrastructure (FeatherTrace [2]) with which performance and overhead data can be collected for off-line processing. Being a research tool rather than a production-quality system, *LITMUS^{RT}* does not target Linux mainline inclusion nor POSIX-compliance: in other words code patches

[1] At the time of submission (April 29 2012), the software can be downloaded cloning the repository available at `https://github.com/Pippolo84/PRAcTISE`

[2] v3.3-rc7 release announce: http://bit.ly/IJsyV3.

[3] http://sources.redhat.com/gdb/

created with it cannot be seamless applied to a "vanilla" Linux kernel.

Lots of other tools exist that make kernel developers lives easier during debugging, some of them can also be used to collect performance data or even extract execution traces from a running system. Among others, these are probably part of every kernel developer arsenal:

- **KVM** [4] **+ GDB**: the very first step after having modified the kernel is usually to run it on a virtualized environment. The KVM virtual machine can here be useful as it can be attached, and controlled, by the GNU Project Debugger (GDB). However, this solution can hardly be used in presence of high concurrency; moreover, it can occasionally affect the repeatability of certain bugs.
- **perf**[9]: the performance counter subsystem in Linux can be used to collect scheduling events and performance data from a real execution. It can also be used in conjunction with LinSched, as it can record an application behaviour that can later be played back in the simulator.
- **Ftrace**[11]: a tracing utility built directly into the Linux kernel. Ftrace is a valuable debugging tool as it brings to Linux the ability to see what is happening inside the kernel. With the ability of synchronise a user-space testing application with kernel execution, one can track function calls up to the point where a bug may happen.
- **LTTng**[4, 5]: the Linux Trace Toolkit is an highly efficient tracing tool for Linux that can help tracking down performance issues and debugging problems involving concurrent execution.

PRACTISE adds one more powerful weapon to this arsenal: the possibility to test and analyse parallel code (like lock-free data structures) in user space via multithreading.

3. PRACTISE Architecture

In this section, we describe the basic structure of our tool. PRACTISE emulates the behaviour of the LINUX scheduler subsystem on a multi-core architecture with M parallel cores. The tool can be executed on a machine with N cores, with N that can be less, equal to or greater than M. The tool can be executed in one of the following modes:

- testing;
- performance analysis.

Each processor in the simulated system is modelled by a software thread that performs a cycle in which:

- scheduling events are generated at random;
- the corresponding scheduling functions are invoked;
- statistics are collected.

In *testing mode*, a special "testing" thread is executed periodically that performs consistency checks on the shared data structures. In the *performance analysis mode*, instead, each thread is *pinned* on a processor, and the memory is locked to avoid spurious page faults; for this reason, to obtain realistic performances it is necessary to set $M \leq N$.

3.1. Ready queues

The Linux kernel scheduler uses one separate ready queue per each processor. A ready task is always enqueued in one (and only one) of these queues, even when it is not executing. This organisation is tailored for partitioned schedulers and when the frequency of task migration is very low. For example, in the case of non real-time best effort scheduling, a task usually stays on the same processor, and periodically a load-balancing algorithm is called to distribute the load across all processors.

This organisation may or may not be the best one for global scheduling policies. For example the `SCHED_FIFO` and `SCHED_RR` policies, as dictated by the POSIX standard, requires that the m highest priority tasks are scheduled at every instant. Therefore, a task can migrate several times, even during the same periodic instance.

The current multi-queue structure is certainly not mandatory: a new and different scheduler could use a totally different data structure (for example a single global ready queue); however, the current structure is intertwined with the rest of the kernel and we believe that it would be difficult to change it without requiring major changes in the rest of the scheduler. Therefore, in the current version of PRACTISE we maintained the structure of distributed queues as it is in the kernel. We plan to extend and generalise this structure in future versions of the tool.

