IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Practical Aspects of Slot-Based Task-Splitting Dispatching in its Schedulability Analysis

Paulo Baltarejo Sousa

Konstantinos Bletsas

Bjorn Andersson

Eduardo Tovar

HURRAY-TR-110404

Version:

Date: 04-15-2011

Practical Aspects of Slot-Based Task-Splitting Dispatching in its Schedulability Analysis

Paulo Baltarejo Sousa, Konstantinos Bletsas, 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 semi-partitioning)scheduling algorithms assign most tasks to just oneprocessor but a few tasks are assigned to two or more processors,and they are dispatched in a way that ensures that a task neverexecutes on two or more processors simultaneously. A certaintype of task-splitting algorithms, called slot-based task-splitting,is of particular interest because of its ability to schedule tasksat high processor utilizations. We present a new schedulabilityanalysis for slot-based task-splitting scheduling algorithms thattakes the overhead into account and also a new task assignmentalgorithm.

© IPP Hurray! Research Group

Practical Aspects of Slot-Based Task-Splitting Dispatching in its Schedulability Analysis

Paulo Baltarejo Sousa*, Konstantinos Bletsas*, Björn Andersson†* and Eduardo Tovar*
*CISTER-ISEP Research Center, Polytechnic Institute of Porto, Portugal
†Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
Email: *{pbs, ksbs,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 certain type of task-splitting algorithms, called slot-based task-splitting, is of particular interest because of its ability to schedule tasks at high processor utilizations. We present a new schedulability analysis for slot-based task-splitting scheduling algorithms that takes the overhead into account and also a new task assignment algorithm.**

***Keywords*-Multiprocessor scheduling, task-splitting, schedulability analysis, real-time system overheads**

I. Introduction

Some years ago, technology constraints forced processor manufacturers to switch from uniprocessor to multiprocessor architectures. Nowadays, multiprocessors implemented on a single chip (called *multicores*) are the preferred platform for many real-time applications. However, real-time scheduling theory for uniprocessors is considered mature but real-time scheduling theory for multiprocessors is an emerging research field.

Traditionally, real-time scheduling algorithms for multiprocessors were categorized as either global or partitioned. *Global* scheduling algorithms store tasks in one global queue, shared by all processors. Tasks can migrate from one processor to another; that is, a task can be preempted during its execution and resume its execution on another processor. At any moment, the m highest-priority tasks are selected for execution on the m processors. Some algorithms of this kind achieve an utilization bound of 100% but generate too many preemptions. *Partitioned* scheduling algorithms partition the task set and assign all tasks in a partition to the same processor. Hence, tasks cannot migrate between processors. Such algorithms involve few preemptions but their utilization bound is at most 50%.

In recent years, the research community created another category of real-time scheduling algorithms called *task-splitting* or *semi-partitioning* [1], [2], [3], [4], [5], [6], [7], [8], [9]. The key idea of these algorithms is that they assign most tasks to just one processor but some 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.

Of particular interest is the class of task-splitting algorithms which subdivide time into *timeslots*. Each timeslot in turn consists of *processor reserves* (i.e. time windows) carefully positioned at a respective 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 placement of these reserves 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. At present, this scheduling theory depends on a set of assumptions that have no bearing on a real operating system. So, taking advantage of such a scheduling algorithm requires modeling these real-world effects into the schedulability analysis. Therefore, in this paper, we present new schedulability analysis for slot-based task splitting, accounting for overheads, and a new task assignment algorithm.

The rest of this paper is structured as follows. Section II presents a particular type of task-splitting scheduling algorithm called *slot-based*; to better illustrate this kind of scheduling algorithm, an example is provided (and used throughout the paper). In Section III the most important overheads of the slot-based task-splitting scheduling algorithm are described, and a new schedulability test is defined, taking into account those overheads. The new schedulability test is applied to the example and the results are presented in Section IV. Section V proposes a new task assignment algorithm which uses the new schedulability test. Finally, Section VI concludes the paper.

