Enhancing the Real-time Capabilities of the Linux Kernel

Paulo Baltarejo Sousa, Nuno Pereira, and Eduardo Tovar
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
4200-072 Porto, Portugal
{*pbs,nap,emt*}*@isep.ipp.pt*

***Abstract*—The mainline Linux Kernel is not designed for hard real-time systems; it only fits the requirements of soft real-time systems. In recent years, a kernel developer community has been working on the PREEMPT-RT patch. This patch (that aims to get a fully preemptible kernel) adds some real-time capabilities to the Linux kernel. However, in terms of scheduling policies, the real-time scheduling class of Linux is limited to the First-In-First-Out (`SCHED_FIFO`) and Round-Robin (`SCHED_RR`) scheduling policies. These scheduling policies are however quite limited in terms of real-time performance. Therefore, in this paper, we report one important contribution for adding more advanced real-time capabilities to the Linux Kernel. Specifically, we describe modifications to the (PREEMPT-RT patched) Linux kernel to support real-time slot-based task-splitting scheduling algorithms. Our preliminary evaluation shows that our implementation exhibits a real-time performance that is superior to the scheduling policies provided by the current version of PREMPT-RT. This is a significant add-on to a widely adopted operating system.**

I. Introduction

Multiprocessors implemented on a single chip, called *multicores*, are a mainstream computing technology. Multicores with 8 cores are common on desktops today and it is already possible to find commercial chips with up to 100 generic processing cores [1]. With chip manufacturers focused on improving performance by increasing the number of cores, it is expected that the number of cores per chip will continue to increase.

Due to its wide adoption, Linux is well positioned to take an important role leveraging the processing power of large multicores and its wide adoption is also driving developments towards enabling real-time computing by using the Linux kernel. The main objective of such efforts is reducing the unpredictability sources that exist in the mainline Linux kernel, as these can cause arbitrary delays to the real-time tasks running on the system.

There are many sources of unpredictability in the Linux kernel: (i) interrupts are generated by the hardware often in an unpredictable manner and when an interrupt arrives, the processor execution switches to handle it; (ii) multiple kernel threads running on different processors in parallel can simultaneously operate on shared kernel data structures, requiring serialization of the access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions.

Currently, the PREEMPT-RT patch[1], is the foremost development effort towards supporting the execution of real-time tasks using the Linux kernel. The PREEMPT-RT patch addresses these sources of unpredictability by making most of the Linux kernel preemptible, by implementing priority inheritance (to avoid priority inversion phenomena), and by converting interrupt handlers into preemptible kernel threads. These are important properties to enable real-time computing. However, appropriate real-time scheduling policies are also needed.

The real-time scheduling class implemented in the PREEMPT-RT patch supports the same scheduling policies of the mainline Linux kernel: the First-In-First-Out (`SCHED_FIFO`) and Round-Robin (`SCHED_RR`) scheduling policies. While these scheduling policies are appropriate for unicore processors, they are not adequate for multicore (or multiprocessor) systems because (i) their performance is poor on multiprocessors - there exist task sets where a system with a load a little over 50% will fail to meet deadlines and (ii) they adopt an active push-pull approach for balancing tasks across processors - since the Linux kernel uses a per-processor runqueue (a runqueue stores ready tasks), such push-pull operations require locking multiple processor runqueues, which are an additional source of unpredictability.

This paper describes the modifications of the (PREEMPT-RT patched) Linux 3.2.11-rt20 kernel to support real-time task-splitting scheduling algorithms where the time is divided into timeslots (called slot-based task-splitting). Slot-based task-splitting scheduling algorithms [2], [3] assign most tasks (called *non-split tasks*) to just one processor and a few (called *split tasks*) to only two processors and have a utilization bound of 65%, configurable up to arbitrarily close to 100% at the cost of more preemptions and migrations.

Among existing slot-based task-splitting scheduling algorithms, the Notional Processor Scheduling – Fractional capacity (NPS-F) [3] is notable for its high utilization bound (configurable from 75% up to arbitrarily close to 100%) and is the focus of the implementation reported in this paper. NPS-F is a semi-partitioned multiprocessor scheduling algorithm: tasks are partitioned to servers (termed *notional processors*), in turn mapped onto the (physical)

[1]Available online at http://www.kernel.org/pub/linux/kernel/projects/rt/

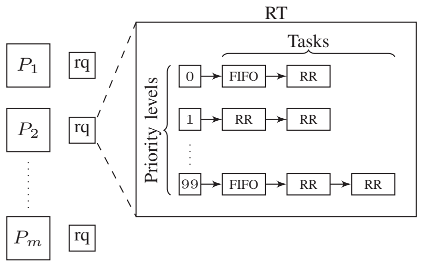

Figure 1. RT scheduling class Runqueue

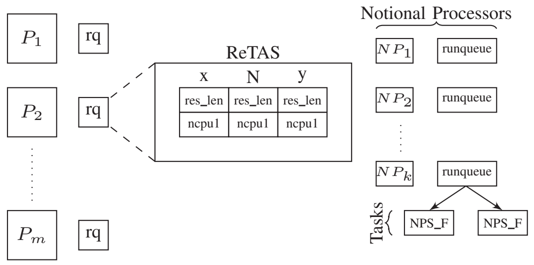

Figure 2. ReTAS Runqueue

processors. Some notional processors use just one physical processor; others use two processors and migrate between them in a controlled manner.