Migration between queues is done using two basic functions: *push* and *pull*. The first one tries to migrate a task from the local queue of the processor that calls the function to a remote processor queue. In order to do this, it may use additional global data structures to

[4] Kernel Based Virtual Machine: http://bit.ly/IdlzXi

select the most appropriate queue. For example: the current implementation of the fixed priority scheduler in Linux uses a priority map (implemented in *cpupri.c*) that records for each processor the priority of the highest priority tasks; the SCHED_DEADLINE [7, 8] patch uses a max heap to store the deadlines of the tasks executing on the processors.

The *pull* does the reverse operation: it searches for a task to "pull" from a remote processor queue to the local queue of the processor that calls the function. In the current implementation of SCHED_{FIFO,RR} and SCHED_DEADLINE, no special data structure is used to speed up this operation. We developed and tested in PRACTISE a min-heap for reducing the duration of the *pull* operation, but, driven by not satisfactory performance figures, we didn't port the structure inside the kernel. Instead, we are currently investigating several different data structures with the aim of comparing them and select the most efficient to be implemented in the next release of the SCHED_DEADLINE patch.

Tasks are inserted into (removed from) the ready queues using the `enqueue()` (`dequeue()`) function, respectively. In Linux, the queues are implemented as red-black trees. In PRACTISE, instead, we have implemented them as priority heaps, using the data structure proposed by B. Brandenburg [5]. However, it is possible to implement different algorithms for queue management as part of the framework: as a future work, we plan to implement alternative data structures that use lock-free algorithms.

3.2. Locking and synchronisation

PRACTISE uses a range of locking and synchronisation mechanisms that mimic the corresponding mechanisms in the Linux kernel. An exhaustive list is given in Table 1. These differences are major culprits for the slight changes needed to port code developed on the tool in the kernel 4.1.

It has to be noted that `wmb` and `rmb` kernel memory barriers have no corresponding operations in user-space; therefore we have to issue a full memory barrier (`__sync_synchronize`) for every occurrence of them.

3.3. Event generation and processing

PRACTISE cannot execute or simulate a real application. Instead, each threads (that emulates a processor) periodically generates random scheduling events according to a certain distribution, and calls the scheduler functions. Our goals are to debug, test, compare and evaluate real-time scheduling algorithms for multicore processors. Therefore, we identified two main events: task *activation* and *blocking*. When a task is activated, it must be inserted in one of the kernel ready queues; since such an event can cause a preemption, the scheduler is invoked, data structures are updated, etc. Something similar happens when a task self-suspends (for example because it blocks on a semaphore, or it suspends on a timer).

The pseudo-code for the task activation is function `on_activation()` described in Figure 1. The code mimics the sequence of events that are performed in the Linux code:

- First, the task is inserted in the local queue.
- Then, the scheduler performs a *pre-schedule*, corresponding to `pull()`, which looks at the global data structure `pull_struct` to find the task to be pulled; if it finds it, does a sequence of `dequeue()` and `enqueue()`.
- Then, the Linux scheduler performs the real schedule function; this corresponds to setting the `curr` pointer to the executing task. In PRACTISE this step is skipped, as there is no real context switch to be performed.
- Finally, a *post-schedule* is performed, consisting of a `push()` operation, which looks at the global data structure `push_struct` to see if some task need to be migrated, and in case the response is positive, performs a `dequeue()` followed by an `enqueue()`. A similar thing happens when a task blocks (see function `on_block()`).

The pseudo code shown in Figure 1 is an overly simplified, schematic version of the code in the tool; the interested reader can refer to the original source code[6] for additional details.

As anticipated, every processor is simulated by a periodic thread. The thread period can be selected from the command line and represents the average frequency of events arriving at the processor. At every cycle, the thread randomly select one between the following events: `activation`, `early finish` and `idle`. In the first case, a task is generated with a random value of the deadline and function `on_activation()` is called. In the second case, the task currently executing on the processor blocks: therefore function `on_block()` is called. In the last case, nothing happens. Additionally, in all cases, the deadline of the executing task is checked against the current time: if the

[5] Code available here: `http://bit.ly/IozLxM`.

