IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa

Bjorn Andersson

Eduardo Tovar

HURRAY-TR-110704

Version: 2

Date: 03-02-2011

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa, Bjorn 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

E-mail:

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

Abstract

Consider the problem of scheduling a set of sporadictasks on a multiprocessor system to meet deadlines using a tasksplittingscheduling algorithm. Task-splitting (also called semipartitioning)scheduling algorithms assign most tasks to justone processor but a few tasks are assigned to two or moreprocessors, and they are dispatched in a way that ensures thata task never executes on two or more processors simultaneously.A particular type of task-splitting algorithms, called slot-basedtask-splitting dispatching, is of particular interest because ofits ability to schedule tasks with 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 and propose some modifications to the slot-based task-splitting algorithm driven by implementation concerns, and we report the first implementation of this family of algorithms in a real operating system running Linux kernel version 2.6.34. Wehave also conducted an extensive range of experiments on a 4-core multicore desktop PC running task-sets with utilizations of up to 88%. The results show that the behavior of our implementation is in line with the theoretical framework behind it.

Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa*, Björn Andersson†* and Eduardo Tovar*
*CISTER-ISEP Research Center, Polytechnic Institute of Porto, Portugal
†Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
Email: *{pbs, baa, emt}@isep.ipp.pt, † baandersson@sei.cmu.edu

***Abstract*—Consider the problem of scheduling a set of sporadic tasks on a multiprocessor system to meet deadlines using a task-splitting scheduling algorithm. Task-splitting (also called semi-partitioning) scheduling algorithms assign 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 particular type of task-splitting algorithms, called slot-based task-splitting dispatching, is of particular interest because of its ability to schedule tasks with 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 and propose some modifications to the slot-based task-splitting algorithm driven by implementation concerns, and we report the first implementation of this family of algorithms in a real operating system running Linux kernel version 2.6.34. We have also conducted an extensive range of experiments on a 4-core multicore desktop PC running task-sets with utilizations of up to 88%. The results show that the behavior of our implementation is in line with the theoretical framework behind it.**

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 that have been very successful. These are taught at major universities world-wide, but are also incorporated in design tools as well as widely used in industry. Unfortunately, those results are quite limited to computer systems with a single processor.

Today, multiprocessors implemented on a single chip (called *multicore*) are the preferred platforms for many embedded real-time applications. This creates the pressing need for attaining an analogous toolkit for multicores. Such a toolkit should ideally exhibit the same properties as the one available for the uniprocessor case, which engineers appreciate much: (i) high utilization bounds; (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 when the deadlines are much shorter than the minimum inter-arrival times.

Researchers have been attempting to create real-time scheduling algorithms with those properties. *Global* scheduling algorithms store tasks in one global queue, shared by all processors. Tasks can migrate from one processor to another; that is, the execution of a task in one processor can be preempted and resumed on another processor. At any moment the m highest-priority tasks among those are selected for execution on m processors. Some algorithms in this class have an utilization bound of 100% but, unfortunately, they generate a large number of preemptions, with significant incurring overhead.

Another relevant set of algorithms is based on *partitioned* scheduling. In these, the task set is partitioned and all tasks in one partition are assigned to the same processor. Tasks cannot migrate from one processor to another and hence, when compared to global scheduling algorithms, these algorithms lead to fewer preemptions. Unfortunately, they also have a utilization bound of at most 50%.

There is however another family of real-time scheduling algorithms that exhibits all the above-mentioned properties: *task-splitting* or *semi-partitioning* algorithms [1], [2], [3], [4], [5], [6], [7], [8], [9]. The key idea behind these algorithms is that they assign most of the tasks to just one processor while 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 each timeslot 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. Among the types of task-splitting scheduling algorithms, this is the class that provides (in theory) 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 multicores with a very large number of processors. For these reasons, we believe an implementation of a slot-based task-splitting algorithm is a valuable and novel contribution to the state-of-the-art. We provide that in this paper.

Several multiprocessor scheduling algorithms have recently been developed and reported in the literature. We discuss next some of those related works. The Litmus$^{\mathrm{RT}}$ [10], [11],

[12] provides a modular framework for different scheduling algorithms (global-EDF, pfair algorithms) for the Linux kernel 2.6.34. In Kato *et al.* [13] the authors have also created a modular framework, called RESCH, for using other algorithms than Litmus$^{\mathrm{RT}}$ (partitioned, semi-partitioned scheduling) for the Linux kernel. In Faggioli *et al.* [14] the authors implemented global-EDF in the Linux kernel and made it compliant with POSIX interfaces.

Both the Litmus$^{\mathrm{RT}}$ and the POSIX compliant implementation do not support task-splitting algorithms. The framework by Kato *et al.* [13] shares however some of our goals in that it provides an implementation of task-splitting algorithms. But it uses another type of task-splitting algorithm (not slot-based) that does not guarantee that deadlines are met at higher processor utilizations. Hence, the current sate-of-the-art does not answer to the question of whether slot-based task-splitting multiprocessor scheduling can be implemented and work in practice or not. To this end, in this paper we discuss and propose an implementation of slot-based task-splitting multiprocessor scheduling algorithms. We show that they perform well in practice. Specifically, we implement in the Linux kernel 2.6.34 an algorithm based on slot-based split-task dispatching [2].

