Technical Report

On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems

Paulo Baltarejo Sousa

Konstantinos Bletsas

Eduardo Tovar

Björn Andersson

HURRAY-TR-110903

Version:

Date: 9/19/2011

On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems

Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, BjÃ¶rn Andersson

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: pbsousa@dei.isep.ipp.pt, ksbs@isep.ipp.pt, emt@dei.isep.ipp.pt, baa@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

In this paper we discuss challenges and design principles of an implementation of slot-based task- splitting algorithms into the Linux 2.6.34 version. We show that this kernel version is provided with the required features for implementing such scheduling algorithms. We show that the real behavior of the scheduling algorithm is very close to the theoretical. We run and discuss experiments on 4-core and 24-core machines.

On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems

Paulo Baltarejo Sousa
CISTER-ISEP Research Center, Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal
pbs@isep.ipp.pt

Konstantinos Bletsas
CISTER-ISEP Research Center, Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal
ksbs@isep.ipp.pt

Eduardo Tovar
CISTER-ISEP Research Center, Polytechnic Institute of Porto
Rua Dr. António Bernardino de Almeida 431, 4200-072 PORTO, Portugal
emt@isep.ipp.pt

Björn Andersson
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, USA
baandersson@sei.cmu.edu

Abstract

In this paper we discuss challenges and design principles of an implementation of slot-based task-splitting algorithms into the Linux 2.6.34 version. We show that this kernel version is provided with the required features for implementing such scheduling algorithms. We show that the real behavior of the scheduling algorithm is very close to the theoretical. We run and discuss experiments on 4-core and 24-core machines.

1 Introduction

Nowadays, multiprocessors implemented on a single chip (called *multicores*) are mainstream computing technology and it is expected that the number of cores per chip continue increasing. They may provide great computing capacity if appropriate scheduling algorithms are devised. Real-time scheduling algorithms for multiprocessors are categorized as: *global*, *partitioned* and *semi-partitioned*.

Global scheduling algorithms store tasks in one global queue, shared by all processors. Tasks can migrate from one processor to another; that is, a task can be preempted during its execution and resume its execution on another processor. At any moment the m (assuming m processors) highest-priority tasks are selected for execution on the m processors. Some algorithms of this category can achieve a utilization

bound of 100%, but generate too many preemptions.

Partitioned scheduling algorithms partition the task set and assign all tasks in a partition to the same processor. Hence, tasks cannot migrate between processors. Such algorithms involve few preemptions but their utilization bound is at most 50%.

Semi-partitioning (also known as *task-splitting*) scheduling algorithms assign most tasks (called *non-split tasks*) to just one processor but some tasks (called *split tasks*) are assigned to two or more processors. Uniprocessor dispatchers are then used on each processor but they are modified to ensure that a split task never executes on two or more processors simultaneously.

Several multiprocessor scheduling algorithms have been implemented and tested using vanilla Linux kernel. Litmus^{RT} [1] provides a modular framework for different scheduling algorithms (global-EDF, pfair algorithms). Kato *et al.* [2] created a modular framework, called RESCH, for using other algorithms than Litmus^{RT} (partitioned, semi-partitioned scheduling). Faggioli *et al.* [3] implemented global-EDF in the Linux kernel and made it compliant with POSIX interfaces.

In this paper we address the Real-time TAsk-Splitting scheduling algorithms (ReTAS) framework [11] that implements a specific type of semi-partitioned scheduling: slot-based task-splitting multiprocessor scheduling algorithms [4, 5, 6]. Slot-based task-splitting scheduling algorithms assign most tasks to just one processor and a few to only two processors. They subdivide the time into equal-duration *timeslots* and each timeslot processor is composed by one or more time *reserves*. These reserves are used to execute tasks. Reserves used for split tasks, which execute on two processors, must be carefully positioned to avoid overlapping in time.

The remainder of this paper is structured as follows. Section 2 provides a description of the main features of the slot-based task-splitting scheduling algorithms. Section 3 discusses some challenges and design principles to implement this kind of algorithms. A detailed description of our implementation is presented in Section 4 while in Section 5 we discuss the discrepancy between theory and practice. Finally, in Section 6 conclusions are drawn.

2 Slot-based task-splitting

Slot-based task-splitting algorithms have two important components: (i) the *task assigning*; and (ii) the *dispatching algorithm*. The task assigning algorithm executes prior runtime and besides assigning tasks to processors is also responsible for computing all parameters required by the dispatching algorithm. The dispatching algorithm works over the timeslot and selects tasks to be executed by processors.

The Sporadic-EKG (S-EKG) [4] extends the periodic task set model of EKG [7] to sporadic task set models. This approach assures that the number of split tasks is bounded (there are at most $m-1$ split tasks), each split task executes on only two processors and the non-split tasks execute on only one processor. The beginning and end of each timeslot are synchronized across all processors. The end of a timeslot of processor p contains a reserve and the beginning of a timeslot of processor $p+1$ contains another reserve, and these two reserves supply processing capacity for a split task. As non-split tasks execute only on one processor they are scheduled according to the uniprocessor EDF scheduling algorithm. A detailed description of that algorithm with an example can be found at [8].

While EKG versions are based on the task, the NPS-F [5,6] uses an approach based on bins. Each bin is assigned one or more tasks and there is a one to one relation between each bin and each notional processor. Then, the notional processor schedules tasks of each bin under the EDF scheduling policy. The time is split into equal-duration timeslots and each timeslot is composed by one or more time reserves. Each notional processor is assigned one reserve in one physical processor. However, up to $m-1$ notional processors could be assigned to two reserves, which means that these notional processors are implemented upon two physical processor reserves, while the remaining notional processors are implemented upon one physical processor reserve.

There is one fundamental difference between S-EKG and NPS-F algorithms. NPS-F can potentially generate a higher number of split tasks than S-EKG. Another difference is related to the dispatching algorithm. The S-EKG allows non-split tasks to be executed on the split task reserves (in the case when these tasks are not ready to be executed) while NPS-F does not; that is, each notional processor executes only on its reserve(s).

Fig. 1 shows a generic execution timeline produced by these scheduling algorithms. The time is divided into equal-duration timeslots of length S. Each timeslot is divided up to 3 reserves: $x[p]$, $y[p]$ and $N[p]$. Reserve $x[p]$ is located in the beginning of the timeslot and is reserved to execute the task or notional processor split between processors p and $p-1$. Reserve $y[p]$ is located in the end of the timeslot and

is reserved to execute the task or notional processor split between processors p and $p+1$. The remaining part ($N[p]$) of the timeslot is used to execute non-split tasks or notional processors that execute on only one processor.

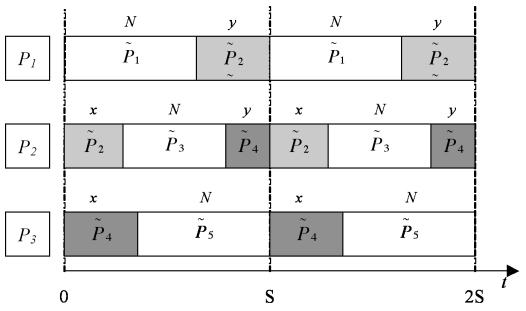

FIGURE 1: *Execution timeline example.*

In the remainder of this paper, we will discuss the implementation of S-EKG and NPS-F algorithms in 4-core and 24-core machines supported by the Linux 2.6.34 kernel version.