[6] `https://github.com/Pippolo84/PRAcTISE`

Linux	PRAcTISE	Action
raw_spin_lock	pthread_spin_lock	lock a structure
raw_spin_unlock	pthread_spin_unlock	unlock a structure
atomic_inc	__sync_fetch_and_add	add a value in memory atomically
atomic_dec	__sync_fetch_and_sub	subtract a value in memory atomically
atomic_read	simple read	read a value from memory
wmb	__sync_synchronize	issue a memory barrier
rmb	__sync_synchronize	issue a read memory barrier
mb	__sync_synchronize	issue a full memory barrier

Table 1: Locking and synchronisation mechanisms (Linux vs. PRAcTISE).

```
pull() {
  bool found = find(pull_struct, &queue);
  if (found) {
    dequeue(&task, queue);
    enqueue(task, local_queue);
  }
}

push() {
  bool found = find(push_struct, &queue);
  if (found) {
    dequeue(&task, local_queue);
    enqueue(task, queue);
  }
}

on_activation(task) {
  enqueue(task, local_queue);
  pull();      /* pre-schedule  */
  push();      /* post-schedule */
}

on_block(task) {
  dequeue(&task, local_queue);
  pull();      /* pre-schedule  */
  push();      /* post-schedule */
}
```

Figure 1: Main scheduling functions in PRACTISE

deadline has passed, then the current task is blocked, and function `on_block()` is called.

Currently, it is possible to specify the period of the thread cycle; the probability of an activation event; and the probability of an early finish.

3.4. Data structures in PRACTISE

PRACTISE has a modular structure, tailored to provide flexibility in developing new algorithms. The interface exposed to the user consists of hooks to functions that each global structure must provide. The most important hooks:

- `data_init`: initialises the structure, e.g., spin-lock init, dynamic memory allocation, etc.
- `data_cleanup`: performs clean up tasks at the end of a simulation.
- `data_preempt`: called each time an `enqueue()` causes a preemption (the arriving tasks has higher priority that the currently executing one); modifies the global structure to reflect the new local queue status.
- `data_finish`: *data_preempt* dual (triggered by a `dequeue()`).
- `data_find`: used by a scheduling policy to find the best CPU to (from) which push (pull) a task.
- `data_check`: implements the *checker* mechanism (described below).

PRACTISE has already been used to slightly modify and validate the global structure we have previously implemented in SCHED_DEADLINE [8] to speed-up `push()` operations (called *cpudl* from here on). We also implemented a corresponding structure for `pull()` operations (and used the tool to gather performance data from both). Furthermore, we back-ported in PRACTISE the mechanism used by SCHED_FIFO to improve `push()` operations performance (called *cpupri* from here on).

We plan to exploit PRACTISE to investigate the use of different data structures to improve the efficiency of the aforementioned operations even further. However, we leave this task as future work, since this paper is focused on describing the tool itself.

One of the major features provided by PRACTISE is the *checking* infrastructure. Since each data structure has to obey different rules to preserve consistency among successive updates, the user has to equip the implemented algorithm with a proper checking function. When the tool is used in testing mode, the `data_check` function is called at regular intervals. Therefore, an on-line validation is performed in presence of real concurrency, thus increasing the probability of discovering bugs at an early stage of the development process. User-space debugging techniques can then be used to fix design or developing flaws.

To give the reader an example, the *checking* function for SCHED_DEADLINE *cpudl* structure ensures the max-heap property: if B is a child node of A, then $deadline(A) \geq deadline(B)$; it also check consistency between the heap and the array used to perform updates on intermediate nodes (see [8] for further details). We also implemented a checking function for *cpupri*: periodically, all ready queues are locked, and the content of the data structure is compared against the corresponding highest priority task in each queue, and the consistency of the flag `overloaded` in the `struct root_domain` is checked. We found that the data structure is always perfectly consistent to an external observer.

