Technical Report

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa

Björn Andersson

Eduardo Tovar

HURRAY-TR-100504

Version:

Date: 05-16-2010

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa, Björn Andersson, Eduardo Tovar

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

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

Abstract

Consider the problem of scheduling a set of sporadic tasks on a multiprocessor to meet deadlines even at high processor utilizations. We assume that task preemption and migration is allowed but because of their associated overhead, their frequency of use should be kept small. Task-splitting (also called semi-partitioning) is a family of algorithms that offers these properties. An algorithm in this class assigns most tasks to just one processor but a few tasks are assigned to two or more processors, and they are dispatched in a way that ensures that a task never executes on two or more processors simultaneously. A certain type of task-splitting algorithms, called slot-based split-task dispatching, is of particular interest because of its ability to schedule tasks at high processor utilizations. Unfortunately, no slot-based task-splitting algorithm has been implemented in a real operating system so far.

In this paper, we discuss challenges and design principles for implementing slot-based task-splitting algorithms on multiprocessor systems and also present an implementation of such an algorithm; it is based on the Linux kernel 2.6.28. We have conducted a range of experiments with an 8-core multicore desktop PC utilized to 88% with real-time tasks executing empty for loops and we observe that the behavior of our implementation provides good correspondence between theory and practice.

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar
CISTER-ISEP Research Center
Polytechnic Institute of Porto
4200-072 Porto, Portugal
{*pbsousa,bandersson,emt*}*@dei.isep.ipp.pt*

***Abstract*—Consider the problem of scheduling a set of sporadic tasks on a multiprocessor to meet deadlines even at high processor utilizations. We assume that task preemption and migration is allowed but because of their associated overhead, their frequency of use should be kept small. Task-splitting (also called semi-partitioning) is a family of algorithms that offers these properties. An algorithm in this class assigns most tasks to just one processor but a few tasks are assigned to two or more processors, and they are dispatched in a way that ensures that a task never executes on two or more processors simultaneously. A certain type of task-splitting algorithms, called slot-based split-task dispatching, is of particular interest because of its ability to schedule tasks at high processor utilizations. Unfortunately, no slot-based task-splitting algorithm has been implemented in a real operating system so far.**

In this paper, we discuss challenges and design principles for implementing slot-based task-splitting algorithms on multiprocessor systems and also present an implementation of such an algorithm; it is based on the Linux kernel 2.6.28. We have conducted a range of experiments with an 8-core multicore desktop PC utilized to 88% with real-time tasks executing empty for loops and we observe that the behavior of our implementation provides good correspondence between theory and practice.

***Keywords*-Multiprocessor scheduling, task-splitting, semi-partitioned scheduling, Linux kernel.**

I. Introduction

The real-time systems research community has developed a comprehensive toolkit comprising scheduling algorithms (RM and EDF), schedulability tests and implementation techniques which have been very successful: they are currently taught at major universities world-wide; they are incorporated in design tools and they are widely used in industry. Unfortunately, the results were limited to computer systems with a single processor only.

Today, a multiprocessor implemented on a single chip (called *multicore*) is the preferred platform for many embedded real-time applications however and this brings the pressing need for developing an analogous toolkit for multicores. Such a toolkit for multicore should ideally exhibit the same properties as the uniprocessor toolkit exhibited and that engineers valued: (i) high utilization bound; (ii) few preemptions; (iii) dispatchers with low time-complexity; and (iv) the ability to provide pre-run-time guarantees to schedule sporadically arriving tasks to meet deadlines even with deadlines much shorter than the minimum inter-arrival times.

Researchers have attempted to create real-time scheduling algorithm with these properties. Partitioned scheduling algorithms partition the task set and assign all tasks in one partition to the same processor. This generates few preemptions but unfortunately such algorithms have a utilization bound of at most 50%. Global scheduling algorithms store tasks in one global queue, shared by all processors. At any moment, the m highest-priority tasks among those are selected for execution on the m processors. A class of global scheduling algorithms, called job-static priority algorithms offers few preemptions but unfortunately, such algorithms have a utilization bound of at most 50%. Pfair is a class of global scheduling algorithms which uses dynamic priorities; some algorithms in this class have the utilization bound 100% but unfortunately, they generate a large number of preemptions.

During recent years, the research community has therefore created a family of real-time scheduling algorithms which exhibit all the above mentioned properties. This family of algorithms is called *task-splitting* or *semi-partitioning* [1], [2], [3], [4], [5], [6], [7], [8], [9]. Recent evaluations based on simulation experiments [3] and implementations in real operating systems [10] have demonstrated the excellent performance of this class of algorithms. The key idea of these algorithms is that they assign most of the tasks to just one processor but some of the tasks (called *split tasks*) are assigned to two or more processors. Uniprocessor dispatchers are used on each processor but they are modified to ensure that a split task never executes on two or more processors simultaneously.

One particularly interesting class of task-splitting algorithms is those algorithms where time is subdivided into timeslots such that within timeslots, processor reserves are carefully positioned with a time offset from the beginning of a timeslot. A split task is assigned to two or more processor reserves located on different processors and the positioning of the processor reserve in time is statically assigned (relative to the beginning of a timeslot) so that no two reserves serving the same split task overlap in time — Fig. 2(a) depicts this. Among the types of split-task scheduling algorithms, this is the class that provides the highest utilization bound. In addition, its run-time dispatching does not depend on any data structures that are shared among

all processors and therefore it has the potential to scale to multicore processors with a very large number of processors. For these reasons, we believe an implementation of a slot-based task-splitting algorithm would be valuable.

Three implementations of multiprocessor scheduling algorithms have recently been developed. $\mathrm{Litmus}^{\mathrm{RT}}$ [11], [12], [13] provides a modular framework for different scheduling algorithms (global-EDF, pfair algorithms) for the Linux kernel 2.6.32. Kato *et al*. [10] has also created a modular framework, RESCH, for using other algorithms than $\mathrm{Litmus}^{\mathrm{RT}}$ (partitioned, semi-partitioned scheduling) for the Linux kernel. Faggioli *et al*. [14] has implemented global-EDF in the Linux kernel and made it compliant with POSIX interfaces. The implementation of $\mathrm{Litmus}^{\mathrm{RT}}$ and the POSIX compliant implementation do not support the class task-splitting at all and hence they are not in the scope of our interest. The framework by Kato *et al*. [10] shares some of our goals in that it provides an implementation of task-splitting algorithms. But it uses another type of task-splitting (that is not slot-based split-task dispatching) which cannot guarantee to meet deadlines at high processor utilization. Hence, the current research literature provides no answer to the question whether slot-based task-splitting multiprocessor scheduling can be implemented and whether it works in practice.

Therefore, in this paper, we show that slot-based task-splitting multiprocessor scheduling can be implemented and it works in practice. We do so by implementing a recently-proposed algorithm based on slot-based split-task dispatching [2] in the Linux kernel 2.6.28[1]. It is a new scheduling policy Sporadic Multiprocessor Scheduling (SMS) in the modular scheduling framework in the Linux kernel and we dub this implementation Sporadic Multiprocessor Linux Scheduler (SMLS). We have conducted a range of experiments with an 8-core multicore desktop PC utilized to 88% with real-time tasks executing empty for loops. In order to make the environment more controlled, we (i) set runlevel to 1, (ii) disconnected the desktop PC from the network and (iii) setup eight non-real-time tasks to ensure that the kernel idle threads never start executing. With this experimental setup, we observe that the behavior of our implementation provides good correspondence between theory and practice. Specifically, we observe that (i) no deadline misses occurred, (ii) the release jitter was at most 33μs and (iii) the time when so-called reserves began deviates with at most 20μs from when they should occur.

The remainder of this paper is structured as follows. Section II gives a background on task-splitting, in particular slot-based split-task dispatching and shows its related challenges for implementation. Section III presents a new task-splitting algorithm that is suited for implementation.

[1]The source code of the implementation is available at http://www.cister.isep.ipp.pt/activities/RESCORE/Software.ashx

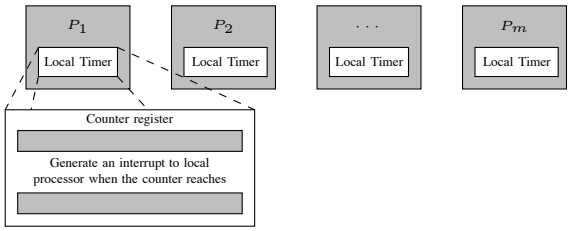

Figure 1: Each processor (P_i) has a local timer.

Section IV illustrates the new task-splitting algorithm that is suited for implementation with an example. Section V shows principles on how to implement slot-based task-splitting and Section VI gives an overview of our implementation. Section VII compares the actual behavior of SMLS to its theoretical behavior. Section VIII gives conclusions.

II. Background

A. System model

Consider n tasks and m identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor in the range $1..m$. Each task τ_i is characterized by worst-case execution time C_i and minimum inter-arrival time T_i and by the time that the execution must be completed, the deadline D_i. We assume $0 \leq C_i \leq D_i$. If we do not state D_i then we assume that $\forall i : D_i = T_i$. For convenience we also define:

$$\mathrm{TMIN} = \min(T_1,\ T_2,\ ...,\ T_n) \tag{1}$$

and let $\tau_{i,k}$ denote the k:th arriving job of task τ_i.

A processor p executes at most one task at a time and no task may execute on multiple processors simultaneously. The utilization of task τ_i, denoted u_i, is defined as $\frac{C_i}{T_i}$ and the system utilization, U_s, is defined as $\frac{1}{m} \cdot \sum_{i=1}^{n} u_i$.

We assume that (i) all processors have the same instruction set and data layout (e.g. big-endian/little-endian), (ii) all processors execute at the same speed and (iii) the speed at which a task executes is independent of which processor it executes on. We 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 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 (see Fig. 1). We assume that this timer provides two functions: (i) one function allows reading the current real-time (that is not calender time) as an integer; and (ii) another function makes it possible to set up the timer to generate an interrupt x time units in the future, where x can be specified.

B. Task-splitting

Consider n=m+1 tasks with T_i=1 and $C_i = 0.5+\epsilon$ (where ϵ is a positive number smaller 1/6) to be scheduled on m processors. It is easy to see that if task migration is not allowed then there is a processor which is assigned at least two tasks. And on this processor, the utilization exceeds 100% and hence a deadline miss occurs. This is problematic since $U_s = \frac{m+1}{m} \cdot (0.5 + \epsilon)$ which becomes $1/2$ as $m \to \infty$ and $\epsilon \to 0$; that is, a deadline miss can occur although only 50% of the entire processing capacity is requested.

Researchers observed [15], [1] that if the execution-time of a task could be "split" into two pieces then it is possible to meet deadlines. For example, assign task τ_i with $i \in \{1, 2, 3, \ldots, m\}$ to processor P_i and assign task τ_{m+1} to two processors (for example processor 1 and processor 2) so that a job by τ_{m+1} executes 0.25+ϵ/2 units on one of the two processors and 0.25+ϵ/2 units on the other. This makes it possible to meet deadlines, assuming that the two "pieces" of task τ_{m+1} are dispatched so that they never execute simultaneously.

Many recent algorithms are based on this idea and they differ in (i) how tasks are assigned to processors and split before run-time and (ii) how tasks are dispatched, particularly, how split tasks are dispatched at run-time. Anderson *et al*. proposed [15] the idea that the second piece of a job of a split task τ_i should arrive T_i time units later. This ensures that the two pieces of such a job do not execute simultaneously but unfortunately it requires that $D_i \geq 2T_i$ so it is recommended only for soft real-time tasks. Andersson and Tovar [1] proposed the idea that time should be subdivided into timeslots of unequal duration and within each timeslot, the first piece of a split task is executed in the beginning of the timeslot and the second piece of a split task is executed in the end of the timeslot. This provides hard real-time scheduling with $D_i = T_i$ and it allows good utilization bounds to be attained and it provides bounds on the number of preemptions but it works only for periodic tasks. Levin *et al*. [16] proposed a related algorithm but with the ability to schedule sporadic tasks. Both algorithms [1], [16] require that when two absolute deadlines are close in time, a task can be assigned a very short segment of time and hence these algorithms [1], [16] are difficult to implement in practice. Kato and Yamasaki [8] proposed a suspension-based split-task dispatching approach where the second piece of a split task is suspended whenever the first piece is executing. This ensures that a split task never executes on two or more processor simultaneously and it provides hard real-time scheduling.