3 Challenges and design principles for implementing slot-based task-splitting

In [9] a set of challenges and a set of design principles for the S-EKG implementation were discussed. However, in this paper we will implement NPS-F as well and for this reason we will need to adapt the design principles. In this section, we will do so as follows: each processor should have its own runqueue (the queue that stores ready jobs). The runqueue of each processor should map the ready tasks with its reserves; that is, which ready tasks are allowed to execute on each reserve. Since some tasks may execute on two processors, what is the best approach for that? If tasks are stored locally on each processor, whenever a task migrates from one processor to another processor, it requires locking both processor runqueues for moving that task from one runqueue to the other runqueue. However, in the case of the NPS-F this could imply moving more than one task. Since the frequency of migration may be high, it turns out that this is not the best approach; so we adopted a different approach. We defined a runqueue per notional processor so each notional processor stores all ready tasks assigned to it. Then, we map each notional processor to processor reserves.

As it is intuitive from observing two consecutive timeslots in Fig. 1, whenever a split task consumes its reserve on processor p, it has to immediately resume execution on its reserve on processor $p+1$. Due to many sources of unpredictability (e.g. interrupts) in a real operating system, this precision is not possible. Consequently, this can prevent the dispatcher of processor $p+1$ to select the split task because processor p has not yet relinquished that task. In order to handle this issue, one option could be that processor $p+1$ sends an inter-processor interrupt (IPI) to processor p to relinquish the split task, and another could be that processor $p+1$ sets up timer x time units in future to force the invocation of its dispatcher. Two reasons have forced us to choose the latter. First, we know that if a dispatcher has not yet relinquished the split task it was because something is preventing it from doing so, such as, the execution of an interrupt service routine (ISR). Second, the use of IPIs will create some dependency between processors that could embarrass the scalability of the dispatcher.

4 Implementation of slot-based task-splitting

4.1 Assumptions about the architecture

We assume *identical processors*, which means that (i) all processors have the same instruction set and data layout (e.g. big-endian/little-endian) and (ii) all processors execute at the same speed.

We also assume that the *execution speed* of a processor does not depend on activities on another processor (for example whether the other processor is busy or idle or which task it is busy executing) and also does not change at runtime. In practice, this implies that (i) if the system supports simultaneous multithreading (Intel calls it *hyperthreading*) then this feature must be disabled and (ii) features that allow processors to change their speed (for example power and thermal management) must be disabled.

We assume that each processor has a *local timer* providing two functions. One function allows reading the current real-time (that is not calendar time) as an integer. Another function makes it possible to set up the timer to generate an interrupt at x time units in the future, where x can be specified.

4.2 Why vanilla Linux kernel?

The vanilla Linux kernel 2.6.34 was chosen to implement the scheduling algorithms S-EKG [4] and NPS-F [5,6]. That kernel version provides the required mechanisms to satisfy the previously mentioned design principles: (i) each processor holds its own runqueue and it is easy to add new fields to it; (ii) it has already implemented red-black trees that are balanced binary trees whose nodes are sorted by a key and most the operations are done in $O(\log n)$ time; (iii) it has the high resolution timers infrastructure that offers a nanosecond time unit resolution, and timers can be set on a per-processor basis; (iv) it is very simple to add new system calls and, finally, (v) it comes with the modular scheduling infrastructure that easily enables adding a new scheduling policy to the kernel.

4.3 ReTAS implementation

The vanilla Linux kernel 2.6.34 has three native scheduling modules: RT (Real-Time); CFS (Completely Fair Scheduling) and Idle. Those modules are hierarchically organized by priority in a linked list; the module with highest priority is the RT, the one with the lowest is the Idle module. Starting with the highest priority module, the dispatcher looks for a runnable task of each module in a decreasing order of priority.

We added a new scheduling policy module, called ReTAS, on top of the native Linux module hierarchy, thus becoming the highest priority module. That module implements the S-EKG and NPS-F scheduling algorithms. The ReTAS implementation consists on a set of modifications to the Linux 2.6.34 kernel in order to support the S-EKG and NPS-F scheduling algorithms and also the cluster version of the NPS-F, called C-NPS-F [5]. These scheduling policies are identified by the `SCHED_S_EKG` and `SCHED_NPS_F` macros.

Since the assigning algorithm is executed prior to runtime, in the next sections we will focus only on the kernel implementation; that is, on the dispatching algorithms.

4.3.1 ReTAS tasks

To differentiate these tasks from other tasks present in the system, we refer to these tasks as ReTAS tasks. Listing 1 shows the pseudo-algorithm of ReTAS tasks. They are periodic tasks and are always present in the system. Each loop iteraction is considered a job. Note that the first job of each task appears in the system at `time0` + `offset` (`time0` is set equal to all tasks in the system) and the remaining jobs are activated according to the `period`. The `delay_until` function sleeps a task until the absolute time specified by `arrival`.

```
arrival:=time0 + offset;
while(true)
{
 delay_until(arrival);
 execute();
 arrival:=arrival + period;
}
```

Listing 1: ReTAS task pseudo-algorithm.

In the Linux operating system a *process* is an instance of a program in execution. To manage all processes, the kernel uses an instance of `struct task_struct` data structure for each process. In order to manage ReTAS tasks, some fields were added to the `struct task_struct` data structure (see Listing 2). `notional_cpu_id` field is used to associate the task with the notional processor. Fields `cpu1` and `cpu2` are used to set the logical identifier of processor(s) in which the task will be executed on. The absolute deadline and also the arrival of each job are set on the `deadline` and `arrival` fields of the `retas_job_param` data structure, respectively.

```
struct retas_task {
  int notional_cpu_id;
  struct retas_task_param task_param{
   unsigned long long deadline; //D_i
  }task_param;
  struct retas_job_param job_param{
   unsigned long long deadline; //d_ij
   unsigned long long arrival; //a_ij
  }job_param;
  int cpu1;
  int cpu2;
  ...
};

struct task_struct {
 ...
 struct retas_task retas_task;
};
```

Listing 2: Fields added to `struct task_struct` kernel data structure.

4.3.2 Notional processors

As mentioned before, ReTAS tasks are assigned to notional processors. Therefore, notional processors act as a runqueue. Each notional processor is an instance of `struct notional_cpu` data structure (see Listing 3), which is identified by a numerical identifier (`id`). Field `cpu` is set with the logical identifier of the physical processor that, in a specific time instant, is executing a task from that notional processor. The purpose of the `flag` will be explained in

Section 4.3.5. Each notional processor organizes all ready jobs in a red-black tree, whose root is the field `root_tasks`, according to the job absolute deadline. The `lock` field is used to serialize the insertion and remotion operations over the red-black tree specially for notional processors that are executed by two processors. `edf` field points to the task with the earliest deadline stored in the red-black tree. Note that `notional_cpus` is a vector defined as global variable.

```
struct notional_cpu{
 int id;
 atomic_t cpu;
 atomic_t flag;
 raw_spinlock_t lock;
 struct rb_root root_tasks;
 struct task_struct *edf;
 ...
};
...

struct notional_cpu notional_cpus[
    NR_NOTIONAL_CPUS];
```

Listing 3: `struct notional_cpu` data structure.