3.5. Statistics

To collect the measurements we use the TSC (Time Stamp Counter) of IA-32 and IA-64 Instruction Set Architectures. The TSC is a special 64-bit per-CPU register that is incremented every clock cycle. This register can be read with two different instructions: RDTSC and RDTSCP. The latter reads the TSC and other information about the CPUs that issues the instruction itself. However, there are a number of possible issues that needs to be addressed in order to have a reliable measure:

- *CPU frequency scaling and power management*. Modern CPUs can dynamically vary frequency to reduce energy consumption. Recently, CPUs manufacturer have introduced a special version of TSC inside their CPUs: *constant TSC*. This kind of register is always incremented at CPU maximum frequency, regardless of CPU actual frequency. Every CPU that supports that feature has the flag *constant_tsc* in `/proc/cpuinfo` proc file of Linux. Unfortunately, even if the update rate of TSC is constant in these conditions, the CPU frequency scaling can heavily alter measurements by slowing down the code unpredictably; hence, we have conducted every experiment with all CPUs at fixed maximum frequency and no power-saving features enabled.

- *TSC synchronisation between different cores*. Since every core has its own TSC, it is possible that a misalignment between different TSCs may occur. Even if the kernel runs a synchronisation routine at start up (as we can see in the kernel log message), the synchronisation accuracy is typically in the range of several hundred clock cycles. To avoid this problem, we have set CPU affinity of every thread with a specific CPU index. In other words we have a 1:1 association between threads and CPUs, fixed for the entire simulation time. In this way we also prevent thread migration during an operation, which may introduce unexpected delays.

- *CPU instruction reordering*. To avoid instruction reordering, we use two instructions that guarantees serialisation: RDTSCP and CPUID. The latter guarantees that no instructions can be moved over or beyond it, but has a non-negligible and variable calling overhead. The former, in contrast, only guarantees that no previous instructions will be moved over. In conclusion, as suggested in [10], we used the following sequence to measure a given code snippet:

```
CPUID
RDTSC
code
RDTSCP
CPUID
```

- *Compiler instruction reordering*. Even the compiler can reorder instructions; so we marked the inline asm code that reads and saves the TSC current value with the keyword *volatile*.

- *Page faults*. To avoid page fault time accounting we locked every page of the process in memory with a call to `mlockall`.

PRACTISE collects every measurement sample in a global multidimensional array, where we keep samples coming from different CPUs separated. After all simulation cycles are terminated, we print all of the samples to an output file.

By default, PRACTISE measures the following statistics:

- duration and number of *pull* and *push* operations;

- duration and number of *enqueue* and *dequeue* operations;

- duration and number of `data_preempt`, `data_finish` and `data_find`.

Of course, it is possible to add different measures in the code of a specific algorithm by using PRACTISE's functions. In the next section we report some experiment with the data structures currently implemented in PRACTISE.

4. Evaluation

In this section, we present our experience in implementing new data structures and algorithms for the Linux scheduler using PRACTISE. First, we show how difficult is to port a scheduler developed with the help of PRACTISE into the Linux kernel; then, we report performance analysis figures and discuss the different results obtained in user space with PRACTISE and inside the kernel.

4.1. Porting to Linux

The effort in porting an algorithm developed with PRACTISE in Linux can be estimated by counting the number of different lines of code in the two implementations. We have two global data structures implemented both in PRACTISE and in the Linux kernel: *cpudl* and *cpupri*.

We used the `diff` utility to compare differences between user-space and kernel code of each data structure. Results are summarised in Table 2. Less than 10% of changes were required to port *cpudl* to Linux, these differences mainly due to the framework interface (pointers conversions). Slightly higher changes ratio for *cpupri*, due to the quite heavy use of atomic operations (see Section 3.2). An example of such changes is given in Figure 2 (lines with a `-` correspond to user-space code, while those with a `+` to kernel code).

Structure	Modifications	Ratio
cpudl	12+ 14-	8.2%
cpupri	17+ 21-	14%

Table 2: Differences between user-space and kernel code.