The two approaches for split-task dispatching that we believe are the most promising for implementing and use in practice are (i) *job-based split-task dispatching* [6], [9] and (ii) *slot-based split-task dispatching* [2]. Job-based split-task dispatching splits a job into two or more subjobs and forms a sequence of subjobs and sets the arrival time of a subjob equal to the absolute deadline of its preceding subjob. Job-based split-task dispatching provides a utilization bound greater than 50% and few preemptions. It has been implemented in a real operating system and through experimental studies [10] of that implementation it was found to outperform many other non-split approaches. (Algorithms using slot-based split-task dispatching were not part of the evaluation.) The main drawback of job-based split-task dispatching is that utilization bounds greater than 69% have not been attained [9].

Slot-based split-task dispatching subdivides time into equal-duration timeslots whose beginning and end 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 a reserve, and these two reserves supply processing capacity for a split-task — see Fig. 2(a). Slot-based split-task dispatching causes more preemptions than job-based split-task dispatching but, in return, it offers higher utilization bounds (higher than 69% and configurable for up to 100%) [2] and a recent study [3] of randomly generated task sets shows that it offers the best performance (among all algorithms, not only task-splitting algorithms) for providing pre-run-time guarantees to arbitrary-deadline sporadic tasks. Despite the good performance of slot-based split-task dispatching in theory, the current research literature provides no answer to the question whether slot-based task-splitting multiprocessor scheduling can be implemented and whether it works in practice.

C. Challenges

From Fig. 2(a), we can identify three challenges for implementing slot-based split-task dispatching:

- C1. Timeslots must begin at the same time on all processors;
- C2. A split-task must migrate instantaneously in the beginning of a timeslot;
- C3. The reserves should begin and end at precisely specified time instants.

Since each generation of multicore processors offers greater core count than its preceding generation, we believe it is also important that an implementation of a multiprocessor scheduling algorithm has a dispatching overhead that is low as a function of the number of processors — ideally independent of the number of processors. This poses no challenges for scheduling non-split tasks. For split-tasks however this brings the following two additional challenges:

- C4. The run-time overhead of migration (manipulation of data structures and concurrency control) should be independent of the number of processors;
- C5. The run-time overhead due to handling of timers (reading the current value of a real-time clock; setting up a timer to generate an interrupt signal

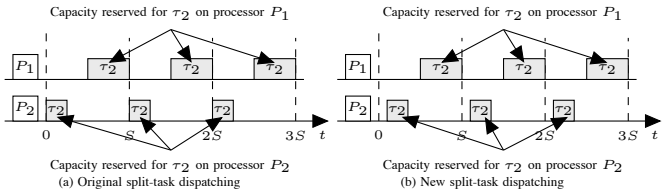

Figure 2: An example of the operation of slot-based task-splitting multiprocessor scheduling. Task τ_2 is a split-task. A non-split task executes only on its dedicated processor; it can execute in a reserve but it does so with a lower priority than a split task.

at a certain time) should be independent of the number of processors.

We will address these challenges in the forthcoming sections. Challenges C1 and C3 will be resolved using high-resolution local timers to each processor. Challenges C4 and C5 will be resolved through carefully designed data structures which avoids synchronization between processors and the local timers will help us overcome C5. The challenge C2 is fundamental however — we can resolve it only by a minor redesign of the actual scheduling algorithm. Section III does that.

III. Slot-based split-task dispatching suited for implementation

Consider Fig. 2(a) again. It shows that task τ_2 must migrate instantaneously at certain instants; this occurs at time S, time $2S$, etc. We can move the reserves on processor 2 so that they start slightly later in each timeslot — Fig. 2(b) shows this.

Let us consider the reserve on processor p such that this reserve is used for the split-task between processor $p-1$ and p. We let $M[p]$ denote the time from the beginning of a timeslot until the beginning of this reserve. For the dispatching algorithm in [2], it holds that $\forall p : M[p] = 0$. In order to implement slot-based split-task dispatching, we need to choose $\forall p : M[p] > 0$; we will now discuss how to choose $M[p]$.

Previous work [2] used a positive integer parameter δ which can be selected by the designer. Based on this parameter, the following definitions were made [2]:

$$S = \frac{\text{TMIN}}{\delta} \tag{2}$$

and

$$\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta + 1)} + \delta \tag{3}$$

and

$$SEP = 4 \cdot (\sqrt{\delta \cdot (\delta + 1)} - \delta) - 1 \tag{4}$$

S is the duration of the timeslot. α is a parameter used for sizing the reserves. SEP is a threshold such that tasks with u_i greater than SEP are assigned their own dedicated processor. SEP also plays the role of being the utilization bound of the algorithm in [2].

Consider Fig. 3 which shows a detailed view of a timeslot. It shows that each processor p has a reserve of duration $x[p]$ and another reserve of duration $y[p]$, and these reserves are used for executing split tasks. The reserve $x[p]$ is used for executing the task split between processor p and $p-1$. The reserve $y[p]$ is used for executing the task split between processor p and $p+1$. If the task which is assigned reserve $x[p]$ has finished execution at time t then processor p selects for execution at time t, a non-split task, which was assigned to processor p. Analogously for $y[p]$. On each processor, there is also a reserve of duration $M[p]$ early in the timeslot and another reserve of duration $N[p]$ in the middle of the timeslot. These reserves are used for executing non-split tasks; the split tasks are forbidden to execute there.

We will assign and split tasks just like in our previous work [2] — we deviate only from our previous work [2] in the way that dispatching of tasks is performed. From our previous work [2], we obtain that (i) adding the duration of the x and y reserve on the same processor gives us at most $(1 - 2 \cdot \alpha) \cdot S$ and (ii) adding the duration of the x reserve on processor p+1 and y reserve on processor p gives us at most $(1 - 2 \cdot \alpha) \cdot S$. Therefore, an appropriate choice is:

$$\forall p : M[p] = \alpha \cdot S \tag{5}$$

It ensures that there is a gap of at least $\alpha \cdot S$ between two reserves on the same processor and also that there is a gap of at least $\alpha \cdot S$ between two reserves on different processors that serve the same split task. One can show that the maximum amount of execution by a split-task in a time interval t, is no higher for $0 < M[p] \leq \alpha \cdot S$ than for $M[p] = 0$ (see Appendix A in [17]). One can also show that the minimum supply of processor time for a split-task in a time interval t, is no lower for $0 < M[p] \leq \alpha \cdot S$ than for $M[p] = 0$ (see Appendix A in [17]). Basically, choosing $M[p] > 0$ makes the execution of a split-task more smooth over time. This gives us that the schedulability analysis in previous work [2] applies also for the case when we choose $M[p] = \alpha \cdot S$ (see Appendix A in [17]). We pay the price of having one extra preemption per timeslot per processor when $M[p] > 0$ though. (This is not visible in Fig. 2(b) but it can be seen with an example with three processors and there is one split-task between processors P_1 and P_2 and another split-task between processors P_2 and P_3. A good illustration of this is the preemption at time 1.25 on processor P_3 in Fig. 4 for task τ_4).

Because of the robustness attained and schedulability maintained by choosing $M[p] = \alpha \cdot S$, we will assume $M[p] = \alpha \cdot S$ in the remainder of this paper. We could use any scheduling algorithm in the reserves but in order to stay as close as possible to the previously proposed slot-based

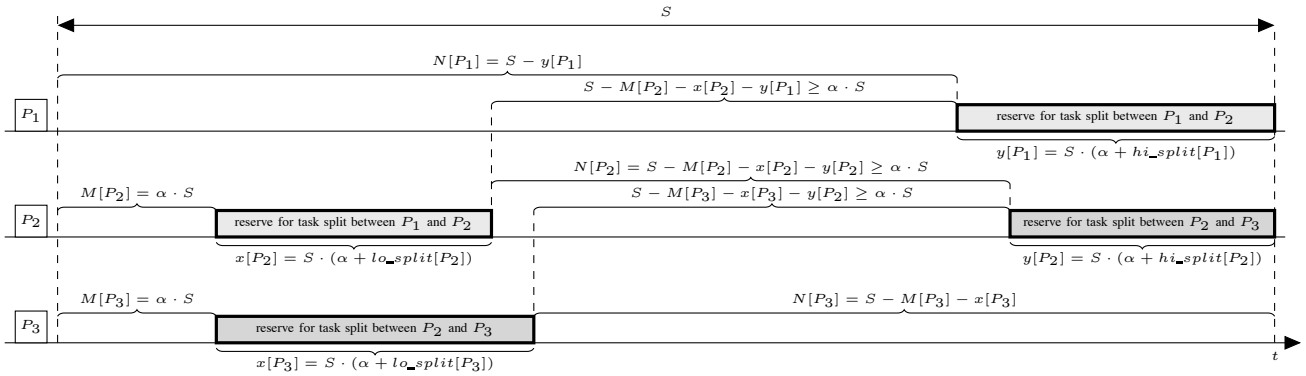

Figure 3: A detailed view of a timeslot, the reserves and their durations, used by our dispatching algorithm that is suited for implementation. The reserves for the split task shared by P_1 and P_2 and shared by P_2 and P_3 are shown by rectangles.

task-splitting algorithm [2], we choose preemptive EDF. The reader can find detailed pseudo-code of the new dispatcher in Appendix B in [17].

IV. An Example

In order to illustrate the behavior of the slot-based split-task dispatching suited for implementation, let us consider an example. We consider a system with four processors ($m = 4$) and seven tasks ($n = 7$) as specified by Table I. The value of δ is four, which means that the processor utilization should be at most 88.85% (SEP parameter is set to 0.8885), except for the processors which have been assigned a task with utilization exceeding SEP.

Table I: Task Set (time unit in millisecond)

Task	C	T	u
τ_1	4.5000	5.0000	0.9000
τ_2	3.5000	6.0000	0.5833
τ_3	3.5000	6.5000	0.5385
τ_4	4.0000	8.0000	0.5000
τ_5	3.0000	7.0000	0.4286
τ_6	3.0000	8.0000	0.3750
τ_7	1.5000	8.5000	0.1765

Table II: Task assignment and splitting

Processor	Task(s)
P_1	τ_1
P_2	τ_2 Part of τ_3
P_3	Part of τ_3 τ_4 Part of τ_5
P_4	Part of τ_5 τ_6 and τ_7

The task assignment works as follows: τ_1 is assigned a dedicated processor (P_1) since the utilization of τ_1 is higher than SEP. τ_2 is assigned to processor (P_2), but assigning task τ_3 to processor P_2 would cause the utilization of processor P_2 to exceed SEP ($0.5833 + 0.5385 > 0.8885$). Therefore, task τ_3 is split between processor P_2 and processor P_3. A portion of task τ_3 is assigned to processor P_2, just enough to make the utilization of processor P_2 equal to SEP, that is 0.3052. This part is referred to as $hi\_split[P_2]$ and the remaining portion (0.2332) of task τ_3 is assigned to processor P_3, which is referred to as $lo\_split[P_3]$. The procedure continues until all tasks have been assigned (see Table II).

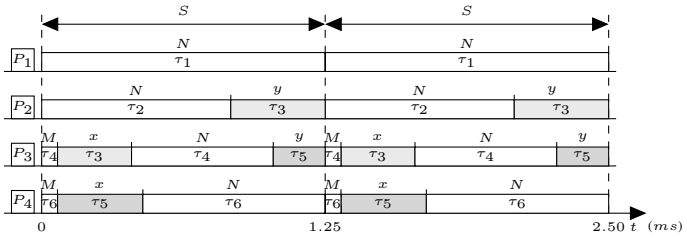

Figure 4: Execution timeline showing how tasks execute if all tasks arrive at time 0.