4.3.3 Timeslot reserves

Each processor needs to know the composition of its timeslot. So, per-processor an instance of the `struct timeslot` (Listing 4) is defined. Fields `timeslot_length`, `begin_curr_timeslot`, `reserve_length` and `timer` are used to set the time division into time reserves. They are also used to identify in each time reserve a given time instant `t` falls in. When a timer expires, the timer callback sets the current task to be preempted and this automatically triggers the invocation of the dispacther. And taking into account the current reserve, the dispacther (that will be described on the next section) tries to pick a task from either the notional processor pointed by `notional_cpu` (for the first option) or notional processor pointed by `alt_notional_cpu` (for the second option).

```
struct timeslot_reserve{
 struct notional_cpu *notional_cpu;   //first
     option
 struct notional_cpu *alt_notional_cpu;  //
     second option
 unsigned long long reserve_length;
};

struct timeslot{
 unsigned long long timeslot_length;
 unsigned long long begin_curr_timeslot;
 struct timeslot_reserve notional_cpus[
     NR_NCPUS_PER_CPU];
 ...
 struct hrtimer timer;
};
```

Listing 4: `struct timeslot` data structure.

4.3.4 ReTAS scheduling module

In the vanilla Linux kernel each processor holds a runqueue of all runnable tasks assigned to it. The scheduling algorithm uses this runqueue to select the "best" process to be executed. The information for these processes is stored in a per-processor data structure called `struct rq` (Listing 5). Many functions that compose the Linux's modular scheduling framework have an instance of this data structure as argument. Listing 5 shows the new data structures required by the ReTAS scheduling module added to the Linux native `struct rq`. The purpose of the `struct timeslot` data structure was described in the previous section.

```
struct retas_rq {
 int post_schedule;
 struct timeslot timeslot;
 struct release release;
 struct resched_cpu resched_cpu;
};

struct rq {
 ...
 struct retas_rq retas_rq;
};
```

Listing 5: `struct retas_rq` added to `struct rq`.

According to the rules of the Linux's modular scheduling framework, each module must implement a set of functions specified in the `sched_class` data structure. Listing 6 shows the definition of `retas_sched_class`, which implements the ReTAS module. The first field (`next`), is a pointer to the next `sched_class` in the hierarchy. Since `retas_sched_class` is declared as the highest priority scheduler module that field points to the `rt_sched_class`, which implements the two POSIX real-time policies (`SCHED_FIFO` and `SCHED_RR`). The other fields are functions that act as callbacks to specific events.

```
static const struct sched_class
    retas_sched_class = {
 .next     = &rt_sched_class,
 .enqueue_task  = enqueue_task_retas,
 .dequeue_task  = dequeue_task_retas,
 .check_preempt_curr = check_preempt_curr_retas
     ,
 .pick_next_task  = pick_next_task_retas,
 ...
};
```

Listing 6: `retas_sched_class` scheduling class.

The `enqueue_task_retas` (Listing 7) is called whenever a ReTAS job enters into a runnable state. It receives two pointers, one for the runqueue of the processor that is running this code (`rq`) and another to the task that is entering in a runnable state (`p`). Then, it updates the job absolute `deadline` by suming the job `arrival` time (this field is updated

through the job release procedure that will be described Section 4.3.6) to the task relative `deadline`, and inserts it into the red-black tree of its notional processor. Additionally, it checks if this job (in the case of being a split task) could be executed by other processor; that is, if it is a split task could happen that when a job is released on this processor could correspond to its reserve on the other processor. If that is the case, then an IPI is sent to the other processor, using the `resched_cpu` function.

```
static void
enqueue_task_retas(struct rq *rq, struct
    task_struct *p, int wakeup,bool flag)
{
 int cpu;
 p->retas_task.job_param.deadline=p->retas_task
     .job_param.arrival+p->retas_task.
     task_param.deadline;
 insert_task(&notional_cpus[p->retas_task.
     notional_cpu_id], p);
 cpu=check_for_running_on_other_cpus(p, rq->cpu
     );
 if(cpu!=-1){
  resched_cpu(cpu);
 }
 return;
}
```

Listing 7: `enqueue_task_retas` function.

When a ReTAS job is no longer runnable, then the `dequeue_task_retas` function is called that undoes the work of the `enqueue_task_retas` function (see Listing 8); that is, it removes the task from the notional processor.

```
static void
dequeue_task_retas(struct rq *rq, struct
    task_struct *p, int sleep)
{
 remove_task(&notional_cpus[p->retas_task.
     notional_cpu_id], p);
 return;
}
```

Listing 8: `dequeue_task_retas` function.

As the name suggests, the `check_preempt_curr_retas` function (Listing 9) checks whether the currently running task must be preempted or not. This function is called following the enqueuing or dequeuing of a task and it checks if there is any ReTAS jobs to be executed. If so, it checks if that job is available; that is, if it is not being executed by another processor (to handle this issue, we use the `atomic_t cpu` field defined on the `struct notional_cpu`) it sets a flag that indicates to the dispatcher that the currently running task must be preempted, otherwise it sets up a local timer (defined in the `struct resched_cpu resched_cpu`) to expire some time later (throught `set_resched_cpu_timer_expires`, which will trigger, at timer expiration, the invocation of the dispatcher).

```
static void
check_preempt_curr_retas(struct rq *rq, struct
    task_struct *p, int sync)
{
 struct task_struct *t=NULL;
 int cpu;
 t=get_edf_task(get_current_notional_cpu(&rq->
     retas_rq.timeslot));
 if(!t){
  t=get_edf_task(get_current_alt_notional_cpu(&
      rq->retas_rq.timeslot));
 }
 if(t){
  if(t!=rq->curr){
   cpu=is_executing_on_other_cpu(t->retas_task.
       notional_cpu_id, rq->cpu);
   if(cpu==-1){
    set_tsk_need_resched(rq->curr);
   }
   else{
    set_resched_cpu_timer_expires(rq);
   }
  }
 }
}
```

Listing 9: `check_preempt_curr_retas` function.

The `pick_next_task_retas` function selects the job to be executed by the current processor (see Listing 10). This function is called by the dispatcher whenever the currently running task is marked to be preempted or when a task finishes its execution. First, it tries to get highest priority ReTAS job from the first notional processor and, if there is no job it checks the second notional processor. If there is one ReTAS job ready to be executed (and it is not the current executing job) then, the next step is to lock the notional processor to that processor (this is done in the `lock_notional_cpu` function and this locking mechanism will be described in Section 4.3.5). If this notional processor is locked it sets up a local timer to expire some time later and returns `NULL`, otherwise it returns the pointer to that job.

```
static struct task_struct *
pick_next_task_retas(struct rq *rq)
{

 struct task_struct *t=NULL;
 int cpu;
 t=get_edf_task(get_current_notional_cpu(&rq->
     retas_rq.timeslot));
 if(!t){//it is assumed that these tasks (of
     alt_notional_cpu) execute only on this cpu
  t=get_edf_task(get_current_alt_notional_cpu(&
      rq->retas_rq.timeslot));
 }
 if(t){
  cpu=lock_notional_cpu(t->retas_task.
      notional_cpu_id, rq->cpu);
  if(cpu!=-1){
   set_resched_cpu_timer_expires(rq,t->
       retas_task.notional_cpu_id);
   t=NULL;
   goto pick_ret;
  }
 }
pick_ret:
 return t;
}
```

Listing 10: `pick_next_task_retas` function.