The difference on the synchronisation code can be reduced by using appropriate macros. For example, we could introduce a macro that translates to `__sync_fetch_and_add` when compiled inside PRACTISE, and to the corresponding Linux code otherwise. However, we decided for the moment to maintain the different code to highlight the differences between the two frameworks. In fact, debugging, testing and analyse the synchronisation code is the main difficulty, and the main goal of PRACTISE; therefore, we thought that it is worth to show such differences rather than hide them.

```
[...]
-void cpupri_set(void *s, int cpu, int newpri)
+void cpupri_set(struct cpupri *cp, int cpu,
+                int newpri)
 {
-   struct cpupri *cp = (struct cpupri*) s;
    int *currpri = &cp->cpu_to_pri[cpu];
    int oldpri  = *currpri;
    int do_mb = 0;
@@ -63,57 +61,55 @@
    if (newpri == oldpri)
         return;

-   if (newpri != CPUPRI_INVALID) {
+   if (likely(newpri != CPUPRI_INVALID)) {
         struct cpupri_vec *vec =
                      &cp->pri_to_cpu[newpri];

         cpumask_set_cpu(cpu, vec->mask);
-        __sync_fetch_and_add(&vec->count, 1);
+        smp_mb__before_atomic_inc();
+        atomic_inc(&(vec)->count);
         do_mb = 1;
    }
[...]
```

Figure 2: Comparison using `diff`.

However, the amount of work shouldered on the developer to transfer the implemented algorithm to the kernel, after testing, is quite low reducing the probability of introducing bugs during the porting. Moreover, this residual amount of handwork could be eliminated using simple translation scripts (e.g., sed). Additional macros will be introduced in future version of PRACTISE to minimise such effort even further.

4.2. Experimental setup

The aim of the experimental evaluation is to compare performance measures obtained with PRACTISE with what can be extracted from the execution on a real machine.

Of course, we cannot expect the measures obtained with PRACTISE to compare directly with the measure obtained within the kernel; there are too many differences between the two execution environments to make the comparison possible: for example, the completely different synchronisation mechanisms. However, comparing the performance of two alternative algorithms within PRACTISE can give us an idea of their relative

performance within the kernel.

4.3. Results

In Linux, we rerun experiments from our previous work [8] on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD^R Opteron^{TM} 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores. This was necessary since the *cpupri* kernel data structure has been modified in the meanwhile [7] and the PRACTISE implementation is aligned with this last *cpupri* version. We generated 20 random task sets (using the `randfixedsum` [6] algorithm) with periods log-uniform distributed in [10ms, 100ms], per CPU utilisation of 0.6, 0.7 and 0.8 and considering 2, 4, 8, 16, 24, 32, 40 and 48 processors. Then, we ran each task set for 10 seconds using a synthetic benchmark [8] that lets each task execute for its WCET every period. We varied the number of active CPUs using the Linux CPU hot plug feature and we collected scheduler statistics through `sched_debug`. The results for the Linux kernel are reported in Figures 3a and 3b, for modifying and querying the data structures, respectively. The figures show the number of cycles (y axis) measured for different number of processors ranging from 2 to 48 (x axis). The measures are shown in boxplot format: a box indicates all data comprised between the 25% and the 75% percentiles, whereas an horizontal lines indicates the median value; also, the vertical lines extend from the minimum to the maximum value.

In PRACTISE we run the same experiments. As depicted in Section 3.3, random scheduling events generation is instead part of PRACTISE . We varied the number of active processors from 2 to 48 as in the former case.

We set the following parameters: 10 milliseconds of thread cycle; 20% probability of new arrival; 10% probability of finish earlier than deadline (cpudl) or runtime (cpupri); 70% probability of doing nothing. These probability values lead to rates of about 20 task activations / (core * s), and about 20 task blocking / (core * s).

The results are shown in Figures 6a and 5a for modifying the *cpupri* and *cpudl* data structures, respectively; and in Figures 6b and 5b for querying the *cpupri* and *cpudl* data structures, respectively.