This work is an evolution of [4], which implements the same scheduling algorithms by modifying the mainline Linux Kernel. In that previous work, a scheduling policy module – ReTAS (Real-time TAsk-Splitting) – was added on top of the native Linux module hierarchy, making ReTAS the highest priority module. However, such approach cannot be employed in conjunction with the PREEMPT-RT patch because important functionalities, such as timer interrupt handlers, needed by the ReTAS scheduler, are implemented within the real-time scheduling class and thus ReTAS cannot have a higher priority than the real-time scheduling class implemented in the PREEMPT-RT patch.

II. Background on Real-Time Scheduling in the Linux Kernel and PREEMPT-RT

The linux kernel scheduler consists of a scheduler core (or dispatcher) and various modules, where each module implements a scheduling class encapsulating a scheduling policy. These scheduler modules are hierarchically organized by priority and the dispatcher looks for a runnable task of each module in a decreasing order of priorities. Currently, the Linux kernel implements three native scheduler modules: RT (Real-Time), CFS (Completely Fair Scheduling) and Idle. The dispatcher first inquires the RT module for a runnable task and, if this module does not have any ready task, the dispatcher then inquires the CFS module. The Idle module is used for the idle task, executed when there is no other runnable task.

As depicted in Figure 1, tasks in the RT scheduling class are organized by priority level. Inside each priority level, the RT module implements two scheduling heuristics: `SCHED_FIFO` and `SCHED_RR`. `SCHED_FIFO` is based on the first-in-first-out heuristic: whenever a `SCHED_FIFO` task is executing, it continues until preempted (by a higher-priority task) or blocked (e.g., by an I/O operation). `SCHED_RR` implements the round-robin heuristic: a `SCHED_RR` task executes (if it is not preempted or blocked) until it exhausts its timeslice.

The mainline Linux defines one runqueue (an instance of `struct rq`, where all ready tasks are stored) per-physical processor and, at any time instant, the processor is executing one task stored in its runqueue. This may result in unbalanced workloads across processors. In order to balance the workload across processors, the RT module adopts an active push-pull strategy as follows: whenever the dispatcher inquires the RT module, it first tries to pull the non-executing highest-priority task from the other runqueue (if it is not in its runqueue) and, after selecting the next running task, it checks if it can push the (freshly) preempted task to another processor which is executing a task with lower priority than the preempted task. Observe that, moving tasks between two runqueues requires locking both runqueues and this may introduce considerable overheads.

The PREEMPT-RT patch reduces the kernel latencies by reducing its non-preemptible sections. This is done by replacing most kernel spinlocks by mutexes, which support priority inheritance, and by transforming all interrupts handlers into preemptive kernel threads, scheduled by the RT scheduling class. These kernel threads have assigned a priority level (50 by default) and, therefore, they can be preempted by other RT tasks with higher-priority. As mentioned before, the RT scheduling class does not implement any scheduling algorithm suitable for multiprocessor systems and the PREEMPT-RT patch does not add any scheduling algorithms suitable for multiprocessor systems.

III. Implementing slot-based task-splitting

Achieving an effective implementation of NPS-F in the Linux Kernel is a challenging task, as it requires efficient mechanisms to: (i) handle migrations; (ii) manage ready tasks; (iii) handle reserves and (iv) mapping of notional processors to physical processors. The following section discusses these challenges and Section III-B describes how the ReTAS scheduler module was integrated into the RT scheduling class.

A. Issues in implementing NPS-F

As explained before, the mainline Linux kernel may incur in overheads due to the way task migrations are implemented. In NPS-F, migrations involve the entire notional processor. As this would typically imply moving multiple tasks, adopting a similar strategy for the implementation of NPS-F would be inefficient. Indeed, our implementation employs a different arrangement that largely solves these issues. Namely, we opt for one runqueue per *notional* (not per physical) processor (see Figure 2 - ReTAS is used to denote the implementation of slot-based task-splitting scheduling algorithms). Under this approach, all ready tasks assigned to a notional processor are always stored on (i.e. inserted to/dequeued from) the same respective (per-notional-processor) runqueue. Then, when a notional processor migrates (i.e. with all its tasks) from processor P_p to processor P_{p+1}, we simply change the runqueues used by P_p and P_{p+1}.