4.3.5 Locking a notional processor

Whenever there is a shared resource there is the need to create a synchronization mechanism to serialize the access to that resource. In this implementation a notional processor can be shared by up two physical processors. So, to serialize the access to those notional processors two functions are used (see Listing 11): `lock_notional_cpu` and `unlock_notional_cpu`. The locking is done in the `pick_next_task_retas` function. As it can be seen from Listing 11, first it identifies the physical processor that is executing tasks from that notional processor. Next, it tries to add one to the `atomic_t flag` variable using the `atomic_add_unless` kernel function. But this operation only succeds if the value is not one; that is, if it is zero, otherwise, it fails. In the first case, success, it atomically adds one to the `flag` and sets the `cpu` variable with the logical identifier of the current processor. And this way it locks the notional processor to that processor. In the second case, failure, the notional processor could be locked by other processor or by the current processor. If it is locked by the current processor nothing changes, otherwise the logical number of the processor is returned and the dispatcher cannot pick any job from that notional processor.

In order to unlock the notional processor, whenever a ReTAS job is the `prev` or the `next` task in the context of the `context_swicth` function, which is invocated by `schedule` function (see Listing 13), it will enforce the execution of the `post_schedule_retas` function (see Listing 12) to unlock the notional processor. Unlocking means setting the `flag` variable equal to zero and the `cpu` variable equal to -1.

```
int lock_notional_cpu(int ncpu, int cpu)
{
 int ret=atomic_read(&notional_cpus[ncpu].cpu);
 if(atomic_add_unless(&notional_cpus[ncpu].flag
     ,1,1)){
  atomic_set(&notional_cpus[ncpu].cpu,cpu);
  ret=cpu;
 }
 if(ret==cpu)
  ret=-1;
 return ret;
}

void unlock_notional_cpu(int ncpu, int cpu)
{
 int x;
 x=atomic_read(&notional_cpus[ncpu].cpu);
 if(x==cpu){
  atomic_set(&notional_cpus[ncpu].cpu,-1);
  atomic_set(&notional_cpus[ncpu].flag,0);
 }
}
```

Listing 11: Lock and unlock notional processor functions.

```
void
post_schedule_retas(struct rq *rq)
{
 int i, ncpu=-1;
 if(rq->retas_rq.post_schedule){
  if(rq->curr->policy==SCHED_S_EKG || rq->curr
      ->policy==SCHED_NPS_F){
   ncpu=rq->curr->retas_task.notional_cpu_id;
  }
  for(i=0;i<rq->retas_rq.timeslot.
      nr_notional_cpus;i++){
   if(likely(rq->retas_rq.timeslot.
       notional_cpus[i].notional_cpu)){
    if(rq->retas_rq.timeslot.notional_cpus[i].
        notional_cpu->id!=ncpu){
     unlock_notional_cpu(rq->retas_rq.timeslot.
         notional_cpus[i].notional_cpu->id,rq->
         cpu);
    }
   }
  }
 }
 rq->retas_rq.post_schedule = 0;
}
```

Listing 12: `post_schedule_retas` function.

```
static inline void
context_switch(struct rq *rq, struct
    task_struct *prev,
        struct task_struct *next)
{
 ...
 if(prev->policy==SCHED_S_EKG ||  prev->policy
     ==SCHED_NPS_F ||
  next->policy==SCHED_S_EKG ||  next->policy==
      SCHED_NPS_F)
  rq->retas_rq.post_schedule = 1;
 ...
 switch_to(prev, next, prev);
 ...
}

asmlinkage void
__sched schedule(void)
{
 struct task_struct *prev, *next;
 struct rq *rq;
 ...
 put_prev_task(rq, prev);
 next = pick_next_task(rq);
   ...
  context_switch(rq, prev, next); /* unlocks
      the rq */
 ...
 post_schedule_retas(rq);
 ...
}
```

Listing 13: `context_swicth` and `schedule` functions.

4.3.6 Job release mechanism

The job release mechanism is supported by the `struct release` and is set per-processor. It is composed by a red-black tree and a timer. The idea is the following: a job is put in the waiting state, next, it is inserted into a red-black tree ordered by the absolute arrival time and, finally, a timer is set to expire at the earliest arrival time of all jobs stored into the red-black tree. When the timer expires, the job with earliest arrival time is removed from the red-black tree and its state changes to running, consequently,

it becomes ready. The timer is armed to expire at the earliest arrival time of remaining jobs stored into the red-black tree. This procedure is triggered by the `delay_until` system call that specifies the absolute arrival time of a job. One feature of this mechanism is that the next release of a job is done on the processor where it finishes its execution; that is, where it executes the `delay_until` system call.

5 From theory to practice

Usually, real-time scheduling algorithms for multiprocessor systems are supported by a set of assumptions that have no correspondence in the real-world. The evaluation of the discrepancy between theory and practice is here addressed taking into account two real-world phenomena: *jitter* and *overhead*. For convenience we define jitter as being the time from when an event must occur until it actually occurs. We have identified three sources of jitter: *reserve*, *release* and *context switch*. We also define overhead as the time that the current executing task is blocked by something else not related to the scheduling algorithm. We have identified two sources for the overhead: *interrupts* and *release*.

5.1 Sources of jitter

Theoretically, when a reserve ends one assumes that jobs are instantaneously switched, but in practice this is not true. Fig. 2 shows $ResJ_{i,k}$, which represents the measured *reserve jitter* of job $\tau_{i,k}$ and denotes the discrepancy between the time when the job $\tau_{i,k}$ should (re)start executing (at the beginning of the reserve A, where A could be x, N or y) and when it actually (re)starts. It should be mentioned that the timers are set up to fire when the reserve should begin, but, unfortunately, there is always a drift between this time and the time instant at which the timer fires. Then, the timer callback executes and, in most cases, sets the current task to be preempted triggering the invocation of the dispatcher. The dispatcher selects a task according to the dispatching algorithm and switches the current task with the selected task.

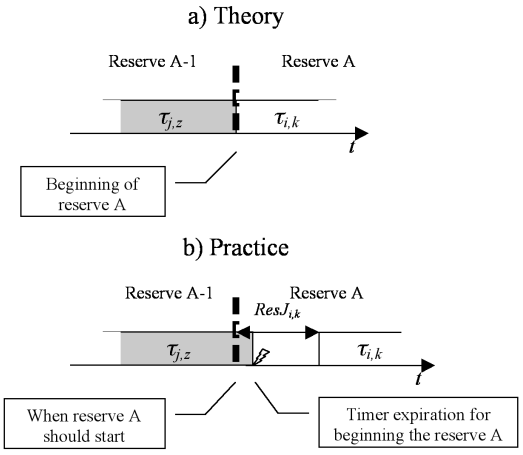

FIGURE 2: *Reserve jitter.*

In theory it is typically assumed that a release of a job is instantaneous and becomes ready immediately. In practice, however, something very different is observed. $RelJ_{i,k}$ (*release jitter* of job $\tau_{i,k}$) denotes the difference in time from when the job $\tau_{i,k}$ should arrive until it is inserted in the ready queue (see Fig. 3). The following steps take place in practice. First, a timer expires to wake up the job and, as mentioned before, there is always a drift between the time instant when the timer should expire and when it actually expires. Next, the job is removed from the release queue and inserted into the ready queue.