The remainder of this paper is structured as follows. Section II introduces the assumptions and system model. Section III overviews the idea of task-splitting. Section IVprovides the detailed background on the slot-based task-splitting scheduling algorithm. Section V proposes a new task-splitting algorithm. We show it is implementable in our framework, and those aspects of the implementation are discussed in Section VI. Section VII provides an evaluation of the approach and also discusses the overheads incurred by the implementation. Finally, in Section VIII conclusions are drawn.

II. Background

A. Assumptions

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

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

We assume that each processor has a local timer (see Fig. 1) providing two functions. One function allows reading the current real-time (that is not calendar time) as an integer. Another function makes it possible to set up the timer to generate an interrupt at x time units in the future, where x can be specified.

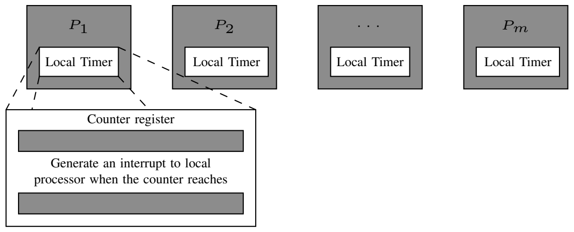

Fig. 1: Each processor (P_i) has a local timer.

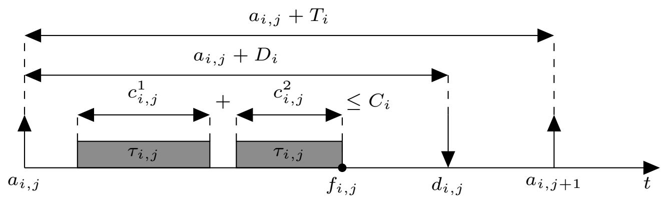

Fig. 2: Job timing parameters.

B. System model

We consider real-time systems composed by 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 its worst-case execution time, C_i, its minimum inter-arrival time, T_i, and by the time that can elapse until execution is completed, the relative deadline D_i. We assume $0 \leq C_i \leq D_i$. If we do not state D_i then we assume that $\forall i : D_i = T_i$.

The utilization of task τ_i, denoted as u_i, is defined as:

$$u_i = \frac{C_i}{T_i}$$

The system utilization, U_s, is defined as:

$$U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} u_i$$

Each job $\tau_{i,j}$ (this notation means the j^{th} job of task τ_i) becomes *ready* to be executed at arrival time ($a_{i,j}$) and continues until *finishing* (or completion) time ($f_{i,j}$).

The duration of this time interval is said to be the *response time* ($rt_{i,j} = f_{i,j} - a_{i,j}$) of the job $\tau_{i,j}$, and the response time (RT_i) of task τ_i is defined as being the maximum response time of all its jobs ($RT_i = \max_{j=1}^{k}(r_{i,j})$).

The *absolute deadline* ($d_{i,j}$) of job $\tau_{i,j}$ is given as $d_{i,j} = a_{i,j} + D_i$ and a deadline miss occurs when $f_{i,j} > d_{i,j}$.

The time difference between two consecutive job releases must be at least equal to T_i. Fig. 2 illustrates the relation among the timing parameters of the job $\tau_{i,j}$. The execution of the job $\tau_{i,j}$ is represented by a gray rectangle and the sum of all execution chunks ($c^x_{i,j}$) must be equal to C_i.

III. Task-splitting

Consider $n = m + 1$ tasks with $T_i = 1$ and $C_i = 0.5 + \epsilon$ (where ϵ is a positive number smaller than 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 that is assigned at least two tasks. Since on that processor the utilization exceeds 100%, 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 would be possible to meet deadlines. Take the following 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 P_1 and P_2) so that a job by τ_{m+1} executes $0.25+\epsilon/2$ units on one of the two processors and $0.25 + \epsilon/2$ 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. 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 $D_i \geq 2T_i$, and so its application is confined to scheduling 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 at the beginning of the timeslot and the second piece is executed at the end of the timeslot. This provides hard real-time scheduling with $D_i = T_i$, it allows good utilization bounds to be attained and it provides bounds on the number of preemptions. However it works only for periodic tasks. Levin *et al.* [16] variation able 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 those algorithms are difficult to implement in practice. Kato and Yamasaki [8] proposed a suspension-based task-splitting 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 processors simultaneously. The two approaches for task-splitting dispatching that we believe are the most promising for practical implementation are the *job-based task-splitting dispatching* [6], [9] and the *slot-based task-splitting dispatching* [2]. The former splits a job into two or more subjobs, forms a sequence of subjobs and sets the arrival time of a subjob equal to the absolute deadline of its preceding subjob. It provides an utilization bound greater than 50% and few preemptions. It has been implemented in a real operating system and through experimental studies [13] that implementation was found to outperform many other non-split approaches. The main drawback of job-based task-splitting dispatching is that utilization bounds greater than 69% have not been attained [9].

Slot-based task-splitting dispatching subdivides time into equal-duration timeslots. The beginning and end of each timeslot are synchronized across all processors. The end of a timeslot of processor p contains a reserve and the beginning of a timeslot of processor $p + 1$ contains a reserve, and these two reserves supply processing capacity for a split task.