II. Slot-based task-splitting

Before describing in detail the slot-based scheduling algorithm [2] let us present the system model and assumptions as well as some important definitions. We consider real-time systems composed by m identical processors and n independent tasks (i.e. sharing no resources except for processors). Tasks of the task set τ are uniquely indexed in the range $1..n$ and processors in the range $1..m$. A task τ_i is characterized by worst-case execution time C_i, minimum inter-arrival time T_i and relative deadline D_i. We assume $0 \le C_i \le D_i$. If D_i is not stated, then $\forall i : D_i = T_i$. The utilization of task τ_i is defined as $u_i = \frac{C_i}{T_i}$ and the system utilization U_s is defined as $U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} u_i$.

Each task τ_i generates a potentially infinite sequence of *jobs*. The j^{th} job of τ_i (denoted $\tau_{i,j}$) becomes *ready* to execute at arrival time $a_{i,j}$ and completes execution at *finishing* time $f_{i,j}$. The *absolute deadline* of job $\tau_{i,j}$ is computed as $d_{i,j} = a_{i,j} + D_i$; a deadline is missed if $f_{i,j} > d_{i,j}$. The arrival times of any two consecutive jobs differ by at least T_i time units.

For convenience we define $\text{TMIN} = \min(T_1,\ T_2,\ ...,\ T_n)$. A designer-set parameter δ controls the frequency of migration of tasks assigned to two processors. Based on this parameter, the duration of the timeslot is computed as $S = \frac{\text{TMIN}}{\delta}$. Also, a parameter used to size the reserves is computed as $\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta + 1)} + \delta$ and parameter SEP (which guides task assignments and is equal to the utilization bound of the algorithm) is computed as $\text{SEP} = 4 \cdot (\sqrt{\delta \cdot (\delta + 1)} - \delta) - 1$.

To better illustrate 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. Let δ=4, which implies that SEP=0.8885. As with partitioned scheduling, this scheduling scheme can be divided into two algorithms: an offline algorithm for task assignment and an online dispatching algorithm.

A. Task Assignment Algorithm

Tasks whose utilization exceeds SEP (henceforth called *heavy tasks*) are each assigned to a dedicated processor. Then, the remaining tasks are assigned to the remaining processors in a manner similar to next-fit bin packing [10]. 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 by the next one (p + 1). Let us apply the task assignment

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 I
TASK SET (TIME UNIT ms)

algorithm to the system previously described. 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 (0.3052) of task τ_3 is assigned to processor P_2, just enough to make the utilization of processor P_2 equal to SEP ($0.5833 + 0.3052 = 0.8885$). This part is referred as $u_{hi}[P_2]$ and the remaining portion (0.2332) of task τ_3 is assigned to processor P_3, which is referred to as $u_{lo}[P_3]$. Fig. 1 shows the final task set assignment to the processors. We can observe: (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 P_2 and P_3 processors that have been assigned split tasks have time windows (called *reserves*) where these split tasks have priority over other tasks assigned to these processors. The length of the reserves are chosen such that no temporal overlap occurs (either (i) between reserves of the same split task on different processors or (ii) between reserves of different split tasks on the same processor), the split tasks can be scheduled, and also all non-split tasks can meet deadlines.

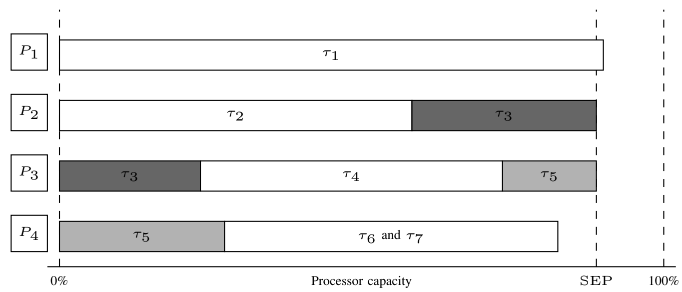


Fig. 1. Tasks assignment to processors.