Fig. 4 shows the execution timeline for the case that for each task, its first job arrives at time 0. The execution of a job is represented by a rectangle labeled with the identifier of the task and above is identified the timeslot component. Timeslot length is equal to $S = \frac{TMIN}{\delta} = \frac{5ms}{4} = 1.25\ ms$. Recall that the online dispatching algorithm works over the timeslot of each processor. As we can see from Fig. 4, for instance, task τ_3 is split between processors P_2 and P_3 and hence it executes only on the x-reserve of processor P_3 and on the y-reserve of processor P_2.

Non-split tasks τ_6 and τ_7 execute on processor P_4 (within M and N reserves), but task τ_7 does not appear in the Fig. 4 because it executes after time 2.5, since its absolute deadline is higher than absolute deadline of task τ_6.

V. How to implement slot-based task-splitting

Recall the challenges listed in Section II. In order to cope with them, we recommend that an implementation of a task-splitting follows the following design principles:

P1. Each processor should have its own run-queue (the queue that stores tasks which have outstanding request for execution). The run queue of processor p should store non-split tasks assigned to processor p. The run-queue of each processor should support the operations

`insert`, `peek_highest_priority_task` and `extract_highest_priority_task` with low time-complexity (using for example a red-black tree).

P2. For each processor p, there should be a data structure with two variables `hi_split` and `lo_split`. The variable `hi_split` of processor p and the variable `lo_split` of processor p+1 should point to the process control block (called `task_struct` in the Linux kernel) for the task that is split between them. If no such task exist then these pointers are NULL.

P3. Each processor should have a variable called `begin_curr_timeslot`. It should hold a time which is no larger than the current time and it should never be less than current time minus S. The variable `begin_curr_timeslot` should be incremented by S to ensure this. This assures that the beginning of the timeslot on each processor is synchronized and avoids the lock mechanism that would be necessary if this variable was global.

P4. Each processor should have a timer queue of events in the future. This should always include the time of the beginning of the next timeslot, that is `begin_curr_timeslot` + S. If applicable, it also contains the time when the reserve in the beginning of the timeslot ends and also the time when the reserve in the end of the timeslot begins. Whenever the timer queue changes (for example an event has expired and therefore should be removed from the timer queue, or a new event is inserted into the timer queue), the processor should disable interrupts, set up a timer x time units in the future where x is the time of the earliest event in the timer queue minus current time, and then enable interrupts. This is a standard approach for timers and it ensures that cumulative drift because of finite speed of the processor does not occur (see page 38 in [18] for discussion).

P5. The operating system should implement a `delay_until` system call (see page 38 in [18]) which makes it possible for a task to sleep until an absolute time. This is important for implementing periodically arriving tasks without suffering from cumulative drift [18].

VI. The implementation

We have implemented the scheduling algorithm as described in Section III (which is a slight modification of our previously proposed scheduling algorithm in [2]) in the Linux kernel 2.6.28. We developed a new scheduling policy (called SMS) in the modular scheduling infrastructure of the Linux kernel and when doing so we followed the design principles stated in Section V.

Currently, the Linux kernel has three native scheduling modules: *RT* (Real-Time), *CFS* (Completely Fair Scheduling) and *Idle*. These modules are hierarchically organized by priority in a linked list and the dispatcher looks for a runnable task of each module in a decreasing order priority.

We added SMS scheduling policy module on top of the native Linux module hierarchy, thus it is the highest priority module (see Fig. 13).

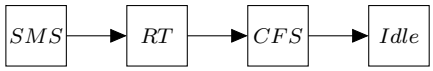

Figure 5: Priority hierarchy of scheduling policy modules

Before describing the SMS module implementation let us give some details about the data structure used by the SMS module. Note that, each processor holds a run-queue to manage all active processes or tasks. In our implementation we added the required data for the SMS algorithm to the `struct rq` data structure (see Listing 1), which is the data structure used for each run-queue. For instance, all non-split tasks of each processor are organized in a red-black tree by the absolute deadline. Red-black trees, which are balanced binary trees whose nodes are ordered by a key and most operations are done in $O(\log n)$ time, are already implemented in the Linux kernel (`lib/rbtree.c`).

```
struct rq {
 ...
 struct split_task {
  struct task_struct *lo_split;
  struct task_struct *hi_split;
 }split_task;

 struct rb_root root_non_split_tasks;

 struct timeslot {
  unsigned long long begin_curr_timeslot;
  unsigned long long m;
  unsigned long long x;
  unsigned long long n;
  unsigned long long y;
  struct hrtimer timer;
 }timeslot;
 ...
};
```

Listing 1: Fields added to `struct rq` kernel data structure. There is one `struct rq` for each processor.

The Linux kernel is currently tick-driven, the dispatcher is invoked periodically (with a period of 1 ms if the macro HZ is set to 1000). Recall however (from Section II-C) that in slot-based split-task dispatching, reserves must begin at precisely specified instants and the periodic tick is not sufficiently precise for our purpose. The Linux kernel is currently provided by the high-resolution timers infrastructure (`kernel/hrtimer.c`) that allow us to specify when a timer should fire at nanosecond resolution. Therefore, we use the high-resolution timers to invoke callback functions for the beginning of a reserve and we also use them to wake up tasks that have executed `delay_until`.

The data of each active process or task in the system is managed using a data structure called `struct task_struct` (see Listing 5). We added also some fields to this data structure required by the algorithm. For instance, `cpu1` and `cpu2` fields are used to set the logical identifier of processor(s) in which the task will be executed. In order to organize SMS tasks by the absolute deadline on a red-black tree we added `struct rb_node node_non_split_task` field.

```
struct task_struct {
 ...
 struct sms_task_param{
  int cpu1;
  int cpu2;
  struct sms_job_param{
   unsigned long long deadline;// absolute deadline
   unsigned long long release; // release time of next job
   ...
  };
  struct rb_node node_non_split_task;
 };
 ...
};
```

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

According to the modular scheduling framework rules, each module must implement the set of functions specified in the `sched_class` structure. Listing 7 shows the definition of `sms_sched_class`, which implements the SMS module. The first field (`next`) of this structure is a pointer to `sched_class` which is pointing to the `rt_sched_class` that implements the RT module.

The other fields are functions that act as callbacks to specific events. The `enqueue_task_sms` is called whenever an SMS task becomes runnable. This function must check if it is a non-split task or a split task. In the former case it must insert a node in the red-black tree and in the latter it does nothing. When an SMS task is no longer runnable, then the `dequeue_task_sms` function is called that undoes the work of the `enqueue_task_sms` function. As the name suggests, `check_preempt_curr_sms` function, checks whether the currently running task must be preempted. This function is called following the enqueuing or dequeuing of a task and it only sets a flag that indicates to the scheduling infrastructure that the currently running task must be preempted. `pick_next_task_sms` function selects the task to be executed by the processor. This function is called by the scheduling infrastructure whenever the currently running task is marked to be preempted. `task_tick_sms` function is mostly called from time tick functions. In the current implementation this function calls the `check_preempt_curr_sms` function, to check, if the current task must be preempted.

```
const struct sched_class sms_sched_class = {
 .next = &rt_sched_class,
 .enqueue_task = enqueue_task_sms,
 .dequeue_task = dequeue_task_sms,
 .check_preempt_curr = check_preempt_curr_sms,
 .pick_next_task = pick_next_task_sms,
 .task_tick = task_tick_sms,
 ...
};
```

Listing 3: `sms_sched_class` definition

The dispatching algorithm is mainly implemented by the `check_preempt_curr_sms` and `pick_next_task_sms` functions. Next, the dispatching algorithm is described assuming that processor p is executing the dispatcher.

One of the arguments of the `check_preempt_curr_sms` function is a pointer (`struct rq * rq`) to the runqueue of processor p (see Listing 4), where all runnable SMS tasks assigned to it are stored, as well as other important data necessary for SMS scheduling algorithm, such as `begin_curr_timeslot`, the timeslot composition reserves and also `hi_split` and `lo_split` pointers. The relative time instant within the current timeslot is given by invocation of the `get_timeslot_reserve` function. Assuming that, `get_timeslot_reserve` invocation returns `RESERVE_X`, which means that the current time instant falls in the x reserves. Then, the next step is to check if the split-task is in running state. If it is (`get_lo_split_task` function returns pointer to the split-task) and if it is not the currently running task on processor p, then, there is the need to check if the split-task is currently running on processor p-1 (which is identified by the `cpu1` field) by invoking the `cpu_curr` function. If it is, an interprocessor interrupt is sent to force rescheduling[2] on processor p-1 to stop the execution of the split-task (invoking the `resched_cpu` function), then, it checks if there is a non-split task ready to execute on processor p. Otherwise, `resched_task` function is invoked to mark the currently running task on it to be preempted.

```
static void check_preempt_curr_sms(struct rq *rq, struct
     task_struct *p)
{
 ...
 r=get_timeslot_reserve(rq);
 switch(r){
  ...
  case RESERVE_X:
   lo_split=get_lo_split_task(rq);
   if(lo_split!=NULL){
    if(lo_split!=rq->curr){
     task=cpu_curr(lo_split->cpu1);
     if(task==lo_split){
      resched_cpu(lo_split->cpu1);
      goto check_non_split_task;
     }else
      resched_task(rq->curr);
    }
   }else
    goto check_non_split_task;
  break;
  case RESERVE_M:
  case RESERVE_N:
```

[2]This should never happen since the two reserves for a split-task are non-overlapping and there is a time gap (at least $\alpha cdotS$) between them. In our experiments we did not observe any occurrence of the event that `resched_cpu` was invoked because a split task was already executing on another processor.

```
    goto check_non_split_task;
   break;
  }
  return;
check_non_split_task:
  task=get_earliest_deadline(rq);
  if(task!=NULL)
   if(task!=rq->curr)
    resched_task(rq->curr);
  return;
}
```

Listing 4: C code fragment of `check_preempt_curr_sms` function.

The algorithm of the `pick_next_task_sms` function is similar to the `check_preempt_curr_sms` function. However, this function returns the pointer to the selected task or `NULL` if there is no ready task to be executed.

Due to space restrictions, we refer the reader to Appendix C in [17] for an extensive description of the implementation.

VII. Experimental Evaluation

We are interested in experimentally assessing whether the behavior of our implementation deviates from theory (with changes described in Section III). For this purpose, we present the experimental setup used and describe the concepts used to characterize the discrepancy between theory and practice. Finally, we present actual experimental results.

A. Experimental setup

The experimental machine is equipped with two Intel®Xeon ®(QuadCore) at 1.60 GHz processors and 4GB of main memory. The experiment was conducted on the Linux kernel 2.6.28 running on runlevel 1 with all interrupts managed by the core 8. We also disabled the network connection. We observed that the in-kernel idle task delays the delivery of timer-expiry callbacks and therefore we setup eight non-real-time tasks, executing idle loops in user space, so that the in-kernel idle task never executes.

We will measure and record the time of occurrences of important events and then compute bounds between the times of these events. In order to characterize deviations of the actual behavior from theory, we define quantities in the next subsection.

B. Concepts used to characterize the outcome of experiments

Fig. 6 shows $meas\_reserve\_J_{i,k}$, which means *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 M, 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 when actually the timer interrupt happens.

$meas\_J_{i,k}$, which means *measured release jitter of job* $\tau_{i,k}$, denotes the difference in time from when the job

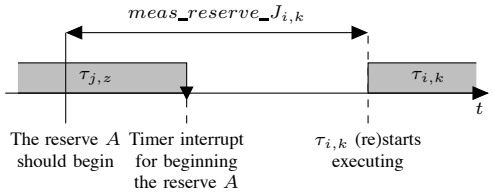

Figure 6: Measured reserve jitter.