Slot-based task-splitting dispatching causes more preemptions than job-based task-splitting dispatching but, in return, it offers higher utilization bounds (higher than 69% and configurable for up to 100%) [2].

IV. Slot-based task-splitting

In this section the slot-based scheduling algorithm [2] is detailed. Before doing it so let us provide some important definitions. For convenience we define:

$$\text{TMIN} = \min(T_1,\ T_2,\ ...,\ T_n)$$

The behavior of this scheduling algorithm depends on a design parameter δ. Based on this parameter, the following definitions were made:

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

and

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

and

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

S is the duration of the timeslot. α is a parameter used for sizing the reserves. SEP is a threshold that defines the utilization bound of the scheduling algorithm.

To provide a better understanding of the slot-based task-splitting scheduling algorithm [2], let us consider an example. Consider a system with 4 processors ($m = 4$) and 7 tasks ($n = 7$) as specified by Table I. Time units are intentionally omitted in Table I since they are not important for understanding the algorithm. Let $\delta = 4$ which means that the SEP is 88.85%. This kind of scheduling algorithm can be divided into two separate algorithms: an offline algorithm for task assignment and an online dispatching algorithm.

A. Tasks Assigning Algorithm

Tasks whose utilization exceed SEP (henceforth called *heavy tasks*) are each assigned to a dedicated processor. The remaining tasks are assigned to the remaining processors in a manner similar to next-fit bin packing [17]. Assignment is done in such a manner that the utilization of processors is exactly SEP. Task splitting is performed whenever a task causes the utilization of the processor to exceed SEP. In this case, this task is split between the current processor p and the next one p + 1.

The task assignment algorithm works as follows. Since τ_1 is a heavy task it is assigned to a dedicated processor (P_1). 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

Task	C	T	u
τ_1	9	10	0.9000
τ_2	7	12	0.5833
τ_3	7	13	0.5385
τ_4	8	16	0.5000
τ_5	6	14	0.4286
τ_6	6	16	0.3750
τ_7	3	17	0.1765

TABLE I: Task set example.

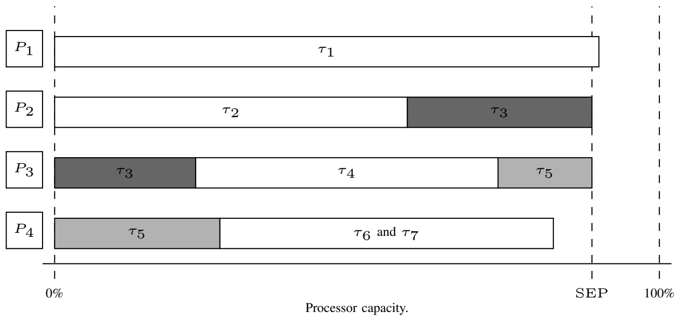

Fig. 3: Tasks assignment to processors.

of processor P_2 equal to SEP; that is, 0.3052. This part is referred as $hi\_split[P_2]$ and the remaining portion (0.2333) of task τ_3 is assigned to processor P_3, which is referred as $lo\_split[P_3]$. Fig. 3 shows the final task set assignment to the processors. From that figure the following can be observed: (i) processor P_1 is a dedicated processor executing only task τ_1; (ii) tasks τ_2, τ_4, τ_6 and τ_7 (henceforth called *non-split tasks*) execute on only one processor and (iii) tasks τ_3 and τ_5 are split tasks.

The processors P_2 and P_3 have been assigned split tasks, have time windows (called *reserves*) where these split tasks have priority over non-split tasks assigned to those processors. The length of the reserves is chosen such that no overlap occurs. The split task can be scheduled and all non-split tasks can meet deadlines.

Time is divided into timeslots of length S and *non-dedicated* processors (those that execute more than one task) usually execute split and non-split tasks. For that, the timeslot might be divided into three parts. The first x time units are reserved for executing the $lo\_split[p]$. The last y time units are reserved for executing the $hi\_split[p]$. The remaining part of the timeslot (henceforth denoted as $N[p]$) is used to execute non-split tasks and is computed as follows:

$$N[p] = S - x - y$$

Reserves x and y for each split task must be sized such that $\frac{x+y}{S} = \frac{C_i}{T_i}$. Depending on the phasing of the arrival and deadline of τ_i relative to timeslot boundaries, the fraction of time available for τ_i between its arrival and deadline may differ from $\frac{x+y}{S}$, since a split task only executes during the reserves. Consequently, it is necessary to inflate reserves by α in order to always meet deadlines:

$$x = S \cdot (\alpha + lo\_split[p])$$

and

$$y = S \cdot (\alpha + hi\_split[p])$$

B. Dispatching Algorithm

On a dedicated processor, the dispatching algorithm is very simple: whenever there is one task ready to be executed, the processor executes it. On a non-dedicated processor, the dispatching algorithm works over timeslot of each processor and whenever the dispatcher is running, it checks to find the time elapsed in the current timeslot. If the current time falls within a reserve (x or y) and if the assigned split task is ready to be executed, then the split task is scheduled to be executed on the processor. Otherwise, the ready non-split task with the earliest deadline is scheduled to be executed on the processor. If the current time does not fall within a reserve, the ready non-split task with the earliest deadline is scheduled to be executed on the processor. Otherwise, if there is no ready non-split task ready to be executed then no task is selected, i.e., processor remains idle.