To implement NPS-F, each physical processor needs to be configured with its timeslot composition. For this purpose, we introduce the following set of variables that store information about the processor reserves. The variable `begin_curr_timeslot` stores (as suggested by its name) the beginning of the current timeslot and it is incremented by S (the timeslot length). Observe that no synchronization mechanism is required for updates to this variable. The timeslot composition is defined by an array of 2-tuples `<res_len, ncpu1>` (see Figure 2). Each element of this array maps a reserve of length (`res_len`) to the notional processor (`ncpu1`). A timer is used to trigger scheduling decisions at the beginning of each reserve.

If one observes two consecutive timeslots, whenever a split notional processor consumes its reserve on processor P_p, whichever task was executing at the time has to "immediately" resume execution on another reserve on processor P_{p+1}. However, due to many sources of unpredictability, common in a real operating system, arbitrary levels of time precision are not possible. Consequently, the dispatcher of processor P_{p+1} can be prevented from selecting the task in consideration from execution because processor P_p has not yet relinquished (the runqueue associated with) that task.

One solution could be for processor P_{p+1} to send an interprocessor interrupt (IPI) to P_p to relinquish (the runqueue associated with) that split task. Another could be for P_{p+1} to set up a timer x time units in the future to force the invocation of its dispatcher. We chose the latter option for two reasons: (i) we know that if a dispatcher has not yet relinquished the split task it was because something is preventing it from doing so (e.g. the execution of an interrupt service routine (ISR)); (ii) using an IPI solution introduces a dependency between processors that can compromise the scalability of the dispatcher.

B. Adding ReTAS to the RT Scheduling Class

Apart from the required code to manipulate notional processors (enqueue and dequeue of ReTAS tasks as well as getting the task with earliest absolute deadline) and the timeslot infrastructure, incorporating ReTAS into the RT scheduling class implies a set of modifications in functions implemented in the `sched_rt.c` file. Those functions are `enqueue_task_rt`, `dequeue_task_rt`, `check_preempt_curr_rt`, and `pick_next_task_rt`.

The `enqueue_task_rt` is called whenever a RT task enters into a runnable state. If the runnable task is a ReTAS task (ReTAS tasks are also RT tasks, with a priority level, but are scheduled according to the `SCHED_NPS_F` scheduling policiy), then it is enqueued into the respective notional processor runqueue. When a RT task is no longer runnable, then the `dequeue_task_rt` function is called to remove the task from the respective notional processor runqueue.

As the name suggests, the `check_preempt_curr_rt` function (Listing 1) checks whether the currently running task must be preempted or not (e.g. when a RT task wakes up). It receives two pointers, one for the processor runqueue that is running this code (`rq`) and another to the woken up task (`p`). If the priority of the woken up task is higher or equal than (lower `prio` values mean higher priority) the currently executing task (pointed by `rq->curr`), it checks if `p` is a ReTAS task and if the current reserve is mapped to task `p` notional processor. If that is the case, then the currently running task is marked for preemption.

```
static void
check_preempt_curr_rt(struct rq *rq, struct task_struct *
     p, int flags)
{
  if (p->prio <= rq->curr->prio)
    if(retas_policy(p->policy))
      if(_check_preempt_curr_retas(rq)){
        resched_task(rq->curr);
        return;
      }
  ...
}
```

Listing 1. Changes on the `check_preempt_curr_rt` function.

The `pick_next_task_rt` function also needs a small modification. This function selects the task to be executed by the current processor and is called by the dispatcher whenever the currently executing task is marked to be preempted or finishes its execution. It first gets the highest-priority RT task, then it gets the highest-priority ReTAS task. After, it selects the highest-priority task between them, if there is some, otherwise it returns `NULL`, which forces the dispatcher to inquiry the CFS scheduling module.

IV. Evaluation

In order to compare the performance of our implementation[2] of the NPS-F scheduling policy with the Linux native real-time scheduling policies, we have conducted a range of experiments with a 4-core platform (Intel(R) Core(TM) i7 CPU @ 2.67GHz). A set of implicit-deadline task sets was generated as follows. We have defined four types of tasks: "normal", 'heavy", "medium", and "light", where normal tasks have a u_i (the individual task utilization) in the range 0.05 to 0.95. Heavy tasks have a u_i in the range 0.65 to

[2]The implementation is available at http://webpages.cister.isep.ipp.pt/~pbsousa/retas/