Insightful observations can be made comparing performance figures for the same operation obtained from the kernel and from simulations. Looking at Figure 3a we see that modifying the *cpupri* data structure is generally faster than modifying *cpudl*: every measure corresponding to the former structure falls below 1000 cycles while the same operation on *cpudl* takes about 2000 cycles. Same trend can be noticed in Figure 6a and 5a. Points dispersion is generally a bit higher than in the previous cases; however median values for *cpupri* are strictly below 2000 cycles while *cpudl* never goes under that threshold. We can see that PRACTISE overestimates this measures: in Figure 6a we see that the estimation for the *find* operation on *cpupri* are about twice the ones measured in the kernel; however, the same happens for *cpudl* (in Figure 5a); therefore, the relative performance of both does not change.

Regarding query operations the ability of PRACTISE to provide an estimation of actual trends is even more evident. Figure 3b shows that a *find* on *cpudl* is generally more efficient than the same operation on *cpupri*; this was expected, because the former simple reads the top element of the heap. Comparing Figure 6b with Figure 5b we can state that latter operations are the most efficient also in the simulated environment.

Moreover, we used PRACTISE to compare the time needed to modify and query the two global data structure for push and pull operations for cpudl. As we can see in Figure 5a and Figure 5b compared against Figure 6a and Figure 6b, the results are the same, as the data structures used are the same. We haven't compared cpudl pull operation against cpupri pull operation since the latter doesn't have a global data structure that hold the status of all run queues where we can issue find and set operations.

5. Conclusions and future work

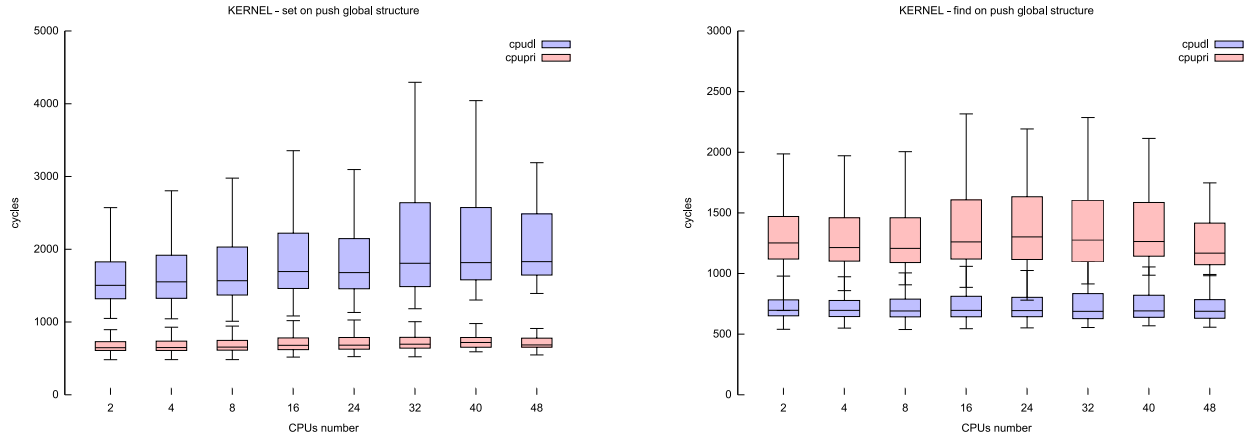
In this paper we introduced PRACTISE, a framework for PeRformance Analysis and TestIng of real-time multicore SchEdulers for the Linux kernel. PRACTISE enables fast prototyping of real-time multicore scheduling mechanisms, allowing easy debugging and testing of such mechanisms in user-space. Furthermore, we performed an experimental evaluation of the simulation environment, and we showed that PRACTISE can also be used to perform early performance estimation.

In future work, we plan to refine the framework adherence to the Linux kernel. In doing so, we have to enhance task affinity management, local run queues capabilities and provide the possibility to generate random scheduling events following probability distributions gathered from real task sets execution traces.