Fig. 4 shows a simplified execution timeline. The timeslot length is $S = 2.5$. In the execution timeline that is presented in Fig. 4, it is assumed that each task is activated only once. We also assume that the release time of all tasks is at time instant zero. The execution of the tasks is represented by rectangles labeled with the task's name. A black circle indicates the end of execution of a task. As it can be seen, the split tasks execute only within reserves (marked x and y). For instance, task τ_3 on processor P_2 executes only on reserves. Outside its reserves it does not use the processor, even if the processor is idle. In contrast, non-split tasks execute mainly outside the reserves but potentially also within the reserves, when there is no split task ready to be executed. There are two clear situations in Fig. 4 that illustrate this. First (marked a), task τ_7 executes at the beginning of the timeslot, which begins at 12.5, because the split task τ_5 has finished its execution on the previous timeslot. Second (marked b), split task τ_5 finishes its execution a bit earlier than the end of its reserve (that finishes at 12.5) and hence there is some available time on the reserve, which is used by non-split task τ_4.

V. Slot-based task-splitting suited for implementation

As it intuitive from observing two consecutive timeslots in Fig. 4, whenever a split task consumes its reserve on processor p, that task has to immediately resume execution on its reserve on processor p + 1. Due to many sources of unpredictability (e.g. interrupts) in a real world operating system this precision is not possible. Consequently, this can prevent the dispatcher of processor p + 1 to select the split task because processor p has not yet relinquished that task, which can imply more overhead and more preemptions. This can be avoided if the reserve on processor p + 1 is available some time units later.

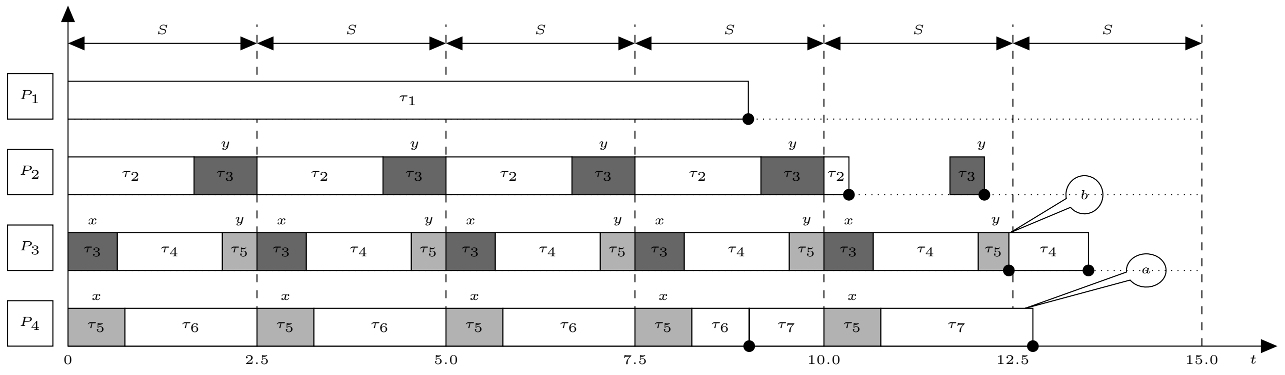
Fig. 4: Task set example execution timeline.

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 the reserve. For the dispatching algorithm in [2], it holds that $\forall p : M[p] = 0$. In order to implement slot-based task-splitting dispatching, we need to choose $\forall p : M[p] > 0$. We will now discuss how to choose $M[p]$.

From our previous work [2], we know that adding the duration of x and y reserve on the same processor will give at most $(1 - 2 \cdot \alpha) \cdot S$. We also know from that previous work that 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:

$$\forall p : M[p] = \alpha \cdot S$$

This choice 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.

Consider the example in Fig. 4 again, it is intuitive that all processors are simultaneously required to take scheduling decisions in the beginning of each timeslot. Taking scheduling decisions is not a problem. But executing the selected tasks could be if there is the need to fetch instructions code from main memory, which, as is known implies bus contention to access memory. This can however be avoided if the timeslots are not aligned; that is, if they are staggered by $M[p]$. Fig. 5 illustrates the proposed approach to make it practical the implementation of the slot-based task-splitting.

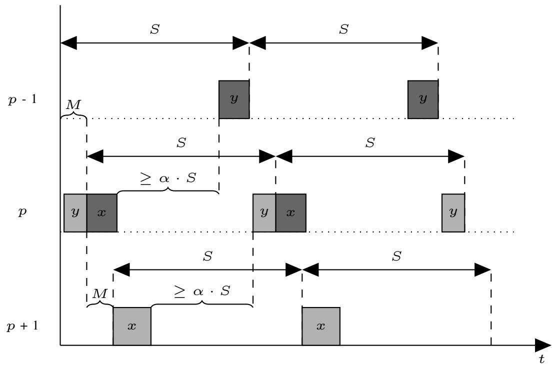
Fig. 5: Staggered timeslots.