$\tau_{i,k}$ should arrive until it is inserted in the ready queue. Fig. 7 shows this and other related quantities. The *measured response time of the job* $\tau_{i,k}$ ($meas\_RT_{i,k}$) is computed as the time difference between the time the job $\tau_{i,k}$ should arrive (this is computed according the input parameter to `delay_until`) and when $\tau_{i,k}$ finishes its execution. The *measured tardiness of job* $\tau_{i,k}$ is defined as $meas\_Tard_{i,k} = \max(0, meas\_RT_{i,k} - D_i)$.

Let $meas\_Nr\_jobs_i$ denote the number of jobs released of task τ_i during an experiment. Then we define:

$$meas\_reserve\_J_i = \max_{k=1...meas\_Nr\_jobs_i}(meas\_reserve\_J_{i,k})$$

The symbols $meas\_J_i$, $meas\_RT_i$ and $meas\_Tard_i$ are defined analogously.

C. Task set generation

In order to generate the task sets we have to define the number of tasks (n), the number of processor (m) and also the target utilization of the task set (U_{target}). With these parameters we compute the utilization of each task (u_i) as follows: $u_i = (n - i + 1) * (U_{target} * m)/(\sum_{i=1}^{n} i)$.

For generating T_i we need to define the minimum T_i, denoted TMIN, and the maximum T_i, denoted TMAX. Then, T_i, in ascending order, is computed as follows: $T_i = \text{TMIN} + (i-1)/(n-1) * (\text{TMAX} - \text{TMIN})$ and in descending order, is computed as follows: $T_i = \text{TMAX} - (i-1)/(n-1) * (\text{TMAX} - \text{TMIN})$. In some cases after computing the T_i we shuffle it and this way we get a random task set. We calculate C_i as $C_i = T_i * u_i$.

Table III shows 24 task sets and the input parameters used to generate each task set. All task sets were generated with m equal to eight and U_{target} equal to 0.888 and δ equal to four. There are task sets with 100 tasks and with 15 tasks. TMIN is equal in all task sets and TMAX is either 15 or 100 ms. For the same experimental parameters we have varied the T_i order (column Tsorted in Table III): (i) we set T_i in ascending order (a) to get high utilization tasks with lower T_i; (ii) we set T_i in descending order (d) to get high utilization tasks with higher T_i and (iii) we shuffle (s) T_i to obtain some random task sets. Our experiments are divided into two groups, the first twelve experiments are periodic and the other ones are sporadic. In these experiments, the

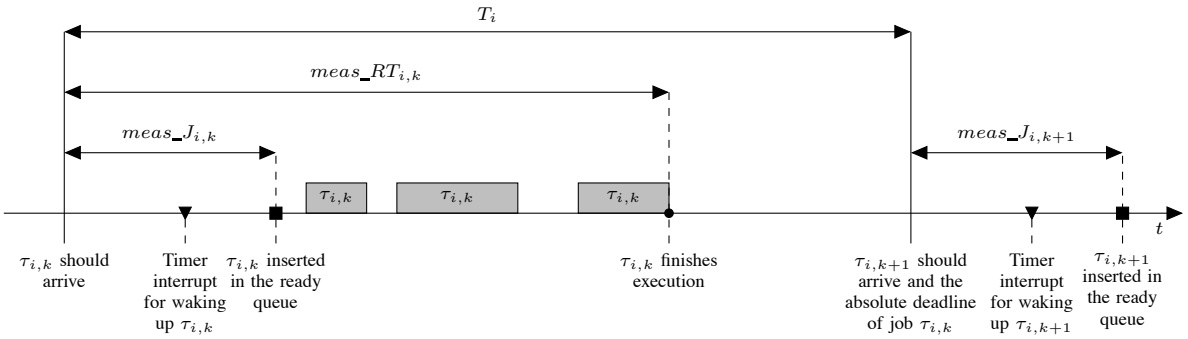

Figure 7: Time intervals that we compute from logged events.

time between two consecutive job arrivals of the same task τ_i is given by a random variable in $[T_i, \text{factor} * T_i]$. For instance, if the minimal inter-arrival time of one task is five ms and factor $= 1.5$ then the maximum inter-arrival time of that task is 7.5 ms (5 multiply by 1.5).

The criterion used to stop the experiment was the number of jobs, thus, when a task has released 100000 jobs the experiment finishes.

D. Experimental results

Table IV presents the results. Note that, here we present the maximum values of the task set. Therefore, we define:

$$meas\_J_\tau = \max_{i=1\ldots n}(meas\_J_i)$$

and analogously $meas\_reserve\_J_\tau$ and $meas\_Tard_\tau$.

All task sets were generated with δ equal to four and TMIN equal to five ms. The timeslot length (S) is equal to 1.25 ms and $\alpha \cdot S$ is equal to 0.035 ms. The latter value is important because it defines the minimal length of every timeslot reserves. And if the measured reserve jitter ($meas\_reserve\_J_\tau$) is higher than this value could cause misbehavior of the algorithm, because some reserves are skipped.

We can see from Table IV that the maximum $meas\_J_\tau$ of all experiments is equal to 0.033 ms and for $meas\_reserve\_J_\tau$ the maximum value is 0.020 ms and no deadline misses occur. The reason for this is that the reserve jitter is 0.020 ms which is smaller than $M[p]$ which is 0.035 ms and hence the observed behavior is similar to the theoretical behavior.

These good results are due to (i) the controlled experimental environment (stated in Section VII-A), (ii) the use of the local high-resolution timers and (iii) the fact that our scheduling algorithm allows each processor to operate without synchronizing with the other processors.

VIII. Conclusions

We have shown that slot-based task-splitting multiprocessor scheduling can be implemented and it works in practice. We did so by implementing an algorithm based on slot-based split-task dispatching [2] in the Linux kernel 2.6.28. We have conducted a range of experiments with an 8-core multicore desktop PC utilized to 88% with real-time tasks executing empty for loops. In order to make the environment more controlled, we (i) set runlevel to 1, (ii) disconnected the desktop PC from the network and (iii) setup eight non-real-time tasks to ensure that the kernel idle threads never start executing. With this experimental setup, we observe that the behavior of our implementation provides good correspondence between theory and practice. Specifically, we observe that (i) no deadline misses occurred, (ii) the release jitter was at most 33μs and (iii) the time when so-called reserves began deviates with at most 20μs from when they should occur.

It should be noted that although this paper presents an implementation and experimental evaluation of the algorithm in [2], the same implementation ideas and the same implementation can also be used for the algorithm in [3] as well because it uses the same dispatch mechanism. This is relevant since the algorithm in [3] was (and still is) the algorithm that, in theory, has the best ability (among state-of-art algorithms) to offer pre-run-time guarantees to arbitrary-deadline sporadic tasks on a multiprocessor.

References

[1] 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.

[2] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *20th Euromicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, 2008, pp. 243–252.

[3] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic tasks on multiprocessors," in *29th*

Table III: Experiment parameters (time unit is ms)

Id	n	m	U_{target}	TMIN	TMAX	**T sorted**	**Factor**	δ
exp01	100	8	0.888	5	15	a	1.0	4
exp02	100	8	0.888	5	15	d	1.0	4
exp03	100	8	0.888	5	15	s	1.0	4
exp04	100	8	0.888	5	100	a	1.0	4
exp05	100	8	0.888	5	100	d	1.0	4
exp06	100	8	0.888	5	100	s	1.0	4
exp07	15	8	0.888	5	15	a	1.0	4
exp08	15	8	0.888	5	15	d	1.0	4
exp09	15	8	0.888	5	15	s	1.0	4
exp10	15	8	0.888	5	100	a	1.0	4
exp11	15	8	0.888	5	100	d	1.0	4
exp12	15	8	0.888	5	100	s	1.0	4
exp13	100	8	0.888	5	15	a	1.5	4
exp14	100	8	0.888	5	15	d	1.5	4
exp15	100	8	0.888	5	15	s	1.5	4
exp16	100	8	0.888	5	100	a	1.5	4
exp17	100	8	0.888	5	100	d	1.5	4
exp18	100	8	0.888	5	100	s	1.5	4
exp19	15	8	0.888	5	15	a	1.5	4
exp20	15	8	0.888	5	15	d	1.5	4
exp21	15	8	0.888	5	15	s	1.5	4
exp22	15	8	0.888	5	100	a	1.5	4
exp23	15	8	0.888	5	100	d	1.5	4
exp24	15	8	0.888	5	100	s	1.5	4

Table IV: Experimental results (time unit is ms)

Id	$meas\_J_\tau$	$meas\_reserve\_J_\tau$	$meas\_Tard_\tau$
exp01	0.02720	0.00771	0
exp02	0.02872	0.00729	0
exp03	0.03078	0.00763	0
exp04	0.02884	0.02009	0
exp05	0.03285	0.01181	0
exp06	0.03252	0.01144	0
exp07	0.00846	0.01075	0
exp08	0.00534	0.00768	0
exp09	0.00635	0.00693	0
exp10	0.00537	0.00619	0
exp11	0.00499	0.00694	0
exp12	0.00645	0.00613	0
exp13	0.02894	0.01903	0
exp14	0.03043	0.00725	0
exp15	0.02885	0.00919	0
exp16	0.01144	0.01472	0
exp17	0.03249	0.01485	0
exp18	0.03088	0.01569	0
exp19	0.01759	0.00713	0
exp20	0.01548	0.01327	0
exp21	0.01747	0.01170	0
exp22	0.01632	0.00681	0
exp23	0.01612	0.00629	0
exp24	0.01670	0.00641	0

IEEE Real-Time Systems Symposium (RTSS 08), Barcelona, Spain, 2008, pp. 385–394.

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

[5] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *21st Euromicro Conference on Real-Time Systems (ECRTS 09)*, Dublin, Ireland, 2009, pp. 239–248.

[6] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *21st Euromicro Conference on Real-Time Systems (ECRTS 09)*, Dublin, Ireland, 2009, pp. 239–248.

[7] S. Kato and N. Yamasaki, "Portioned EDF-based scheduling on multiprocessors," in *8th ACM/IEEE International Conference on Embedded Software (EMSOFT 08)*, Atlanta, GA, USA, 2008, pp. 139–148.

[8] ——, "Real-time scheduling with task splitting on multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 07)*, Daegu, Korea, 2007, pp. 441–450.

[9] N. Guan, M. Stigge, and W. Y. G. Yu, "Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 10)*, Stockholm, Sweden, 2010, pp. 165–174.

[10] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Technical Report CMU-ECE-TR09-12, Tech. Rep., 2009.

[11] 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 *27th IEEE Real-Time Systems Symposium (RTSS 06)*, Rio de Janeiro, Brazil, 2006, pp. 111–126.

[12] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *29th IEEE Real-Time Systems Symposium (RTSS 08)*, Barcelona, Spain, 2008, pp. 157–169.

[13] B. Brandenburg and J. Anderson, "On the implementation of global real-time schedulers," in *30th IEEE Real-Time Systems Symposium (RTSS 09)*, Washington, D.C., USA, 2009, pp. 214–224.

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

[15] J. H. Anderson, V. Bud, and U. C. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS 05)*, Palma de Mallorca, Balearic Islands, Spain, 2005, pp. 199–208.

[16] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *22nd Euromicro Conference on Real-Time Systems (ECRTS 10)*, Brussels, Belgium, 2010, pp. 3–13.

[17] P. B. Sousa, B. Andersson, and E. Tovar, "Implementing slot-based task-splitting multiprocessor scheduling," CISTER Research Center, ISEP/IPP, Polytechnic Institute of Porto, Tech. Rep., 2010, http://www.cister.isep.ipp.pt/docs/.

[18] A. Burns, *Concurrency in Ada*. Cambridge University Press, 1998.

[19] A. Kumar, "Multiprocessing with the completely fair scheduler," IBM, Tech. Rep., 2008.

APPENDIX A.
ALLEVIATING IMPLEMENTATION-RELATED ISSUES

In this section, we will show a previously known result, then modify our scheduling algorithm and then show that the previously known result holds also for this modified algorithm.