Furthermore, we will exploit PRACTISE to perform a comparative study of different data structures to improve *pull* operation performance. In particular we will try to implement some lock-free data structures and subsequently compare their performances against

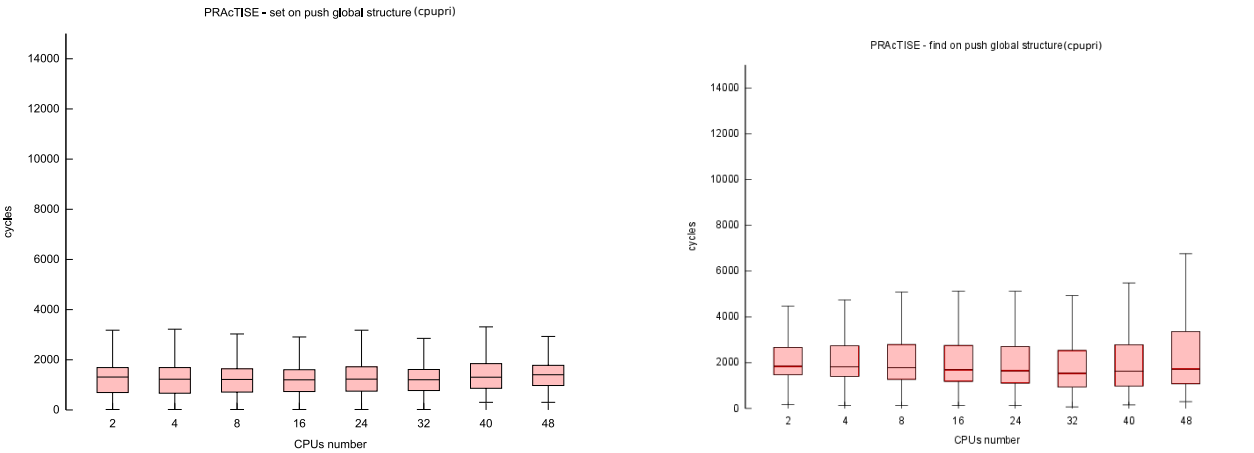
[7] More info here: `http://bit.ly/KjoePl`

[8] rt-app: `https://github.com/gbagnoli/rt-app`.



(a) modify (b) query

Figure 3: Number of cycles (mean) to a) modify and b) query the global data structure (*cpudl* vs. *cpupri*), kernel implementation.



(a) modify (b) query

Figure 4: Number of cycles (mean) to a) modify and b) query the global data structure (*cpupri*), on PRACTISE.

the heap already presented.

As a concluding use-case, it is worth mentioning that PRACTISE has already been used as a testing environment for the last SCHED_DEADLINE release on the LKML [9]. The *cpudl* global data structure underwent major changes that needed to be verified. The tested code has been finally merged within the patch set.

References

[1] Linux Testbed for Multiprocessor Scheduling in Real-Time Systems ($LITMUS^{RT}$). http://www.litmus-rt.org/index.html.

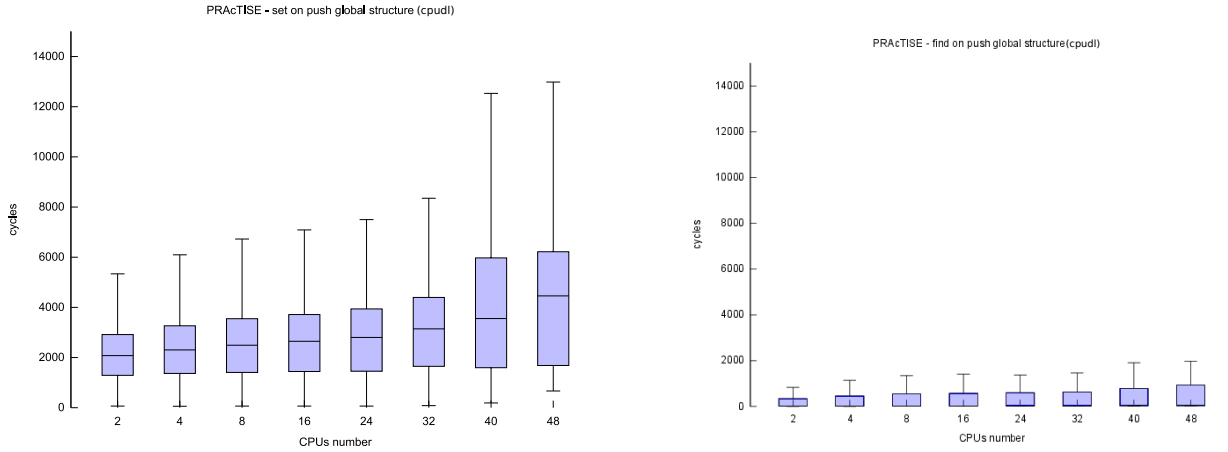
[2] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. 3th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2007)*, National ICT Australia, July 2007.