VI. OS SUPPORT FOR IMPLEMENTING SLOT-BASED TASK-SPLITTING ALGORITHM

We are now able to introduce a set of design principles required to implement the slot-based task-splitting scheduling algorithm [2] in real operating system. The five design principles are as follows:

P1. Each processor should have its own run-queue (the queue that stores tasks that 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 low-time complexity for insertion, removal and searching operations.

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 of the task that is split between them. If no such task exists then these pointers are `NULL`.

P3. Each processor should have a variable called `begin_curr_timeslot`. It should hold a time value that is no larger than the current time, and it should never be less than current time minus S (timeslot length). The variable `begin_curr_timeslot` should be incremented by S to ensure this.

P4. Each processor should have a timer-queue to store the events that are known to happen at a later point in time. 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 resulting from finite speed of the processor does not occur (see page 38 in [18] for further discussion).

P5. The operating system should implement a `delay_until` system call (see page 38 in [18] for further details), 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].

The standard Linux kernel 2.6.34 was chosen to implement the scheduling algorithm proposed in [2] with the modifications as suggested in Section V since that kernel version provides the required tools to satisfy the previously mentioned design principles: (i) each processor holds its own run-queue and it is easy to add new fields to it; (ii) it has already implemented red-black trees that are balanced binary trees whose nodes are sorted by a key and most the operations are done in $O(\log n)$ time; (iii) it has the high resolution timers infrastructure that offers a nanosecond time unit resolution and timers can be set on a per-CPU basis; (iv) it is very simple to add new system calls and, finally, (v) it comes with the modular scheduling infrastructure that easily enables adding a new scheduling policy to the kernel.

Real-time systems require predictability, but, unfortunately, the standard Linux kernel cannot provide such predictability. There are many sources of unpredictability in a Linux kernel including the following: (i) interrupts one are the events with the highest priority, consequently when one arises the processor execution switches to handle the interrupt (usually interrupts arise in an unpredictable fashion); (ii) on Symmetric Multi Processing (SMP) systems there are multiple kernel threads running on different processors in parallel, and those can simultaneously operate on shared kernel data structures requiring serialization on access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions; (iv) the high resolution timer infrastructure is based on local Advanced Programmable Interrupt Controller (APIC), disabling and enabling local interrupts can disrupt the precision of that timer and, finally, (v) the hardware that Linux typically runs on does not provide the required determinism, which would mean that the timing behavior of the system would be predictable with all latencies being time-bounded and known prior to run-time.

Real-time operating systems (RTOSs) use several approaches to solve the above mentioned drawbacks. However, most of RTOSs consist on simple modifications of standard Linux kernels [19], [20]. And the great variety of RTOSs suggests that there is no consensus about an implementation for RTOSs. Therefore, we believe that if the scheduling algorithms we propose in this paper work properly in a standard Linux kernel version, then it could also be implemented and work properly in other RTOSs versions.

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

We added a new scheduling policy (called *SMS* that stands for sporadic multiprocessor scheduling) module on top of the native Linux module hierarchy, thus it is the highest priority module.

VII. Experimental Evaluation

We have implemented SMS in the Linux kernel 2.6.34 based on the design principles reasoned out in the previous sections (see [21] for further details). We are interested in experimentally assessing whether the behavior of our implementation deviates or not from the theory (considering the assumptions as reasoned in Section VI). For that purpose we first introduce the concepts used to characterize the discrepancy between theory and practice as well as the mechanism used for getting data from the kernel space to the user space. We will stress-test the implementation with tasks of small T_i, making implementation overheads significant so that deadline misses could occur. We will also run experiments with "normal" task sets and demonstrate that our implementation works well.

A. Concepts used to characterize the outcome of the experiments

The evaluation of the discrepancy between theory and practice is especially based on two metrics. One of them is specific to the slot-based task-splitting scheduling algorithms (*reserve jitter*), and the other one is related to the mechanism implemented for releasing jobs (*release jitter*). Next, we define these and other metrics.

Fig. 6 shows $meas\_ResJ_{i,k}$, which represents the *measured reserve jitter* of job $\tau_{i,k}$ and denotes the discrepancy between the time when the job $\tau_{i,k}$ should (re)start executing (at the beginning of the reserve A, where A could be x, N or y) and when it actually (re)starts. It should be mentioned that the timers are set up to fire when the reserve should begin, but, unfortunately, there is always a drift between this time and the time instant at which the timer fires. Then, the timer callback executes and, in most cases, sets the current task to be preempted triggering the invocation of the dispatcher. The dispatcher selects a task according to the dispatching algorithm and switches the current task by the selected task.

$meas\_RelJ_{i,k}$ (*measured release jitter* of job $\tau_{i,k}$) denotes the difference in time from when the job $\tau_{i,k}$ should arrive until it is inserted in the ready queue (see Fig. 7).

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

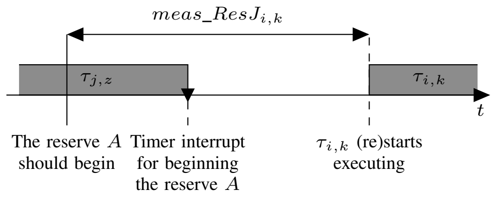
$meas\_ResJ_{i,k}$
$\tau_{j,z}$
$\tau_{i,k}$
t
The reserve A should begin
Timer interrupt for beginning the reserve A
$\tau_{i,k}$ (re)starts executing

Fig. 6: Measured reserve jitter.