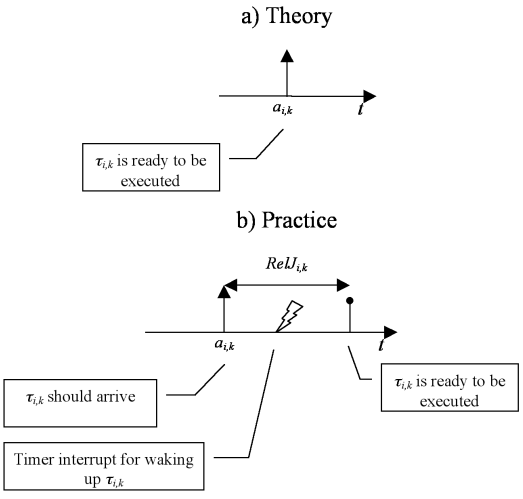

FIGURE 3: *Release jitter.*

A final important source of jitter is the context switching. A context switch is the procedure for switching processor from one job to another.

In theory, the time required to switch jobs is usually neglected. However, switching from one process to another requires a certain amount of time for saving and loading registers, among other operations. $CtswJ_{i,k}$ (*context switch jitter* of job $\tau_{i,k}$) denotes the difference in time from when the job $\tau_{i,k}$ should start executing until it actually (re)starts (see Fig. 4). Note that, we consider the context switch time incurred by the EDF scheduling policy, because the context switch time incurred by reserves are accounted for by $ResJ$.

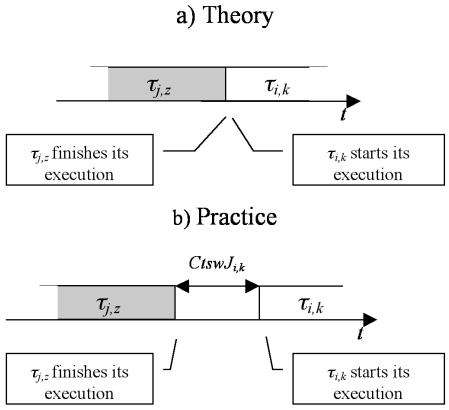

FIGURE 4: *Context switch jitter.*

5.2 Sources of overhead

Usually, in theory, we ignore interrupts, but in practice they are one of main sources of timing unpredictability in this kind of system. In practice, when an interrupt arises the processor suspends the execution of the current job in order to execute the associated ISR. $IntO_{i,k}$ (*interrupt overhead* of job $\tau_{i,k}$) denotes the time during which job $\tau_{i,k}$ is prevented from executing due to the execution of an ISR (see Fig. 5).

When a job is executing on a processor, jobs of other tasks could appear in the system. In our implementation, to release a job, the processor stops what it is doing to release that job. $RelO_{i,k}$ (*release overhead* of job $\tau_{i,k}$) denotes the time during which job $\tau_{i,k}$ is prevented from executing due to the releases of other jobs (see Fig. 6).

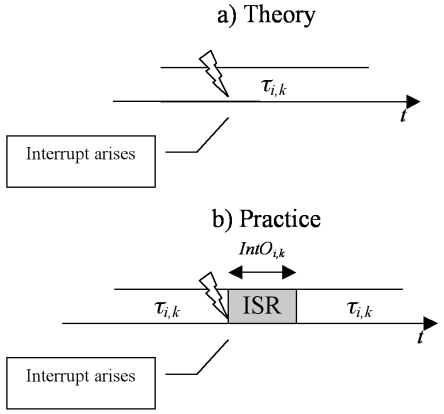

FIGURE 5: *Interrupt overhead.*

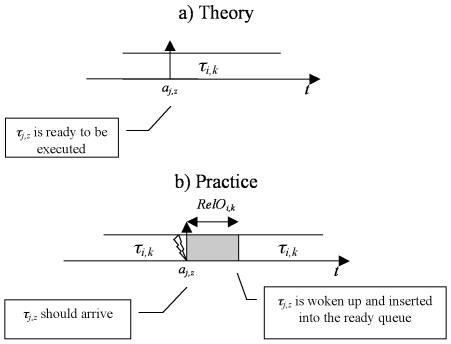

FIGURE 6: *Release overhead.*

5.3 Experimental setup

In order to evaluate the discrepancy between theory and practice, we have conducted a range of experiments with 4-core (Intel(R) Core(TM) i7 CPU @ 2.67GHz) and 24-core (AMD Opteron (TM) Processor 6168 @ 1.90GHz) machines with real-time tasks executing empty for-loops. In order to make the environment more controlled, we (i) set runlevel to 1 (that is no windowing system), (ii) disabled network interface and also the filesystem journal mechanism, (iii) all interrupts are handled by the last processor and (iv) setup one non real-time task per core, as a background task, to ensure that the kernel idle threads never start executing.

We generated 17 random task sets, which utilization target varied from 0.88 and 0.885. The period and utilization of tasks varied from 5 ms up to 50 ms and from 0.1 up to 1.0, respectively. The number of tasks varied from 6 up to 28 tasks in the case of the 4-core machine. The time duration of each experiment was approximately 1000 s. All task sets were

scheduled using the S-EKG and NPS-F scheduling algorithms (which have comparable jitter/overheads). Since each experiment took 1000 s, the whole set of experiments took 34000 s.

5.4 Discussion of results

We collected the maximum values observed for each type of jitter and also for each type of overhead irrespective of the algorithm used (S-EKG or NPS-F). Table 1 presents the experimental results for: reserve jitter ($ResJ$), release jitter ($RelJ$), context switch jitter ($CtswJ$), the overhead of interrupt 20 (related to the hard disk) and the overhead of tick. Note that, we do not directly present the release overhead. Rather, since, the release overhead is part of what is experienced as release jitter, we simply present the worst-case $RelJ$ (which also accounts for $RelO$). The column identified with MAX_{max} gives the maximum value observed in all experiments. The third column (AVG_{τ_i}) gives the average value experimented by the task that experienced the MAX_{max} value. The fourth column (MIN_{max}) gives the minimum of the collected values (note that is the minimum of the maximum values). The last column displays the average value of the task that experienced the MIN_{max} value. Before analyzing the results, we draw the attention of the reader to the time unit, μs, which means that the impact of those jitters/overheads is relatively small. Recall that the period of tasks in the various experiments varied from 5 ms up to 50 ms.

The highest $ResJ$ values were constantly experienced by split tasks. This is due to the task migration mechanism required for split tasks (described in Section 4). In that mechanism, if a task is not available, a timer is set to expire some time later. The value chosen for this delay was 5 μs.