[3] John M. Calandrino, Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn. Linsched: The linux scheduler simulator. In J. Jacob and Dimitrios N. Serpanos, editors, *ISCA PDCCS*, pages 171–176. ISCA, 2008.

[4] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proc. Ottawa Linux Symposium (OLS 2006)*, pages 209–224, July 2006.

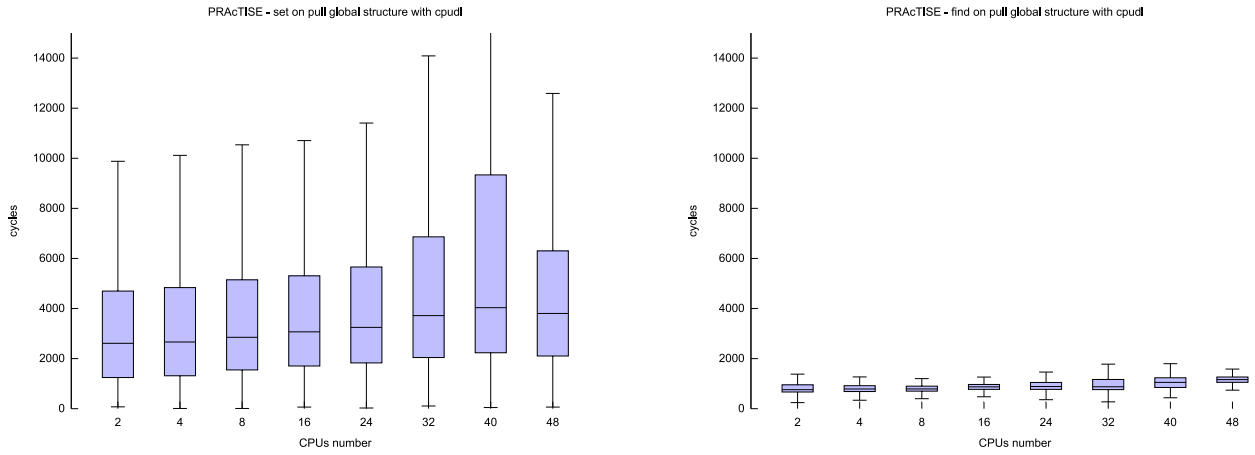
[5] Mathieu Desnoyers. Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer. Linux Foundation Collaboration Summit, April 2009.

[9] LKML (Linux Kernel Mailing List) thread available at: `https://lkml.org/lkml/2012/4/6/39`



(a) modify (b) query

Figure 5: Number of cycles (mean) to a) modify and b) query the global data structure (*cpudl*), on PRACTISE.



(a) modify (b) query

Figure 6: Number of cycles (mean) to a) modify and b) query the global data structure for speed-up SCHED_DEADLINE pull operations, on PRACTISE.

[6] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, Brussels, Belgium, July 2010.

[7] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC)*, Honolulu (USA), March 2009.

[8] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011)*. July 2011.

[9] Arnaldo Melo. The new linux 'perf' tools. In *17 International Linux System Technology Conference (Linux Kongress)*, Georg Simon Ohm University Nuremberg (Germany), September 21-24 2010.

[10] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 Instruction Set Architectures. Intel White Paper, September 2010.

[11] Steven Rostedt. The world of ftrace. Linux Foundation Collaboration Summit, April 2009.