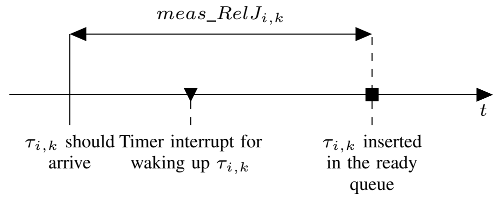
$meas\_RelJ_{i,k}$
t
$\tau_{i,k}$ should arrive
Timer interrupt for waking up $\tau_{i,k}$
$\tau_{i,k}$ inserted in the ready queue

Fig. 7: Measured release jitter.

reserve jitter of each task τ_i as follows:

$$meas\_ResJ_i = \max_{k=1 \ldots meas\_Nr\_jobs_i}(meas\_ResJ_{i,k})$$

and the maximum reserve jitter of each task set τ as follows:

$$meas\_ResJ_\tau = \max_{i=1 \ldots n}(meas\_ResJ_i)$$

Analogously we use the notations $meas\_RelJ_i$ and $meas\_RelJ_\tau$.

B. Mechanisms used to fetch data from the kernel

Two mechanisms were implemented 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. Another one traces all events in the system and is therefore called `trace`. To get data from the kernel space to the user space these mechanisms use device drivers. The `char` device implements a circular queue to collect the concerned information and an user space program (running a process by each processor) reads that information. When the experiment finishes, the collected information is written to a file. Each file is associated to one processor.

C. Experimental setup

The experimental machine is equipped with one Intel®i7 ®(QuadCore) at 2.67 GHz processor and 4GB of main memory. The experiment was conducted on the Linux kernel 2.6.34 running on runlevel 1 with all interrupts managed by the fourth core. We also disabled the network connection and the journal mechanism of the filesystem. The main idea of this setup is to create a running environment with few sources of unpredictability.

In order to experimentally assess whether the behavior of our implementation deviates or not from the theory we have considered two types of task sets that are described next.

1) Controlled task set: 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 each processor (U_{target}). With these parameters we compute the utilization of each task (u_i) in ascending order as follows:

$$meas\_ResJ_i = \max_{k=1 \ldots meas\_Nr\_jobs_i}(meas\_ResJ_{i,k})$$

$$\underset{i=1 \ldots n}{u_i} = i * (U_{target} * m)/(\sum_{i=1}^{n} i)$$

and in descending order, as follows:

$$\underset{i=1 \ldots n}{u_i} = (n - 1 + i) * (U_{target} * m)/(\sum_{i=1}^{n} i)$$

We are also able to compute the average of u_i (UAVG) as follows:

$$\text{UAVG} = (U_{target} * m)/n$$

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:

$$\underset{i=1 \ldots n}{T_i} = \text{TMIN} + (i-1)/(n-1) * (\text{TMAX} - \text{TMIN})$$

Finally, C_i is derived as follows:

$$\underset{i=1 \ldots n}{C_i} = T_i * u_i$$

For convenience we let UMIN denote the minimum u_i and UMAX the maximum u_i of a task set. Using the above formulas we can create a set of task sets that we think could be used to deeply assess the slot-based task-splitting scheduling algorithm features. This way we created three types of task sets where: (i) u_i and T_i are both set in ascending order, thus the task with TMIN is the task with UMIN (henceforth referred as UMIN2UMAX); (ii) u_i is set in descending order and T_i is set in ascending order, thus the task with TMIN is the task with UMAX (henceforth referred as UMAX2UMIN) and, finally, (iii) all tasks have u_i set equal to UAVG and T_i is set in ascending order (henceforth referred as UAVG2UAVG).

Concerning the periodicity we defined two types of task sets: periodic and sporadic. We keep the ascending order of T_i for both, but in the periodic task set the T_i of each task τ_i is fixed for all jobs, while in the sporadic task sets the T_i randomly varies between T_i and $T_i * f$. Therefore, combining these two types results into six types of task sets.

All task sets were generated with $m = 4$ and $\delta = 4$. The U_{target} was set equal to 0.888 meaning that all processors (except the last) have a workload of approximately 0.8885. The criterion used to stop the experiment was the number of jobs, thus, when a task has released one million of jobs, that ends the experiment. The time duration of each experiment was greater or equal to 50 minutes and all experiments took approximately 20 hours.

First, we considered a task set with 8 tasks (the explanation for number 8 is: using the above method for creating task sets, 8 is the minimal number of tasks that avoid heavy tasks and we did not waste a processor executing only one task).

Parameters TMIN and TMAX were set equal to 3 and 5 ms, respectively. We believe that 3 ms for TMIN is an interesting value because the host computer is a "normal" computer without any special feature and S is equal to 750 μs, which is less than one tick duration (HZ Linux kernel variable was set equal to 1000). With TMAX equal to 5 we created task sets where the difference between TMAX and TMIN is 2 ms (a very small value). For the sporadic task sets f was set equal to 1.5. We will refer to this set of experiments (composed by six task sets) as Exp_1.

Note that, with this combination of task sets we have created some extreme assignments. For instance, some processors execute only split tasks and some reserves are very small (see Fig. 8).