The MAX_{max} $RelJ$ value is too high (31.834 μs), but comparing both AVG_{τ_i} (0.329 μs and 0.369 μs) observed shows that something prevented the release mechanism of doing that job release. The reason for this is related to the unprectability of the underlying operating system. There are many sources of unpredictability in a Linux kernel: (i) interrupts are the events with the highest priority, consequently when one arises, the processor execution switches to handle the interrupt (usually interrupts arise in an unpredictable fashion); (ii) on Symmetric Multi Processing (SMP) systems there are multiple kernel threads running on different processors in parallel, and those can simultaneously operate on shared kernel data structures requiring serialization on access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions; (iv) the high resolution timer infrastructure is based on local Advanced Programmable Interrupt Controller (APIC), disabling and enabling local interrupts can disrupt the precision of that timer and, finally, (v) the hardware that Linux typically runs on does not provide the necessary determinism, which would permit the timing behavior of the system to be predictable with all latencies being time-bounded and known prior to run-time.

The same reasons could be given to explain the MAX_{max} $CtswJ$ value. However, usually the magnitude of $CtswJ$ is not too high, because this operation is done by the scheduler, which executes in a controlled context.

In Table 1, we present the overhead results of two ISRs: $irq20$ and $tick$ ($tick$ is a periodic timer interrupt used by the system to do a set of operations,like for instance invoking the scheduler). The reason for this is, $irq20$ can be configured to be executed by one specific processor but $tick$ cannot. In our opinion, it does not make sense to present other values besides MAX_{max}, because in these experiments this is a sporadic and rare event. In contrast, $tick$ is periodic with a frequency of approximately 1 ms. The values observed show that this overhead is very small.

	MAX_{max} (μs)	AVG_{τ_i} (μs)	MIN_{max} (μs)	AVG_{τ_i} (μs)
$ResJ$	8.824	5.255	7.919	5.833
$RelJ$	31.834	0.329	10.029	0.369
$CtswJ$	2.218	0.424	0.587	0.305
$IntO$ - irq20	24.226	-	-	-
$IntO$ - tick	0.571	0.272	0.408	0.243

TABLE 1: *Experimental results (4-core machine).*

	MAX_{max} (μs)	AVG_{τ_i} (μs)	MIN_{max} (μs)	AVG_{τ_i} (μs)
ResJ	48.087	8.493	19.243	9.498
RelJ	37.264	0.731	11.609	0.848

TABLE 2: *Experimental results (24-core machine).*

Because of space limitations, in this paper our analysis is focused on 4-core machine results, in [10] some results of a set of experiments with the 24-core machine are presented. Nevertheless, Table 2 shows some results related to the *ResJ* and *RelJ* on the 24-core machine. The explanation for the MAX_{max} values is the same that was given for the 4-core machine results. The AVG_{τ_i} values are in all cases higher than those for the 4-core machines. This is due to the speed of the processors: 4-core processors operate at 2.67GHz while 24-core processors operate at 1.9GHz.

6 Conclusions

We have presented the Real-time TAsk-Splitting scheduling algorithms (ReTAS) framework that implements S-EKG and NPS-F slot-based task-splitting scheduling algorithms. The main purpose of this framework is to show that slot-based task-splitting scheduling algorithms can be implemented in a real-operating system (using the vanilla Linux kernel) and work in practice. Using this framework we have identified and measured the real-operating system jitters and overheads. In spite of the unpredictability of the Linux kernel we observed a good correspondence between theory and practice. These good results are due to: (i) the controlled experimental environment; (ii) the use of the local high-resolution timers and (iii) the fact that these scheduling algorithms only involve very limited synchronization on shared system resources between processors.

Acknowledgements

This work was supported by the CISTER Research Unit (608 FCT) and also by the REHEAT project, ref. FCOMP-01-0124-FEDER-010045 funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness) and by National Funds (PT), through the FCT - Portuguese Foundation for Science and Technology.

References

[1] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, *LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers*, in proceedings of 27th IEEE Real-Time Systems Symposium (RTSS 06), Rio de Janeiro, Brazil, pp. 111–126, 2006.

[2] S. Kato and R. Rajkumar and Y. Ishikawa, *A Loadable Real-Time Scheduler Suite for Multicore Platforms*, in Technical Report CMU-ECE-TR09-12, 2009. Available online:http://www.ece.cmu.edu/~shinpei/papers/techrep09.pdf.

[3] D. Faggioli and M. Trimarchi and F. Checconi and C. Scordino, *An EDF scheduling class for the Linux kernel*, in proceedings of 11th Real-Time Linux Workshop (RTLWS 09), Dresden, Germany, pp. 197–204, 2009.

[4] B. Andersson and K. Bletsas, *Sporadic Multiprocessor Scheduling with Few Preemptions*, in proceedings of 20th Euromicro Conference on Real-Time Systems (ECRTS 08), Prague, Czech Republic, pp. 243–252, 2008.

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

[6] K. Bletsas and B. Andersson, *Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound*, in proceedings of 30th IEEE Real-Time Systems Symposium (RTSS 09), Washington, DC, USA, pp. 385–394, 2009.

[7] B. Andersson and E. Tovar, *Multiprocessor Scheduling with Few Preemption*, in proceedings of 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA 06), Sydney, Australia, pp. 322–334, 2006.

[8] P. B. Sousa and B. Andersson and E. Tovar, *Implementing Slot-Based Task-Splitting Multiprocessor Scheduling*, in proceedings of 6th IEEE International Symposium on Industrial Embedded Systems (SIES 11), Västerås, Sweden, pp. 256–265, 2011.

[9] P. B. Sousa and B. Andersson and E. Tovar, *Challenges and Design Principles for Implementing Slot-Based Task-Splitting Multiprocessor Scheduling*, in Work in Progress (WiP) session of the 31st IEEE Real-Time Systems Symposium (RTSS 10), San Diego, CA, USA, 2010. Available online: http://cse.unl.edu/rtss2008/archive/rtss2010/WIP2010/5.pdf

[10] P. B. Sousa K. Bletsas and E. Tovar and B. Andersson, *On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems*, (extended version of this paper) in Technical Report HURRAY-TR-110903, 2011.

[11] P. B. Sousa, *ReTAS*. Available online: http://webpages.cister.isep.ipp.pt/~pbsousa/retas/.

A Task set generator

Before explaining the task set generator, we describe the algorithm of the task used to run the experiments. We classified these tasks as real-time tasks executing empty for-loops. Listing 14 shows part of the task code. Each task receives a set of arguments (see Table 3). As their names suggest some of them are the minimum and the maximum of some task parameter. These parameters are used by `get_time_value` function to generate time values between `min` and `max`, like `min_offset_time` and `max_offset_time` The idea behind this approach is to have a flexible task implementation that allows the use fixed and also randomly (using uniform distribution) generated parameters.