A. Previously known result

Recall from previous work [2] that a slot-based split-task multiprocessor scheduling algorithm has been proposed. For most tasks, it holds that a task is assigned to a single processor but for a few tasks, it holds that a task is assigned to two processors (these tasks are called *split-tasks*). Split tasks execute in reserves. A split tasks may execute on an x-reserve, which starts when the timeslot starts, or in an y-reserve, which finishes when the timeslot finishes. For such a setting, we know from previous work (Appendix A in [2]) that:

Lemma 1. *For any interval of length $\Lambda \geq \delta \cdot S$, if t_{sp} denotes the cumulative time within said interval belonging to split task reserves x, z on some processor, then:*

$$\frac{\Lambda - t_{sp}}{\Lambda} \geq \frac{\delta \cdot (S - x - z)}{\delta \cdot S + (x + z)}$$

Proof: Observe that, for any Λ, t_{sp} (the cumulative time belonging to reserves for split tasks within the interval considered) is maximal if the interval of length Λ starts at the same time as reserve z.

Hence

$$t_{sp} \leq \left\lfloor \frac{\Lambda}{S} \right\rfloor \cdot (x + z) + \min(\Lambda - \left\lfloor \frac{\Lambda}{S} \right\rfloor \cdot S, x + z) \tag{6}$$

The right-hand side of Inequality 6 $\big($which we will denote as $h(\Lambda)$ $\big)$ is a continuous function of $\Lambda \in [\delta \cdot S..\infty)$. It is piecewise differentiable, non-decreasing in intervals $(k \cdot S,\ k \cdot S + (x + z))$ and constant in intervals $(k \cdot S + (x + z),\ (k + 1) \cdot S)$, $\forall k \in \mathbb{N}$.

$$\frac{dh(\Lambda)}{d\Lambda} = \begin{cases} 1, & k \cdot S < \Lambda < k \cdot S + x + z \\ 0, & k \cdot S + x + z < \Lambda < (k + 1) \cdot S \end{cases}$$

Then we have

$$\frac{d}{d\Lambda}\left(\frac{h(\Lambda)}{\Lambda}\right) = \begin{cases} \frac{\Lambda - h(\Lambda)}{\Lambda^2} \geq 0, & k \cdot S < \Lambda < k \cdot S + x + z \\ \frac{-h(\Lambda)}{\Lambda^2} < 0, & k \cdot S + x + z < \Lambda < (k+1) \cdot S \end{cases}$$

The global maximum for $\frac{h(\Lambda)}{\Lambda}$ over $[\delta \cdot S,\ \infty)$ then occurs for some L in the set $\{k \cdot S + (x+z)\} \cap [\delta \cdot S,\ \infty)$, $\forall k \in \mathbb{N}$ which is the same set as $\{k \cdot S + (x+z)\}$, $\forall k \in \{\delta,\ \delta+1,\ \delta+2,\ \ldots\}$.

Additionally, for any integer $k \geq \delta$, it holds that

$$\begin{gathered} \frac{h(\Lambda)}{\Lambda}\Big|_{\Lambda = k \cdot S + (x+z)} - \frac{h(\Lambda)}{\Lambda}\Big|_{\Lambda = (k+1) \cdot S + (x+z)} = \\ \frac{(k+1)(x+z)}{k \cdot S + (x+z)} - \frac{(k+2)(x+z)}{(k+1) \cdot S + (x+z)} = \\ \frac{(x+z) \cdot (S - x - z)}{(k \cdot S + (x+z)) \cdot ((k+1) \cdot S + (x+z))} \geq 0 \end{gathered}$$

therefore $\frac{h(\Lambda)}{\Lambda}$ is maximised over $[\delta \cdot S,\ \infty)$ for $\Lambda = \delta \cdot S + (x + z)$. Equivalently, $\frac{\Lambda - h(\Lambda)}{\Lambda} = 1 - \frac{h(\Lambda)}{\Lambda}$ is minimised over $[\delta \cdot S,\ \infty)$ for $\Lambda = \delta \cdot S + (x + z)$ and the respective minimum is $\frac{\delta \cdot (S - x - z)}{\delta \cdot S + (x + z)}$ (which proves the lemma). ■

We also know that:

For any split task τ_i to always be schedulable, it must hold that:

$$\frac{\delta \cdot (x + y)}{(\delta + 1) \cdot \frac{\mathrm{TMIN}}{\delta} - (x + y)} \geq \frac{C_i}{T_i} \tag{7}$$

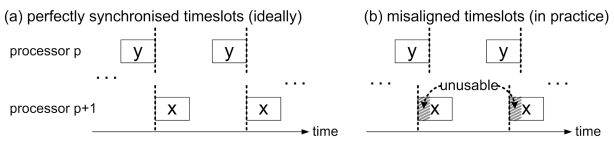

Figure 8: The reserves of a split task (a) ideally do not overlap but (b) this may occur in practice due to timeslot misalignment, wasting part of the time budget for the task.

B. Modifying algorithm and prove that it does not jeopardize timeliness

As already mentioned at the end of Section II, a practical issue with our previously-proposed dispatching approach [2], arises from the difficulty in achieving perfect temporal alignment of timeslot boundaries across processors. For example:

- If each processor reads the time from its own clock, then the crystals of those clocks may have slightly different oscillation frequencies. The slight clock drift will eventually accumulate, leading to severely misaligned timeslots accross processors, unless frequent forced resynchronisation of timeslots is performed.
- Even if all processors read the time from a single source (i.e. a system-wide clock or a global variable only updated by a particular processor with the reading of its clock), slight misalignment may occur.

In turn, timeslot misalignment impairs the schedulability of split tasks. With misaligned timeslot boundaries across "neighboring" processors p and $p+1$, the reserves of the task split between them may overlap. During this time, the split task can only use one of the two, thus part of its time budget is wasted (see Fig 8). Therefore, potential overlap of the reserves of a split task must be precluded.

As we mentioned in Section III, we may achieve this bybreaking up the back-to-back execution of the y- and x-reserve of a split task, by shifting its x-reserve later in time by an offset $M{<}2{\cdot}\alpha{\cdot}S$. This offset acts as a temporal "cushion", preventing reserve overlap (thus ought to be longer than the potential misalignment of reserves). The offset M may be identical for all split tasks or, instead, each split task τ_i may use a different $M[p]$ (p being the index of the processor where the x-reserve of τ_i executes). Fig 9 depicts this modified dispatching.

We will prove that schedulability is not compromised by this modification.

Definition 1. *The worst-case usable time ratio $\rho(\tau_i)$ for a split task τ_i is defined as the lowest observable ratio $\frac{t^{split}}{\Delta t}$, over all time windows of length $\Delta t \geq T_i$, where t^{split} represents sum of the lengths of the subintervals of Δt during which τ_i may execute*

Definition 2. *The worst-case usable time ratio $\rho(\tau[p])$ for the set of non-split tasks $\tau[p]$ assigned to processor p is defined as the lowest observable ratio $\frac{t^{non-split}}{\Delta t}$, over all time windows of length $\Delta t \geq T_i$, where $t^{non-split}$ represents sum of the lengths of the subintervals of Δt during which the execution of tasks of $\tau[p]$ is allowed and the execution of split tasks is disallowed on processor p.*

Lemma 2. *If, for a split task τ_i, it holds that $\rho(\tau_i) \geq \frac{C_i}{T_i}$, then τ_i never misses a deadline.*

Proof: By definition, from its arrival until its deadline T_i a job by τ_i will have been granted no less than $\rho(\tau_i) \cdot T_i \geq \frac{C_i}{T_i} \cdot T_i = C_i$ time units of execution. Therefore, it will have completed by the time of its deadline. ■

Lemma 3. *Assume that the reserves x, y of a split task τ_i have been sized according to the algorithm of Fig* **??** *and that no spacing offset M has been applied for the dispatching. It then holds that:*

$$\frac{\delta \cdot (x+y)}{(\delta+1) \cdot S - x - y} \geq \frac{C_i}{T_i}$$

Proof: Remember that, in the absence of any offset M for the x-reserve, we had selected the amount of reserve inflation α such that Inequality 7 is always met. In turn, the left-hand side of Inequality 7 corresponds to $\rho(\tau_i)$. ■

Theorem 1. *Shifting the x-reserve of a split task τ_i later by an offset of $M_z < 2 \cdot \alpha \cdot S$ (with z denoting the index of the processor where the x-reserve of τ_i executes) does not compromise the schedulability of τ_i.*

Proof: Fig 10(a) depicts (drawn for $\delta{=}1$) the time window for which $\rho(\tau_i)$ is observable for split task τ_i, in the absence of spacing between its reserves. But if such spacing is introduced (via an offset M for its x-reserve), there are four candidates

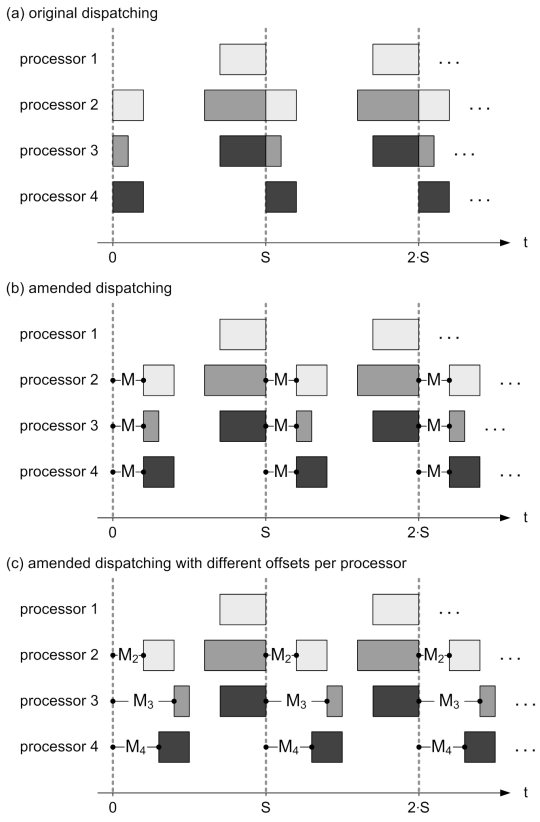

Figure 9: Example of dispatching, in (b) and (c), with x-reserves offset by $M < 2 \cdot \alpha \cdot S$ time units, vs the original dispatcher (a). Reserves of different split tasks are colored differently.

for this "worst-case" time window. Each one starts just as a reserve becomes unavailable (either x or y) and ends just as a reserve becomes available (again, either x or y). All four combinations are seen in Fig 10(b), with each time window annotated by its length (L_1 to L_4) and the respective t^{split}.

Let, ρ_1^{split} to ρ_4^{split} denote, respectively for each of the above cases, the ratio of time usable for the execution τ_i (respectively, t_1^{split} to t_4^{split})to the overall interval length (respectively, L_1 to L_4). Since, for any split task τ_i, the ratio $\frac{t^{split}}{L}$ (with the constraint that $L \geq \delta \cdot S$) is minimal for one of those four candidate intervals, it follows that

$$\rho(\tau_i) = \min_{k \in \{1,2,3,4\}} \rho_k^{split} \stackrel{\text{def}}{=} \min_{k \in \{1,2,3,4\}} \frac{t_k^{split}}{L_k} \tag{8}$$

Then, from Lemma 2, a *necessary* condition for it to be possible for τ_i to miss deadlines is

$$\exists k \in \{1,2,3,4\} : \rho_k^{split} < \frac{C_i}{T_i} \tag{9}$$

We will prove that the above condition can never hold, by examining the four cases one by one:
Case 1: For it to hold that $\rho_1^{split} < \frac{C_i}{T_i}$ it has to be that

$$\frac{t_1^{split}}{L_1} < \frac{C_i}{T_i} \stackrel{Lem.\ 3}{\Longrightarrow}$$

$$\frac{\delta \cdot (x+y)}{(\delta+1) \cdot S - x - y - M} < \frac{\delta \cdot (x+y)}{(\delta+1) \cdot S - x - y}$$

$$\Leftrightarrow M < 0$$

which is impossible. Hence it holds that $\rho_1^{split} \geq \frac{C_i}{T_i}$.