For the second type of task sets we multiplied the number of tasks (8) by the number of processors (4). So we have task sets with 32 tasks and TMIN and TMAX were set equal to 3 and 5 ms, respectively. There is no special reason for this number but we think that this is a considerable number of tasks for 4 processors. Fig. 9 shows the task sets assignment to processors. This set of experiments will be referred as Exp_2.

A further pool of experiments was conducted where the TMAX parameter was set to 50 ms. We keep the number of tasks, and the set of experiments with the number of tasks equal to 8 will be referred to as Exp_3 while Exp_4 will refer to the set of experiments with 32 tasks. In this way we get periods for tasks differing in more than 1 ms (the length of *tick*).

2) Random task set: In order to generate the random task sets we have to define the number of processor (m), the target utilization of each processor (U_{target}), the minimum (UMIN) and the maximum (UMAX) utilization, and also the minimum (TMIN) and the maximum interarrival time (TMAX). Listing 1 shows the algorithm used to generate random task sets.

```
...
n=0;
SumUi=0.0;
while(1){
 factor = (double)rand() / RAND_MAX;
 u = UMIN + factor(UMAX - UMIN);
 if((SumUi + u) <= (m * Utarget)){
  Ui[n++] = u;
  SumUi = SumUi + u;
 }else{
  break;
 }
}
for(i = 0;i < n;i++){
 factor = (double)rand() / RAND_MAX;
 Ti[i] = TMIN + factor(TMAX - TMIN);
}
...
```

Listing 1: Algorithm for creating random task sets.

All task sets were generated with m equal to 4 and U_{target} was set equal to 0.888 (δ parameter was set equal to 4 and this way SEP was 88.85%). The criterion used to stop the experiment was the number of jobs, thus, when a task has released one hundred of thousands jobs the experiment finishes.

We generated task sets where TMIN varied from 5 ms up to 50 ms. TMAX was set for all task sets equal to 100 ms. UMIN was set for all task sets equal to 0.01. UMAX varied from 0.1 up to 0.51. Using these parameters we got task sets where the number of tasks varied from 12 up to 67. We also ran these task sets as periodic and also as sporadic task sets (using the same method that we used for controlled task sets).

The time duration of the experiment varied from 5 minutes to 80 minutes, in a total of approximately 17 hours. This set of experiments will be referred to as Exp_5.

D. Experimental results

Before analyzing the results presented in Table II let us denote MAX, AVG and STDEV as being the maximum, the average and the standard deviation of the values observed of all task sets in each experiment. Recall that each experiment is composed by a set of several task sets. The main goal of these three metrics is: (i) to highlight the worst-case value (MAX), which likely is due to unpredictability of the system (mentioned on Section VI) and (ii) using the other two metrics (AVG and STDEV) to show the trend and the variation from the average.

Exp.	$meas\_RelJ$			$meas\_ResJ$			Dead. miss
	MAX	AVG	STDEV	MAX	AVG	STDEV	
Exp_1	11.77	6.65	1.67	**33.06**	3.52	5.17	Yes
Exp_2	11.48	7.35	1.55	3.58	2.07	0.16	No
Exp_3	6.88	5.17	1.57	**53.73**	4.46	9.96	No
Exp_4	10.82	6.90	1.41	4.47	2.42	0.32	No
Exp_5	8.72	5.83	0.70	3.98	2.55	0.38	No

TABLE II: Experimental results (μs time unit).

According to the values presented in Table II the release jitter ($meas\_RelJ$) is not influenced by neither the number of tasks nor by the periodicity. However, the results of the reserve jitter ($meas\_ResJ$) in most cases tend to be uniform, but the maximum values of Exp_1 and Exp_3 experiments are too high in comparison to the others. These values could cause some disruption on the behavior of the algorithm since some reserves could be skipped (the minimum length of a reserve is given by $M[p] = \alpha \cdot S$, which is 20.9 μs in this case). However, the implementation has mechanisms to deal with this type of events. These maximum values do not reflect the trend as we can realize by the AVG and STDEV values and also by the results reported in Fig. 10 and Fig. 11. As it can be seen, those are extreme values that are due to the unpredictability of the system. Furthermore, the last values were experienced by split-tasks that execute on the last processor (and this processor manages all interrupts).

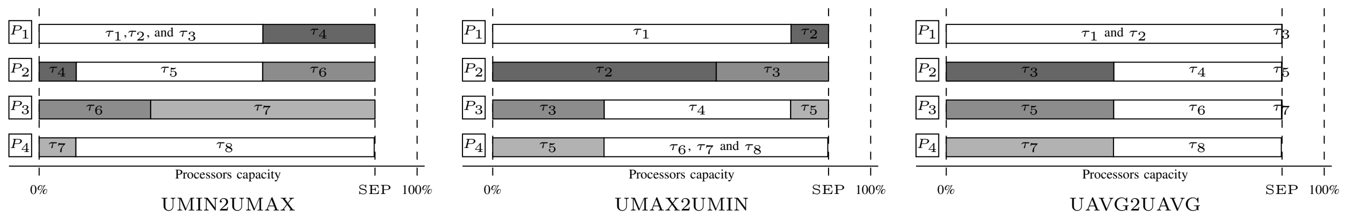

Fig. 8: Task sets with 8 tasks: assignment to processors.