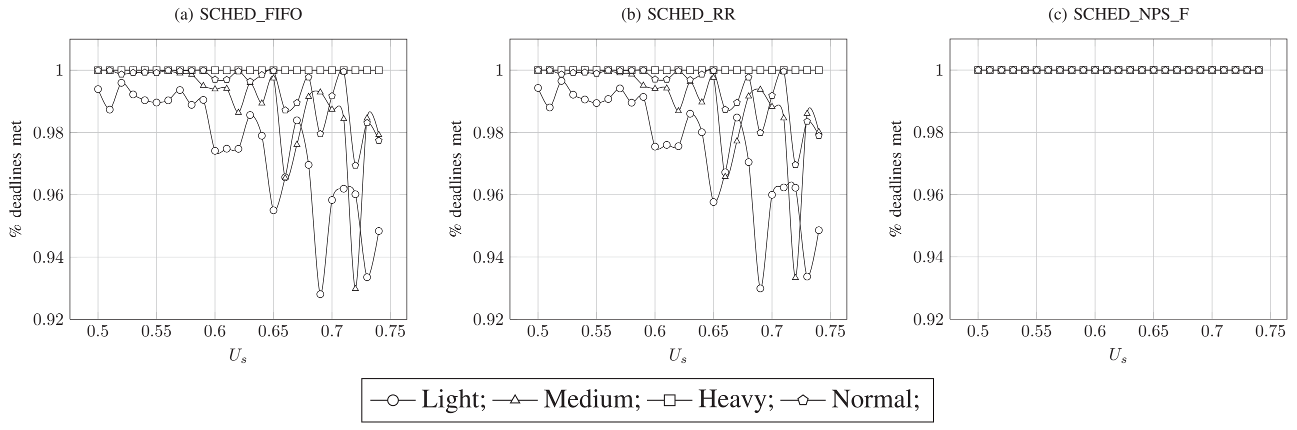

Figure 3. Success (deadlines met) Ratio of Tasks using SCHED_FIFO, SCHED_RR and SCHED_NPS_F

0.95, while medium tasks have a u_i in the range 0.35 to 0.65. Finally, light tasks have u_i in the range 0.05 to 0.35.

For each of the four task types, we generated 5 task sets and repeated this for 25 different U_s ($U_s = \sum_{\tau_i \in \tau} \frac{C_i}{T_i}$) values, varying from 0.50 to 0.75 (this value is the utilization bound of the NPS-F scheduling algorithm) with an increment of 0.01. The periodicity of all tasks was uniformly generated in the range 5 ms to 50 ms. The characteristics of these tasksets are not particularly suited for NPS-F. They were generated with the purpose of testing high-utilization periodic workloads with different characteristics. There is a relevant setup phase in NPS-F, prior to runtime, were tasks are assigned to processors. This phase is only part of NPS-F, and thus other algorithms have a slightly more simplified setup.

We ran each task set using the SCHED_FIFO, SCHED_RR, and SCHED_NPS_F for a total of over 39 hours. Figure 3 plots the percentage of all task instances executed during the experiment which met its deadline.

As it can be seen by inspecting Figure 3, that SCHED_NPS_F was the only scheduling policy that met all deadlines, as predicted by the theory. Both SCHED_FIFO and SCHED_RR fail to meet all deadlines. Observe that longer experiments could reveal a different percentage of deadlines met, however, the theory tells us that all deadlines will be met in a correct NPS-F scheduler. Interestingly, with heavy tasks, none of the schedulers fails deadlines. This is expected as, for this case, the task set generation method produces a number of tasks which is smaller or equal to m, thus each processor is only assigned one task.

V. Conclusions

This paper addressed the relevant problem of providing adequate real-time scheduling policies for multicore systems using the Linux kernel. We have overviewed the implementation challenges posed by this implementation and overviewed the structure and the main modifications introduced. We also presented an evaluation showing that our implementation is able to meet all deadlines and that its real-time performance is superior to that of the other real-time scheduling policies available in the Linux kernel.

Our contribution is completely compatible with the PREEMPT-RT patch and was implemented with minor modifications. In our opinion, adding adequate scheduling algorithm to the Linux kernel (compatible with PREEMPT-RT patch) is an important concern to make this widely adopted operating system more suitable for real-time systems.

Acknowledgment

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project Ref. FCOMP-01-0124-FEDER-022701 and also the REHEAT project, ref. FCOMP-01-0124-FEDER-010045.

References

[1] Tilera, "TILE-Gx processor family overview," http://www.tilera.com/products/processors/TILE-Gx_Family.

[2] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemption," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA 06)*, Sydney, Australia, 2006, pp. 322–334.

[3] K. Bletsas and B. Andersson, "Notional processors: an approach for multiprocessor scheduling," in *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, San Francisco, CA, USA, 2009, pp. 3–12.

[4] P. B. Sousa, K. Bletsas, E. Tovar, and B. Andersson, "On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems," in *Proc. of the 13th Real-Time Linux Workshop (RTLWS'13)*, Prague, Czech Republic, 2011, pp. 207–218.