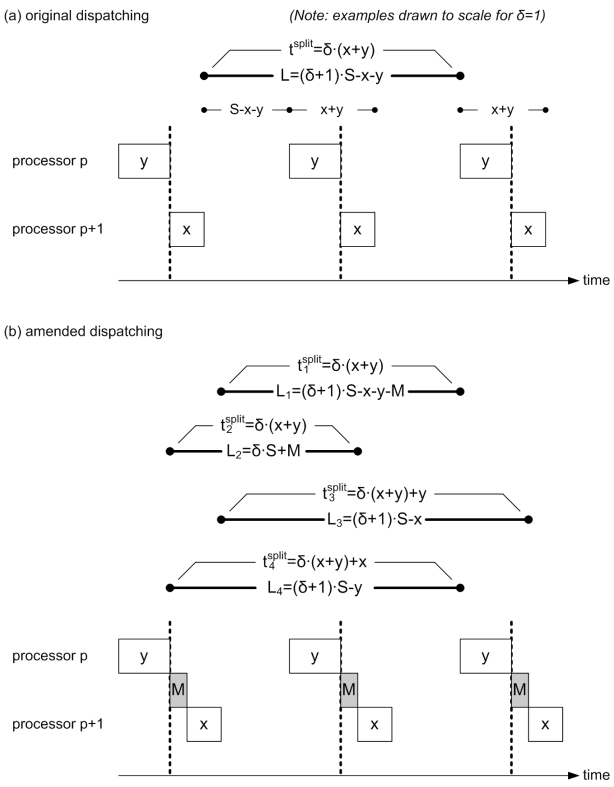

Figure 10: Candidate intervals for most unfavorable supply of processor time to a split task

Case 2: For it to hold that $\rho_2^{split} < \frac{C_i}{T_i}$ it has to be that

$$\begin{aligned} \frac{t_2^{split}}{L_2} < \frac{C_i}{T_i} &\overset{Lem.\ 3}{\Longrightarrow} \\ \frac{\delta \cdot (x+y)}{\delta \cdot S + M} < \frac{\delta \cdot (x+y)}{(\delta+1)\cdot S - x - y} & \\ \Rightarrow M > S - (x+y) = S - (\frac{C_i}{T_i} + 2 \cdot \alpha) \cdot S & \\ \Rightarrow M > S - \big((1 - 4 \cdot \alpha) + 2 \cdot \alpha\big) \cdot S = 2 \cdot \alpha \cdot S & \end{aligned} \tag{10}$$

which is impossible. Hence it holds that $\rho_2^{split} \geq \frac{C_i}{T_i}$.

Case 3: For it to hold that $\rho_3^{split} < \frac{C_i}{T_i}$ it has to be that

$$\begin{aligned} \frac{t_3^{split}}{L_3} < \frac{C_i}{T_i} &\overset{Lem.\ 3}{\Longrightarrow} \\ \frac{\delta \cdot (x+y) + y}{(\delta+1)\cdot S - x} < \frac{\delta \cdot (x+y)}{(\delta+1)\cdot S - x - y} &\Rightarrow \\ \frac{\Big(\delta \cdot (x+y)\Big) + y}{\Big((\delta+1)\cdot S - x - y\Big) + y} < \frac{\Big(\delta \cdot (x+y)\Big)}{\Big((\delta+1)\cdot S - x - y\Big)} &\Rightarrow \\ y < 0 & \end{aligned}$$

which is impossible. Hence it holds that $\rho_3^{split} \geq \frac{C_i}{T_i}$.

Case 4: Similarly as with Case 3 (swap places for x and y in the proof). Hence, it holds that $\rho_4^{split} \geq \frac{C_i}{T_i}$.

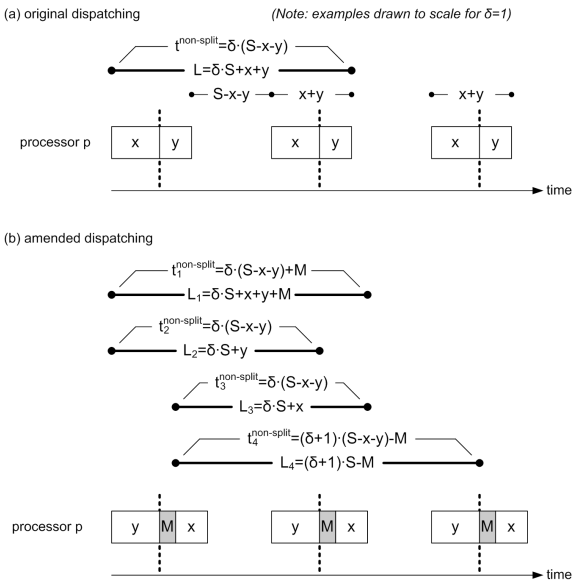

Figure 11: Candidate intervals for the most unfavorable supply of processor time to $\tau[p]$.

Thus, we have shown that the condition of Statement 9 (necessary for it to be possible for split task τ_i to miss its deadline) cannot be met. Hence, the schedulability of τ_i is unaffected by moving the start of its x-reserve, relative to the timeslot boundary, by an offset $M < 2 \cdot \alpha \cdot S$. ■

Thus the schedulability of split tasks is not affected by the offset M. We show the same for non-split tasks: To show this, we can reuse reasoning in our previously published paper [2] on the schedulability of non-split tasks (from Section 3.2, Observations), roughly from Equation 13 onwards).

By inspection, this reasoning is equally valid in the presence of an offset M up until Inequality 22, which is derived by application of Lemma 1. If the statement of Lemma 1 can also be shown to hold in the presence of an offset $M < 2 \cdot \alpha \cdot S$ for the x reserve on processor p, then it is possible to continue our reasoning from that point onwards and reach the same conclusion (i.e. that non-split tasks cannot miss deadlines).

Thus, to prove that non-split tasks remain schedulable, it suffices to prove the same statement as that of Lemma 1 in the presence of an offset $M < 2 \cdot \alpha \cdot S$ for the x-reserves.

Noting though that the left-hand side of the inequality in the statement of Lemma 1 corresponds to $\rho(\tau[p])$, what we need to prove can be equivalently formulated as:

Lemma 4. $\rho(\tau[p]) \geq \dfrac{\delta \cdot (S - x - z)}{\delta \cdot S + (x + z)}$

Proof: Fig 11(a) depicts (drawn to scale for $\delta = 1$) the time window for which $\rho(\tau[p])$ is observable for the set $\tau[p]$ of non-split tasks scheduled on processor p, in the absence of any spacing between its x and y reserves. However, if such a spacing is introduced (via an offset M for its x-reserve), there are four candidates for this "worst-case" time window. Each of them starts just as a reserve begins (either x or y) and ends just as a reserve terminates (again, either x or y). The four different combinations are shown in Fig 11(b), where each time window is annotated by its length (respectively, L_1 to L_4) and the respective amount of time ($t_1^{non-split}$ to $t_4^{non-split}$) for the execution of non-split tasks within it.

Let, $\rho_1^{non-split}$ to $\rho_4^{non-split}$ denote, for each respective case, a lower bound on the ratio of time usable for the execution of $\tau[p]$ (respectively, $t_1^{non-split}$ to $t_4^{non-split}$) to the overall interval length (respectively, L_1 to L_4). The bounds are derived by assuming that non-split tasks never get to execute within the reserves of any idle split tasks.

Since $\frac{t^{non-split}}{L}$ is minimal for one of the four candidate intervals (with the constraint that $L \geq \delta \cdot S$), it follows that

$$\rho(\tau[p]) = \min_{k \in \{1,2,3,4\}} \rho_k^{non-split} \stackrel{\text{def}}{=} \min_{k=1}^{4} \frac{t_k^{non-split}}{L_k} \tag{11}$$

We will see that in all cases, $\rho_k^{non-split} \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S + x + y}$:

Case 1: Assume that $\rho_1^{non-split} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$. Then:

$$\frac{t_1^{non-split}}{L_1} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y} \Rightarrow$$
$$\frac{\delta \cdot (S-x-y)+M}{\delta \cdot S+x+y+M} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y} \Rightarrow M < 0$$

which is impossible. Hence $\rho_1^{non-split} \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$.
Case 2: Assume that $\rho_2^{non-split} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$. Then:

$$\frac{t_2^{non-split}}{L_2} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y} \Rightarrow$$
$$\frac{\delta \cdot (S-x-y)}{\delta \cdot S+y} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y} \Rightarrow x < 0$$

which is impossible. Hence $\rho_2^{non-split} \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$.
Case 3: As in Case 2 (swap places for x and y). Hence $\rho_3^{non-split} \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$.
Case 4: Assume that $\rho_4^{non-split} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$. Then:

$$\frac{t_4^{non-split}}{L_4} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y} \Rightarrow$$
$$\frac{(\delta+1) \cdot (S-x-y)-M}{(\delta+1) \cdot S-M} < \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$$

Let λ denote $x+y$, for space considerations. Then:

$$\frac{(\delta+1) \cdot (S-\lambda)-M}{(\delta+1) \cdot S-M} < \frac{\delta \cdot (S-\lambda)}{\delta \cdot S+\lambda} \Leftrightarrow$$
$$\frac{\Big(\delta \cdot (S-\lambda)\Big)+(S-\lambda-M)}{\Big(\delta \cdot S+\lambda\Big)+(S-\lambda-M)} < \frac{\Big(\delta \cdot (S-\lambda)\Big)}{\Big(\delta \cdot S+\lambda\Big)} \Rightarrow$$
$$(S-\lambda-M) < 0 \Leftrightarrow x+y+M > S$$

which is impossible. Hence $\rho_4^{non-split} \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$.

Thus, it holds that $\rho(\tau[p]) \geq \frac{\delta \cdot (S-x-y)}{\delta \cdot S+x+y}$. ■

Having proven Lemma 4, suffices for proving that the schedulability of non-split tasks is not compromised, if an offset $M < 2 \cdot \alpha \cdot S$ is introduced for x-reserves.

APPENDIX B.
DISPATCHING ALGORITHM

We presented in Section III a new dispatchnig algorithm where the x-reserve on processor p does not start when the timeslot starts; instead the x-reserve on processor p starts $M[p]$ time units after the timeslot starts. Figure 12 shows a high-level overview this dispatching algorithm.

```
1. while TRUE do
2.   if first_iteration() then //some variables set just once
3.     p:=get_host_processor();
4.     hi_task:=get_hi_task(p); //split task run at timeslot end
5.     lo_task:=get_lo_task(p); //split task run at timeslot start
6.     y:=optimally_size_reserve(hi_task,p,S); //using Eq. 3 in [2]
7.     x:=optimally_size_reserve(lo_task,p,S); //using Eq. 4 in [2]
8.     M:=α · S
9.     N:=S-x-y-M
10.  end if
11.
12.  t:=(current_time()-t_boot) mod S;   //since timeslot start
13.
14.  if 0≤t<M then //within first reserve for non-split task task
15.    execute_non_split_task_with_earliest_deadline();
16.  end if
17.
18.  if M≤t<M+x then //within reserve for low split task
19.    if has_arrived_but_not_completed(lo_task) then
20.      execute_lo_task(p);
21.    else
22.      execute_non_split_task_with_earliest_deadline();
23.    end if
24.  end if
25.
26.  if M+x≤t<M+x+N then //not within a reserve of a split task
27.    execute_non_split_task_with_earliest_deadline();
28.  end if
29.
30.  if M+x+N≤t<S then //within reserve for high split task
31.    if has_arrived_but_not_completed(hi_task) then
32.      execute_hi_task(p);
33.    else
34.      execute_non_split_task_with_earliest_deadline();
35.    end if
36.  end if
37.
38. end while
```

Figure 12: A high-level overview of the new dispatching algorithm, which runs on every non-dedicated processor

APPENDIX C.
IMPLEMENTATION IN THE LINUX KERNEL 2.6.28

Based on the design principles in Section V, we have implemented the algorithm [2] in the Linux kernel 2.6.28. We refer to the algorithm in [2] as the Sporadic Multiprocessor Scheduler (SMS) and we refer to our implementation as Sporadic Multiprocessor Linux Scheduler (SMLS). To differentiate these tasks from other native Linux tasks present in the system, in this paper, these tasks will be referred as SMS tasks.