Argument	Description
`task_id`	The logical identifier of the task
`release_time`	This variable is set with the time zero of the experiment
`min_exec_time`	The minimum execution time of the task
`max_exec_time`	The maximum execution time of the task
`min_inter_arrival_time`	The minimum inter arrival time of the task
`max_inter_arrival_time`	The maximum inter arrival time of the task
`deadline`	The relative deadline of the task
`min_offset_time`	The minimum offset time of the first task release
`max_offset_time`	The maximum offset time of the first task release
`nr_jobs`	The number of jobs of the task
`seed`	Seed for random generator

TABLE 3: *Task arguments.*

```
unsigned long long inline get_time_value(unsigned long long min, unsigned long long max)
{
 if(min==max)
  return min;
 return (min + ((1.0*rand()/RAND_MAX)*(max-min)));
}
void inline do_work(unsigned long long *min,unsigned long long *max)
{
 unsigned long long i,ten_ns;
 ten_ns=get_time_value(*min,*max)/10;
 for(i=0; i<ten_ns; i++){
   asm volatile ("nop" ::);
   ...
   asm volatile ("nop" ::);
 }
}

int main(int argc, char** argv)
{
 unsigned long long nj=0;
 unsigned long h=0,l=0;
 int task_id=atoi(argv[1]);
 unsigned long long release_time=(unsigned long long)atoll(argv[2]);
 unsigned long long min_exec_time=(unsigned long long)atoll(argv[3]);
 unsigned long long max_exec_time=(unsigned long long)atoll(argv[4]);
 unsigned long long min_inter_arrival_time=(unsigned long long)atoll(argv[5]);
 unsigned long long max_inter_arrival_time=(unsigned long long)atoll(argv[6]);
 unsigned long long deadline=(unsigned long long)atoll(argv[7]);
 unsigned long long min_offset_time=(unsigned long long)atoll(argv[8]);
 unsigned long long max_offset_time=(unsigned long long)atoll(argv[9]);
 unsigned long long nr_jobs=atoll(argv[10]);
 unsigned long long seed=(unsigned long long)atoll(argv[11]);
 srand(seed);
 release_time = release_time + get_time_value(min_offset_time,max_offset_time);
 nj=0;
 while(nj<nr_jobs){
  l=(unsigned long)release_time;
  h=release_time>>32;
  syscall(__NR_retas_delay_until,h,l);
  do_work(&min_exec_time,&max_exec_time);
  release_time = release_time + get_time_value(min_inter_arrival_time,max_inter_arrival_time);
  nj++;
 }
 return 0;
```

```
}
```

Listing 14: ReTAS task code.

Listing 16 shows part of the task set generator code. It receives a set of input parameters (see Table 4) and outputs a text file (see Listing 15). The procedure used to generate task sets is very simple. First, it iteratively generates the utilization of each task, using the uniform distribution with `UMIN` and `UMAX` as parameters. Then, it sums the utilization of all tasks (`SumUi`) generated until that iteration. If `SumUi` fall in the range between `UTARGMIN` and `UTARGMAX` then the procedure finishes, but if `SumUi` exceeds `UTARGMAX`, it discards all generated values and restarts the procedure. Second, it generates the minimal interarrival time of each task. Finally, it writes to the `file_name` file the task set parameters.

Generally C_i is computed multiply U_i by T_i ($C_i = T_i * U_i$), however, three parameters are set using `min_*` and `max_*`. Let us show how to compute them: the `min_inter_arrival_time` is set with the value generated, but the `max_inter_arrival_time` is computed by multiplying `min_inter_arrival_time` by the `factor`; the `max_exec_time` is computed by mulpitlying `min_inter_arrival_time` by U_i, but the `min_exec_time` is computed dividing `max_exec_time` by the `factor`; the `mim_offset_time` is set with the number of tasks multiplying by one second and the `max_offset_time` is computed multiplying `max_offset_time` by the `factor`. Note that, the `factor` has to be greater or equal to 1.

Listing 15 shows a task set example generated. The time unit is *ns* and the format of the file is the following: It is composed by 8 columns separated by commas and the mapping between file columns and task arguments presented in Table 3 is done by the following order: `task_id`, `min_exec_time`, `max_exec_time`, `min_inter_arrival_time`, `max_inter_arrival_time`, `deadline`,`min_offset_time` and `max_offset_time`. Since the `min_*` and `max_*` of task parameters are equal to each other, this means that the `factor` value was set equal to 1, otherwise they will be different.

Parameter	Description
`m`	Number of processors
`UTARGMIN`	The minimum utilization target of each processor
`UTARGMAX`	The maximum utilization target of each processor
`TMIN`	The minimal interarrival time of tasks
`TMAX`	The maximum interarrival time of tasks
`UMIN`	The minimal utilization of tasks
`UMAX`	The maximum utilization of tasks
`Factor`	Factor to determine the range of each parameter
`file_name`	The output file name
`seed`	Seed for random generator

TABLE 4: *Input parameters.*

```
1,2049267,2049267,15989152,15989152,15989152,2500000000,2500000000,
2,4696466,4696466,20601285,20601285,20601285,2500000000,2500000000,
3,1704351,1704351,15847839,15847839,15847839,2500000000,2500000000,
4,4304547,4304547,21097998,21097998,21097998,2500000000,2500000000,
5,1369002,1369002,6049432,6049432,6049432,2500000000,2500000000,
...
```

Listing 15: Task set generator output file.

```
void get_ui(int m, double UTARGMIN,double UTARGMAX,double UMIN,double UMAX,double *Ui, int *
    nr_tasks)
{
 int i, n=0,flag=1;
 double SumUi=0.0, factor,u;
 *nr_tasks=0;
 while(flag){
  n=0;
  SumUi=0.0;
  for(i=0;i<NR_TASKS && flag;i++){
   factor=(double)rand()/RAND_MAX;
   u=UMIN+factor*(UMAX-UMIN);
   Ui[n]=u;
   n++;
```

```
    SumUi+=u;
    if((SumUi/(1.0*m)) >= UTARGMIN && (SumUi/(1.0*m)) <= UTARGMAX){
     flag=0;
    }else{
     if((SumUi/(1.0*m)) > UTARGMAX){
      break;
     }
    }
   }
   if(i==NR_TASKS){
    if(flag){
     printf("Warning: maximum number of tasks achieved!\n");
     exit(-1);
    }
   }
  }
  *nr_tasks=n;
}
void get_ti(unsigned long long TMIN,unsigned long long TMAX,unsigned long long *Ti, int nr_tasks)
{
 int i;
 double factor;
 for(i=0;i<nr_tasks;i++){
  factor=(double)rand()/RAND_MAX;
  Ti[i]=TMIN+factor*(TMAX-TMIN);
 }
}
int main(int argc, char** argv)
{

 int nr_tasks=0,m;
 char file_name[20];
 unsigned long long TMIN,TMAX,Ti[NR_TASKS],seed;
 double Ui[NR_TASKS], UTARGMIN,UTARGMAX,UMIN,UMAX,factor;

 m=atoi(argv[1]);
 UTARGMIN=atof(argv[2]);
 UTARGMAX=atof(argv[3]);
 TMIN=atoll(argv[4]);
 TMAX=atoll(argv[5]);
 UMIN=atof(argv[6]);
 UMAX=atof(argv[7]);
 strcpy(file_name,argv[8]);
 seed=atoll(argv[9]);
 factor=atof(argv[10]);
 srand(seed);

 get_ui(m,UTARGMIN,UTARGMAX,UMIN,UMAX,Ui, &nr_tasks);
 get_ti(TMIN,TMAX,Ti,nr_tasks);

 write_2_file(argv[8],nr_tasks,Ui,Ti,m,UTARGMIN,UTARGMAX,UMIN,UMAX,TMIN,TMAX,factor);

 return 0;
}
```

Listing 16: Task set generator.