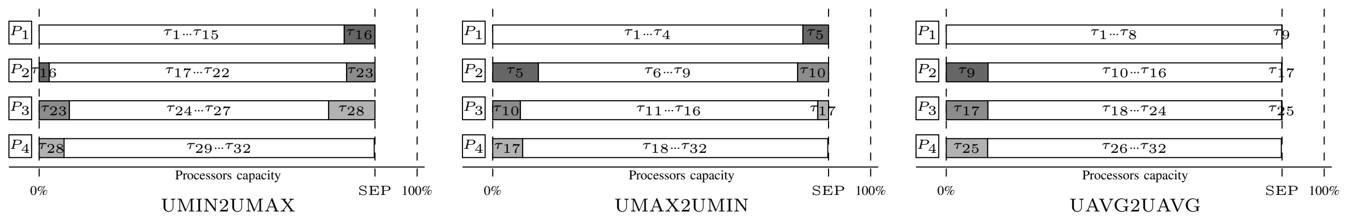

Fig. 9: Task sets with 32 tasks: assignment to processors.

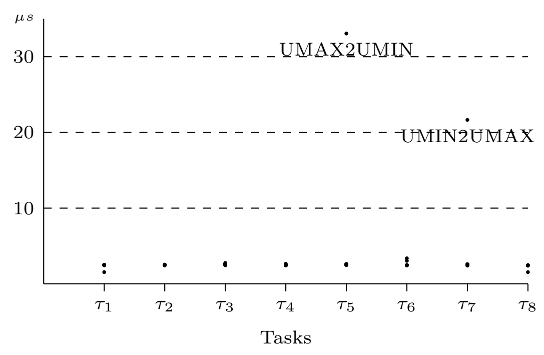

Fig. 10: $meas\_ResJ_i$ of experiment Exp_1.

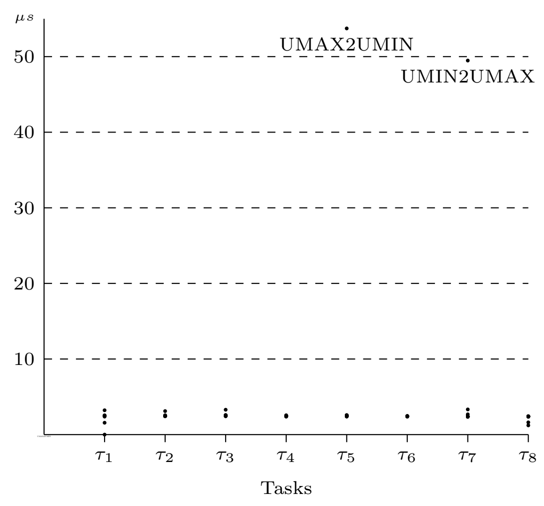

Fig. 11: $meas\_ResJ_i$ of experiment Exp_3.

Table II also shows that in Exp_1 some jobs missed the deadline, namely, three jobs of task τ_2 on sporadic experiment UMAX2UMIN and also three jobs of task τ_3 on sporadic experiment UAVG2UAVG. Note that 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 `delay_until`) and when $\tau_{i,k}$ finishes its execution. The absolute deadline of a job is set in the same way. A deadline miss of a job $\tau_{i,k}$ does not alter the time when job $\tau_{i,k+1}$ should arrive. These are split tasks and they experience more overheads since they are preempted more times than non-split tasks. But there is an important concern that could lead to some additional delay related to the split tasks. Split tasks are released on both processors that they are assigned to, but at each time in only one processor. Due to this it could happen that a split task could be released by processor p and that time instant could coincide with the reserve for that task on processor $p+1$. In this case, processor p enqueues that task and sends an inter processor interrupt to processor $p+1$, which could lead to some delay on the execution of that task. Note that τ_2 on sporadic experiment UMAX2UMIN is a task with high u_i and small T_i. However, we re-run again the same experiments and we did not get any deadline miss.

In spite of some extreme conditions the results show a good correspondence between theory and practice. Some extreme values were obtained but they are related to the underlying operating system unpredictability. Thus, we can conclude that this kind of algorithm can be implemented in a real operating system and perfoms well.

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] with some modifications in the Linux kernel 2.6.34. We have conducted a range of experiments with a 4-core multicore desktop PC utilized to 88% with real-time tasks executing empty for loops that took approximately 37 hours. In spite of the unpredictability of the Linux kernel we observed a good correspondence between theory and practice.

These good results are due to: (i) the controlled experimental environment (stated in Section VII-C); (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.

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 could 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.

Acknowledgements

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

References

[1] 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 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] 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.

[11] 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.

[12] 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.

[13] 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.

[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] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," in *Approximation algorithms for NP-hard problems*. Boston, MA, USA: PWS Publishing Co., 1997, pp. 46–93.

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

[19] Xenomai. (2011, Feb.) Real-time framework for linux. [Online]. Available: http://www.xenomai.org/

[20] RTAI. (2011, Feb.) Realtime application interface for linux. [Online]. Available: https://www.rtai.org/

[21] P. B. Sousa, B. Andersson, and E. Tovar, "Implementing slot-based task-splitting multiprocessor scheduling," CISTER-ISEP, Polytechnic Institute of Porto, TR-100504, 2010.