A. A primer on scheduling in the Linux kernel 2.6.28

The introduction of scheduling classes, in the Linux 2.6.23 kernel version, made the core scheduler quite extensible. The scheduling classes encapsulate scheduling policies and are implemented as modules [19]. These modules are hierarchically organized by priority in a linked list and the dispatcher will look for a runnable task of each module in a decreasing order priority. Currently, Linux kernel has three native scheduling modules: *RT* (Real-Time), *CFS* (Completely Fair Scheduling) and *Idle*.

The high resolution timers infrastructure were merged into Linux kernel 2.6.16. High resolution timers offer a nanosecond time unit resolution and are not dependent on the periodic ticks. These timers can be set on per-cpu basis and the callback can be executed in the hard interrupt context, which could avoid additional preemptions.

Red-black trees are balanced binary trees whose nodes are sorted by a key, the most operations are done in $O(\log n)$ time. Linux kernel has already implemented red-black tree.

B. How to implement slot-based task-splitting

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 to store information about the run-state of a process, for example, address space, list of open files, process scheduling class, just to mention some. All process descriptors are stored in a circular doubly-linked list. This linked-list is *not* the ready queue and hence there is no need to remove or insert elements in this list every time a process changes state; removal and insertion of elements in this linked list is only done when a new process is created and terminated.

To support the SMS algorithm some additional fields were added to this data structure, wrapped in the `sms_task_param`. Listing 5 shows the most important ones.

Each SMS task has a specific identifier, which is stored in the `task_id` field. Fields `cpu1` and `cpu2` are used to set the logical identifier of processor(s) in which the task will be executed on. Note that, according to the SMS algorithm each non-split task executes only on one processor, and each split-task executes on only two processors. In the former, these fields are set with the same identifier, in the latter, the split-task is executed on processors whose logical identifiers are defined by `cpu1` and `cpu2`. The relative deadline of each task is set on the `deadline` field of the `sms_rt_param` data structure. To manage the jobs, an SMS task uses the `sms_job_param` data structure. Note that, each SMS task is implemented as a Linux process and jobs are an accounting abstraction and the `nr` field is used for this purpose. For the absolute deadline and the release time of each job are used the `deadline` and the `release` fields, respectively.

In order to organize SMS tasks by the release time and by the absolute deadline time we defined two `struct rb_node node_job_release` and `struct rb_node node_non_split_task` fields.

Following design principle P1, each processor holds a run-queue of all runnable tasks assigned to it. The scheduling algorithm uses this run-queue to select the "best" process to be executed. In Linux, the information for these processes is stored in a per-processor data structure called `struct rq`. Listing 6 shows new data structures required by the SMS algorithm that were added to the Linux native `struct rq`.

Note that, each processor can be assigned at most two split tasks and many non-split tasks. As recommended on P2, `lo_split` and `hi_split` pointers are used to point to the split-tasks. The non-split tasks are organized in a red-black tree by the absolute deadline, whose root is `root_non_split_tasks`. The scheduler works as follows: if the current time falls in the x or in the y reserves, it selects `lo_split` or `hi_split` to be executed on, respectively; otherwise, it selects the non-split task with earliest absolute deadline from the red-black tree.

The original SMS algorithm defines at most two reserves x and y in the timeslot of each processor, and one split-task τ_i executes one portion on reserve y of the processor p and the other portion on reserve x of the processor $p+1$. Nevertheless, looking for two consecutive timeslots we realize that whenever a split-task finishes the execution on processor p, the task has to immediately resume execution on its reserve on processor $p+1$. As we have stated on Section III this is impossible in practice. So we divide the timeslot into four parts: M, x, N, and y. Note that, in spite of the length of the timeslot being equal to all processors, the timeslot composition is different on different processors. So, each processor stores in per-processor data structure the timeslot composition: `struct timeslot` (Listing 6). In order to get the control of the

```
struct task_struct {
 ...
 struct sms_task_param{
  int cpu1;
  int cpu2;
  struct sms_rt_param{
   unsigned long long deadline;
   ...
  };
  struct sms_job_param{
   unsigned long long deadline;
   unsigned long long release;
   atomic_t nr;
   ...
  };
  struct rb_node node_non_split_task;
  struct rb_node node_job_release;
 };
 ...
};
```

Listing 5: Fields added to `struct task_struct` kernel data structure

```
struct rq {
 ...
 struct split_task {
  struct task_struct *lo_split;
  struct task_struct *hi_split;
 }split_task;

 struct rb_root root_non_split_tasks;

 struct timeslot {
  unsigned long long begin_curr_timeslot;
  unsigned long long m;
  unsigned long long x;
  unsigned long long n;
  unsigned long long y;
  struct hrtimer timer;
 }timeslot;
 ...
};
```

Listing 6: Fields added to `struct rq` kernel data structure. There is one `struct rq` for each processor.

system and mainly to invoke the scheduler, the operating systems use a periodic timer interrupt mechanism (called *tick*). Since the period of the tick is usually set equal to $1\ ms$ it could not fit for the timing requirements of the systems. On the other hand, the length and the beginning of the each reserves must be multiple of the tick, which is a restriction and consequently reduces flexibility of the system and also imposes some restriction to schedulability analysis. To solve the identified problem we use a timer interrupt mechanism by which timer interrupts can be specified with nanosecond resolution. `struct timeslot` (Listing 6) data structure has also a variable called `struct hrtimer timer`, that is used to implement a timer mechanism that states the beginning of each timeslot part and in this way invoke the scheduler. `begin_curr_timeslot` variable states the beginning of the current timeslot, which is fundamental to synchronize the beginning of timeslot of all processors, as recommended in P3, and also to determine in which part of the timeslot a given time instant fall.

Additionally, a global variable called `timeslot_length` is used to update the local `begin_curr_timeslot` variable, as recommended on P3 and P4, and this way all processors are timely synchronized.

One of the most important feature of the real-time tasks is the periodicity. In order to achieve correct periods with small delays, as recommended on P5, we implemented in the kernel a job release mechanism. This mechanism is supported by red-black trees and also by high resolution timers and is triggered by the `delay_until` system call that will be described in Section C-C. All fields required for this mechanism are wrapped in the `struct job_release` (Listing 6) data structure.

To add a new scheduling policy to the Linux kernel it is necessary to create a new module. In this implementation, the SMS module was added on the top of the modules hierarchy, thus it is the highest priority module. Consequently, our system is hierarchically organized as it is shown in the Fig. 13.

According to the modular scheduling framework rules, each module must implement the set of functions specified in

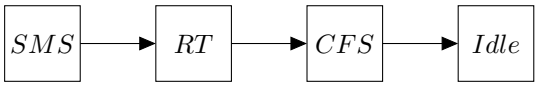

Figure 13: Priority hierarchy of scheduler modules

```
const struct sched_class sms_sched_class = {
 .next = &rt_sched_class,
 .enqueue_task = enqueue_task_sms,
 .dequeue_task = dequeue_task_sms,
 .check_preempt_curr = check_preempt_curr_sms,
 .pick_next_task = pick_next_task_sms,
 .task_tick = task_tick_sms,
 ...
};
```

Listing 7: `sms_sched_class` definition

the `sched_class` structure. Listing 7 shows the definition of `sms_sched_class`, which implements the SMS module. The first field (`next`) of this structure is a pointer to `sched_class` which is pointing to the `rt_sched_class` that implements the RT module.

The other fields are functions that act as callbacks to specific events.

The `enqueue_task_sms` is called whenever an SMS task enters in a runnable state (see Listing 8). This function must check if it is a non-split task or a split task. If it is a non-split task, it updates the absolute deadline and inserts it into the red-balck tree. In case of being a split task it checks if the current time falls within of the x or y reserve of the other processor that this task is split. If it falls an interprocessor interrupt signal is sent to the other processor to force a rescheduling.

When an SMS task is no longer runnable, then the `dequeue_task_sms` function is called that undoes the work of the `enqueue_task_sms` function (see Listing 9.

As the name suggests, `check_preempt_curr_sms` function, checks whether the currently running task must be preempted. This function is called following the enqueuing or dequeuing of a task and it only sets a flag that indicates to the scheduler core that the currently running task must be preempted.

`pick_next_task_sms` function selects the task to be executed by the current processor (see Listing 10).. This function is called by the scheduler core whenever the currently running task is marked to be preempted or when a task finishes

```
static void enqueue_task_sms(struct rq *rq, struct task_struct *p, int wakeup)
{

 ...
 if(p->sms_task_param.cpu1==p->sms_task_param.cpu2){ //it is a non splitted task
  p->sms_task_param.job.deadline=p->sms_task_param.job.release+p->sms_task_param.rt_param.deadline;
  insert_non_split_task(rq, p) ;
 }
 else{

  if(p->sms_task_param.cpu1==rq->cpu){
   r=get_cpu_timeslot_reserve(p->sms_task_param.cpu2);
   if(r==RESERVE_X){
    resched_cpu(p->sms_task_param.cpu2);
   }
  }
  else{
   r=get_cpu_timeslot_reserve(p->sms_task_param.cpu1);
   if(r==RESERVE_Y){
    resched_cpu(p->sms_task_param.cpu1);
   }
  }
 }
 ...

 return;
}
```

Listing 8: Code fragment of `enqueue_task_sms` function

```
static void dequeue_task_sms(struct rq *rq, struct task_struct *p, int sleep)
{
 ...
 if(p->sms_task_param.cpu1==p->sms_task_param.cpu2){ //it is a non splitted task
  remove_non_split_task(rq, p) ;
 }
 ...
}
```

Listing 9: Code fragment of `dequeue_task_sms` function

```
static struct task_struct *pick_next_task_sms(struct rq *rq)
{
 ...
 r=get_timeslot_reserve(rq);
 switch(r){
  ...
  case RESERVE_X:
   hi_split=get_lo_split_task(rq);
   if(lo_split!=NULL){
    task=cpu_curr(lo_split->cpu1);
    if(task==lo_split)
      goto pick_non_split_task;
    return lo_split;
   }
   else
    goto pick_non_split_task;
  break;
  case RESERVE_M:
  case RESERVE_N:
   goto pick_non_split_task;
  break;
  case RESERVE_Y:
   ...
  break;
 }
 return NULL;
pick_non_split_task:
 task=get_earliest_deadline(rq);
 if(task!=NULL)
  return task;
 return NULL;
}
```

Listing 10: Code fragment of `pick_next_task_sms` function

its execution. Assuming that, `get_timeslot_reserve` invocation returns `RESERVE_X`, which means that the current time instant falls in the x reserves. Then, the next step is to check if the split-task is in running state by invoking the `get_lo_split_task` function, which returns the pointer to split-task or `NULL`, if it is or if it is not, respectively. If it is not in running state, the earliest deadline non-split task is selected to execute on processor by invoking the `get_earliest_deadline` function. Note that if there is no non-split task ready to execute this function returns `NULL`. If it is on running state and if it is not the currently running task on processor p-1 (which is identified by the `cpu1` field) by invoking the `cpu_curr` function) then split- task is selected to execute on processor.

`task_tick_sms` function is mostly called from time tick functions. In the current implementation this function calls the `check_preempt_curr_sms` function, to check, if the current task must be preempted.

C. New system calls

System calls is one way provided by the Linux kernel to allow user space processes to access the kernel space. We introduce new system calls and modify the Linux native `sched_setscheduler` system call, to support the new scheduling policy identified by `SCHED_SMS`.