B Periodicity

B.1 Job release mechanism

The vanilla Linux kernel does not support any mechanism that sleeps a task until a specific absolute time. This is important for implementing periodically arriving tasks without suffering from cumulative drift. We have implemented a mechanism that allows that: job release mechanism. This mechanism was briefly described in Section 4, so, here we will described it with more detail.

This procedure is supported by the `struct release` data structure (see Listing 17) and is composed by two functions: system call `delay_until` and timer callback `wake_up_job`. The `struct release` data structure is defined per-processor and is composed by an red-black tree (which root is the `root_tasks`), by a `struct task_struct` pointer that points to the task stored in the red-black tree with earliest (`erf` field) and a high resolution timer (`timer` field).

```
struct release {
 struct rb_root root_tasks;
 struct task_struct *erf;
 struct hrtimer timer;
}release;
```

Listing 17: `struct release` data structure.

This procedure is triggered by `delay_until` system call invocation (see Listing 19). Since, this procedure must be as faster as possible, the first steps are disabling the local interrupts and also preemptions. This system call has two 32 bits arguments (`h` and `l`) that specify the next arrival time of the current executing task. Actually, this is a 64 bits variable, therefore, the next steps merge the two 32 bits variables into one 64 bits variable called `arrival_time`. In the Linux kernel, `rq->curr` points to the current executing task on the current processor. After we get the pointer to the current task, then, we have to set up the next release of the job. The first step is checking if the `arrival_time` value is higher than the current time, returned by `retas_clock` function. If it is no deadline miss occurred, otherwise, a deadline miss occurred.

In the former case, a set of steps must be done in order to relinquish the task from processor and set up the timer expiration for the next release: (i) the state of the current task is changed to `TASK_INTERRUPTIBLE`; then, (ii) the current task is inserted into the release red-black tree ordered by absolute arrival time using the `insert_job` function that returns the pointer to the task with the earliest release time; (iii) if the earliest release task is the current task, then the timer expiration must be set accordingly with its arrival time. However, it must be checked if the next release is larger than the current time plus a safe time interval (`EPSILON`) - the purpose of the `EPSILON` is to assure that when the `delay_until` returns the timer was not set to expire in the past. Otherwise, the timer is set up to expire for the minimal time instant in future.

In the later case, a deadline miss occurred, the task continues in the running state (the set of steps done in this case are only for logging and statistical purposes).

Finally, the preemption and interrupts are enabled again and the task must be relinquished from processor. For that, and `schedule` function is invoked.

When the release timer expires the `wake_up_job` callback is invoked. Due to the same reasons given for the `delay_until` system call the local interrupts and preemption are disabled. After that, a loop is executed and all tasks whose arrival time fall into the current time plus a safe time interval (`EPSILON`) are removed from the release red-black tree, their state are changed to `TASK_RUNNING` state and finally, are inserted into the ready red-black tree. The next timer expiration is set according to the following rules: if there is at least one task on the release red-black tree, the timer is set up to expire according to the task with earliest arrival, otherwise, it sets the timer to expiration with a large value that, in practice, this timer will not expire. Finally, the local interrupts and preemption are enabled.

```
asmlinkage long
sys_retas_delay_until(unsigned long h, unsigned long l)
{
 struct rq *rq=NULL;
 struct task_struct *p=NULL;
 struct hrtimer *timer=NULL;
 unsigned long long arrival_time;
```

```
 unsigned long flags;
 local_irq_save(flags);
 preempt_disable();

 arrival_time=0; arrival_time=h;
 arrival_time<<=32;arrival_time|=l;
 rq = cpu_rq(smp_processor_id());
 timer=&rq->retas_rq.release.timer;
 rq->curr->retas_task.job_param.arrival = arrival_time;
 if(arrival_time > retas_clock()){
  rq->curr->state=TASK_INTERRUPTIBLE;
  p=insert_job(rq, rq->curr);
  if(p->retas_task.job_param.arrival < ktime_to_ns(timer->_expires) + EPSILON){
   __hrtimer_start_range_ns(timer, ns_to_ktime(p->retas_task.job_param.arrival),0,
       HRTIMER_MODE_ABS_PINNED,0);
  }
 }
 else{
  _dequeue_task_retas(rq, rq->curr);
  retas_stat_miss_deadline(rq->curr);
  atomic_inc(&rq->curr->retas_task.job_param.nr);
  _enqueue_task_retas(rq, rq->curr);
 }
 local_irq_restore(flags);
 preempt_enable_no_resched();
 schedule();
 return 0;
}
```

Listing 18: `delay_until` system call.

```
static enum hrtimer_restart
wake_up_job(struct hrtimer *timer)
{
 struct rq *rq;
 struct task_struct *p;
    int wake_up_the_job;
 unsigned long flags;

 local_irq_save(flags);
 preempt_disable();
 rq= cpu_rq(smp_processor_id());
 do{
  wake_up_the_job = 0;
  p=peek_erf_job(rq);
        if(p){
   if(p->retas_task.job_param.arrival <= retas_clock() + EPSILON){
    remove_job(rq,p);
    if(p->exit_state!=EXIT_DEAD && p->exit_state!=EXIT_ZOMBIE){
     wake_up_the_job = 1;
     atomic_inc(&p->retas_task.job_param.nr);
     p->state=TASK_RUNNING;
     activate_task(rq, p, 0);
    }
   }
  }
    }while (wake_up_the_job);
 p=peek_erf_job(rq);
 if(p){
  hrtimer_set_expires(timer, ns_to_ktime(p->retas_task.job_param.arrival));
 }
 else{// set the timer to a value which is so large that this timer will not expire.
  hrtimer_set_expires(timer, ktime_set(KTIME_SEC_MAX, 0));
 }
 local_irq_restore(flags);
 preempt_enable_no_resched();
 return HRTIMER_RESTART;
}
```

Listing 19: `wake_up_job` timer callback.

C 24-core machine experimental results

We generated 17 random task sets with an utilization target between 0.88 and 0,885. The period and utilization of tasks varied from 5 ms up to 50 ms and from 0.1 up to 1.0, respectively. The number of tasks varied from 74 up to 191 tasks in the case of the 24-core machine. The time duration of each experiment was approximately 1000 s. All task sets were scheduled using the S-EKG and NPS-F scheduling algorithms (which have comparable jitter/overheads). Since each experiment took 1000 s, the whole set of experiments took 34000 s.

Table 5 presents the experimental results observed in the experiments ran on 24-core machine. There are two values that require more attention. The MIN_{max} value observed for the $CtswJ$ is smaller than the AVG_{τ_i} of task that experienced the MAX_{max}, which denotes that some kind of tasks incur more overheads than others. The task that experienced the MAX_{max} is a split task that executes on processor 21 and 22 and the task that experienced MIN_{max} is a non-split task that executes on a dedicated processor; that is, the task assigning algorithm generated an assigning that this processor only executed non-split tasks.

	MAX_{max} (μs)	AVG_{τ_i} (μs)	MIN_{max} (μs)	AVG_{τ_i} (μs)
$ResJ$	48.087	8.493	19.243	9.498
$RelJ$	37.264	0.731	11.609	0.848
$CtswJ$	9.743	**1.223**	**1.009**	0.542
$IntO$ - irq20	35.584	-	-	-
$IntO$ - tick	6.431	**1.307**	**1.064**	0.755

TABLE 5: *More experimental results (24-core machine).*