Initially, an SMS task is created as any task in the system, using the `fork` system call. After that, in order to be scheduled according to the SMS scheduling policy, the `sched_class` field of the `struct task_struct` has to be set with the address of `sms_sched_class` variable that implements the SMS scheduling algorithm in the kernel. The Linux native `sched_setscheduler` system call sets the scheduling policy and also scheduling parameters of the task. So, this system call was modified to support the new scheduling class, called `SCHED_SMS`, and also to set the specific SMS task parameters,

task_id, cpu1, cpu2 and deadline, just to mention some.

The purpose of some of the new system calls is self-explanatory from their names, like sms_enable_stats, sms_enable_trace, sms_set_timeslot_length and sms_set_begin_curr_timeslot.

The time precision of the job release is guaranteed by a timer interrupt mechanism supported by the struct job_release (see Listing 6) data structure. The idea is the following: the task is put in the waiting state until the release time and is inserted in a red-black tree ordered by the release time and also is set up a timer to expire at the release time. When the timer expires the task is removed from the red-black tree and its state is changed to running and consequently it becomes available to be selected by the scheduler to execute. However, in the implementation additional details must be into account. This procedure is composed by two functions: system call delay_until and timer callback wake_up_task.

This procedure is triggered by delay_until system call invocation (see Listing 11). In the Linux kernel, curr points to the current executing task on the processor. So, the first step is to set up the next release of the task. Then, it must be checked if the next release of the task is higher than the current time, returned by sms_clock function. If it is, no deadline miss occurred and 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. Otherwise, a deadline miss occurred and the task continues in the running state. If no deadline miss occurred, there is the need to disable interrupts in order to guarantee that the next code will be executed without any interruption. Then, the curr task is inserted in the red-black tree ordered by release time. Next, the next task release will be peeked from the red-black tree and if the next release is of the curr task, then the timer expiration must be set.

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, once from here until the end of the function some processing time will be necessary - then the timer is set up for the release of the task. Otherwise, the timer is set up to expire for the minimal time instant in future. After this, the interrupts are enabled again and the task must be relinquished from processor. For that, the state of the task is changed to TASK_INTERRUPTIBLE and schedule function is invoked.

When the release timer expires the wake_up_task callback is invoked. The first step is to get the task with the earliest release from the red-black tree. After that, a loop will be executed, to activate all tasks whose release fall in the current time plus a safe time interval (EPSILON). If there is at least one task on the tree, the timer is set up to expire according to the release of the task with earliest release present in the red-black tree.

Note that, if at least one task is woke up it is necessary to check if the currently running task need to be preempted. This is done by the check_preempt_curr_sms1 function, which algorithm is similar to the check_preempt_curr_sms function. In the end of the function we set up the release timer to the next task release, if there is at least one task, or for a large period that this timer will not expire.

D. How to use the new scheduler

Listing 12 and Listing 13 show the pseudo-code algorithm for the system boot and for the SMS task, respectively. The system boot program reads the task set configuration from a text file and computes all parameters: the timeslot length, the reserves of each processor and also task parameters. After that, using system calls, whose names starts by sms_, sets up the system and creates all tasks, using Linux native fork system call. Then, using exec system call it launches all SMS tasks.

Each SMS task with the arguments passed by the system boot program, sets the mask variable with the processor(s) allowed to execute this task and invokes the sched_setaffinity system call to tell to the scheduler which processor(s) is this task allowed to run on. After that, it sets param variable and invokes the sched_setscheduler system call to change the scheduling policy to SCHED_SMS and also to pass the required parameters of the task to the scheduler. Finally, the task will be in an infinite loop, where it is delayed until the release time instant, wake up at the release time instant, executes its work (using do_work function) and so on.

E. Getting data

The SMLS provides two mechanisms to get data concerning the execution of the scheduler. One of them, called stats, collects information about each job that reflects its “life” in the system and the other one traces all events in the system and it is called trace. Before describing these mechanisms let us describe how to get data from kernel space to user space. For that, we implemented a char device driver for each mechanism and a user space program. The char device implements a circular queue to collect the correspondent information and the user space program reads this information to an array. When the experiment finishes, the content of the array is written to a file. Each file is associated to one processor. These files are off-line parsed in order to get global results of the execution of the task sets. To avoid lock mechanisms the required front and rear indexes for the circular queue are manipulated using the atomic instructions.

```
asmlinkage long sys_sms_delay_until(const char __user * r)
{
 ...
 if(copy_from_user(&release_time,r,sizeof(unsigned long long))){
  return -EFAULT;
 }
 ...
 rq = cpu_rq(smp_processor_id());
 rq->curr->sms_task_param.job.release=release_time;
 if(release_time > sms_clock()){
  local_irq_disable();

  insert_job_release(rq, rq->curr);
  p=peek_erf_job_release(rq);
  now=sms_clock();
  if(p==rq->curr){
   if(p->sms_task_param.job.release > now + EPSILON2){
    set_job_release_timer_expires(rq,ns_to_ktime(p->sms_task_param.job.release));
   }
   else{
    set_job_release_timer_expires(rq,ns_to_ktime(now + EPSILON2));
   }
  }

  local_irq_enable();
  rq->curr->state=TASK_INTERRUPTIBLE;
  schedule();
  return 0;
 }
 else{
  now=sms_clock();
  ...

  if(rq->curr->sms_task_param.cpu1==rq->curr->sms_task_param.cpu2){ //it is a non splitted task
   if(rq->cpu==rq->curr->sms_task_param.cpu1){
    remove_non_split_task(rq,rq->curr);
    rq->curr->sms_task_param.job.deadline=rq->curr->sms_task_param.job.release+rq->curr->sms_task_param.rt_param.
        deadline;
    insert_non_split_task(rq, rq->curr);
   }
  }
  ...
  return -1;
 }
}
```

Listing 11: Task release procedure

1) `stats` *mechanism:* The `stats` mechanism collects information about each job. Listing 14 shows an excerpt of the file generated by `stats` mechanism for the task set presented on Section IV on processor P_2. In order to understand the contents of the Listing 14, first, let us take a look at Fig. 14 that illustrated the"life" of the job $\tau_{i,k}$ in the system.

The gray rectangles represent the time when job $\tau_{i,k}$ was executing on the processor. During this time, this job is referred to be the `current` task of the processor p. However, when the kernel code has to be executed on processor p the task continues being `current` task. This is what happen, when the functions that manage the periodic interrupt mechanism (called `Tick`) and when the scheduler core function and also when the functions that manage the arrival of the other SMS jobs have to be executed. So, in practice, the job is not executing at all that time. So, we consider that the measured execution time of a job $\tau_{i,k}$ ($meas\_C_{i,k}$) as the sum of all chunks of time when a job was the current task minus the time spent to execute: (i) Tick functions; (ii) the scheduler core functions to do the context swicth (`ctsw`) and (ii) the functions that manage the `arrival` of other SMS jobs. Thus, the execution time of job $\tau_{i,k}$ is the sum of the time represented by gray rectangles minus the time spent by the `tick1`, `tick2`, `ctsw1`, `ctsw2` and `ctsw3` and also minus $arrival_{j,x}$. We define the $meas\_Tick_{i,k}$ as the sum of all chunks of time spent to execute the Tick functions, the $meas\_CtSw_{i,k}$ as the sum of all chunks of time spent to do the context switch and the $meas\_Arrival_{i,k}$ as the sum of all chunks of time spent to wake up and insert on the ready queue other SMS jobs when the job $\tau_{i,k}$ was the current task. We also define $meas\_T_{i,k}$ as the interval between two consecutive job (of the same task) insertions on the ready queue.

Let us return our attention to the Listing 14. Each field is separated by comma. The first field is the number of the line (208), the second one is the identifier of the task (3) and the thirty one is the job number (130). The $meas\_C_{3,130}$ (3419387 ns) appears in the fourth and in the fifth, sixth and seventh the $meas\_Tick_{3,130}$ (3919 ns), $meas\_CtSW_{3,130}$

```
...
read_config_file();
compute_global_parameters();
sort_task_by_U();
compute_tasks_parameters(parameters);
compute_cpus_parameters();

//system calls

sms_enable_stats();
sms_enable_trace();

for(i=0;i<m;i++)
 sms_set_cpu_timeslot_composition(i,M,x,N,y);


sms_clock(&t);
sms_set_timeslot_length(S);
sms_set_begin_curr_timeslot(t);


for(i=0;i<n;i++){
 p=fork();
 if(p==0){
  exec(sms_task, parameters);
 }
}
...
```

Listing 12: SMS system boot algorithm

```
...

set_cpus_allowed(mask,cpu1,cpu2);
sched_setaffinity(0,mask);

param.cpu1=cpu1;
param.cpu2=cpu2;
param.deadline=deadline;
sched_setscheduler(0,SCHED_SMS,&param)

next_release = now() + period;
while (true)
{
 delay_until(next_release);
 do_work();
 next_release = next_release + period;
}
...
```

Listing 13: SMS task algorithm

(2990 ns) and $meas\_Arrival_{3,130}$ (0 ns), respectively. The eighth field states the $meas\_RT_{3,130}$ (5492414 ns) and the ninth field the $meas\_T_{3,130}$ (6499688 ns). The tenth field states the $meas\_J_{3,130}$ (516 ns) and the eleventh field the maximum $meas\_reserve\_J_{3,130}$ (1760 ns)

2) `trace` mechanism: The trace mechanism is an event oriented mechanism. It collects a set of events timely sorted. Let us take a look at Listing 15. All fields are separated by commas and the general format is: in the first field is identified the event and the on the second one the time (in ns) when the event happened. The remaining information is dependent of the collected event. For example, the first line of Listing 15 specifies that job 100 of the task 5 was assigned to the

```
...
208,3,130,3419387,3919,2990,0,5492414,6499688,516,1760,10,0
209,5,121,2933485,3840,2495,464,6163878,7001578,2298,1532,10,0
210,4,106,3910685,5382,7071,647,6824156,8001381,1940,1515,12,0
...
```

Listing 14: Excerpt of a file generated by `stats` mechanism

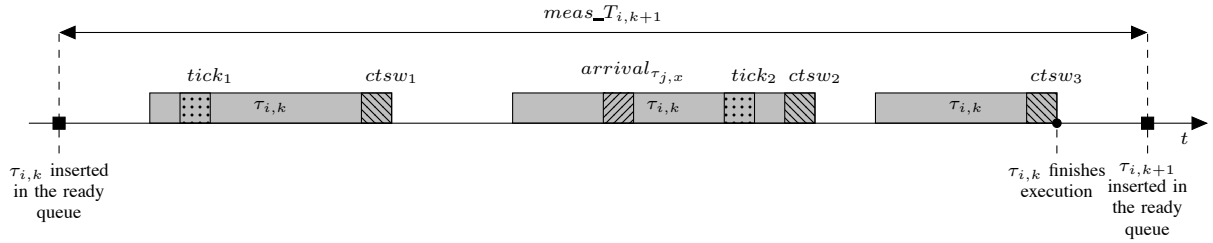

Figure 14: `stats` measurements

```
...
2,2797045528195,5,100
1,2797045552209,4,88,2797045550955
5,2797045751851,2797045751088
3,2797045753306,5,100
2,2797045754512,4,88
6,2797045821438,2797045820748
3,2797045822754,4,88
2,2797045824085,3,108
7,2797046147881,2797046147141
3,2797046148918,3,108
2,2797046149749,4,88
8,2797046772829,2797046772142
...
```

Listing 15: Excerpt of a file generated by `trace` mechanism

processor to be executed on it (event 2) at 2797045528195 ns. Next, some time later, at 2797045552209 ns, the job 88 of the task 5 was released on the system (event 1). However, it is 1254 ns late for release because it should be release at 2797045550955 ns (last field on the line). Event 5 specify the beginning of the M part of the timeslot. This event happened at 2797045751851 ns but it should happen at 2797045751088 ns, therefore with a drift of 763 ns. When a job relinquish the processor is classified as event 3 and the begin of x, N and y parts of the timeslot are classified as event 6, 7 and 8, respectively.