Time is divided into timeslots of length S and *non-dedicated* processors (i.e. executing more than one task) usually execute split and non-split tasks. For such a processor p, the timeslot might be divided into three parts. The first x time units are reserved for executing the first split task on that processor; the last y time units are reserved for executing the second split task on that processor. The execution of the first (respectively, second) split task on processor p can be perceived as the execution of a task with utilization $u_{lo}[p]$ (respectively, $u_{hi}[p]$). The remaining part of the timeslot (henceforth denoted as N) is used to execute non-split tasks assigned to processor p and its length is computed as $N[p] = S - x[p] - y[p]$.

Reserves $x[p+1]$ and $y[p]$ for each split task τ_i must be sized such that $\frac{x[p+1]+y[p]}{S} = \frac{C_i}{T_i}$. 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 $\frac{x[p+1]+y[p]}{S}$, since a split task only executes during the reserves. Consequently, it is necessary to inflate reserves by α in order to always meet deadlines: $x[p] = S \cdot (\alpha + u_{lo}[p])$ and $y[p] = S \cdot (\alpha + u_{hi}[p])$. Table II shows the timeslot composition of each processor. The timeslot length is $S = \frac{\text{TMIN}}{\delta} = \frac{5.0000}{4} = 1.2500$.

CPU	x	N	y
P_1	0.0000	1.2500	0.0000
P_2	0.0000	0.8337	0.4163
P_3	0.3264	0.6947	0.2289
P_4	0.3764	0.8736	0.0000

TABLE II
TIMESLOT COMPOSITION OF EACH PROCESSOR (ms)

B. Dispatching Algorithm

On a dedicated processor, dispatching is trivial: whenever the (only) task is ready, it executes. On a non-dedicated processor, the dispatching algorithm works over the 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 ($x[p]$ or $y[p]$) and if the assigned split task is ready to be executed, then the split task is scheduled to run on the processor. Otherwise, the ready non-split task with the earliest deadline is scheduled to execute 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 run 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. 2 shows an execution timeline of the task set example. It assumes that all tasks arrive at time $t = 0$. Task execution is represented by a rectangle 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 inside its reserves. Outside its reserves, it does not use the processor, even if the processor is idle. In contrast, the non-split tasks execute mainly outside the reserves (x and y) but potentially also within the reserves, in particular when there is no split task ready to be executed. There are two clear situations in Fig. 2 that illustrate this. First (marked a), task τ_7 executes at the beginning of the timeslot, which begins at 6.25, because 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 6.25) and hence there is some available time on the reserve, which is used by non-split task τ_4.

III. Discrepancy Between Theory and Practice:Implementation on Linux kernel 2.6.34

Based on the design principles defined in [11], we have implemented the algorithm [2] in the Linux kernel 2.6.34 (the reader is referred to [12]).

In this section we introduce the real-world effects on the schedulability theory. Let us denote $\tau[p]$ as a set of non-split tasks ($\tau^{ns}[p]$) and split tasks ($\tau^{s}[p]$) assigned to processor p.

We will deal with heavy tasks later. The Demand Bound Function (dbf) [13] gives an upper bound on the required amount of execution time by a task set $\tau[p]$ on processor p over a time interval of length L. In the context of pure partitioned scheduling (and arbitrary deadlines), it is computed as follows:

$$dbf_{\tau[p]}(L,p) = \sum_{i \in \tau[p]} \max\left(0, \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \quad (1)$$

In the context of implicit-deadline task sets (meaning that $T_i = D_i$ for every task τ_i) the previous equation becomes:

$$dbf_{\tau[p]}(L,p) = \sum_{i \in \tau[p]} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot C_i \quad (2)$$

Similarly (in the context of partitioned scheduling), the Supply Bound Function (sbf) gives a lower bound on the amount of execution time supplied to the task set assigned to processor p over a time interval of length L without any constraint and is computed as follows:

$$sbf_{\tau[p]}(L,p) = L \quad (3)$$

Intuitively, a partitioned real-time system is schedulable if $dbf(L,p) \leq sbf(L,p)$, on every processor p and for every interval length $L > 0$.

Next, we will describe how the demand- and supply-bound are adapted for the task splitting scheme in consideration [2], before getting into incorporating the various overheads that occur in practice.

A. Schedulability test without overheads

Let us consider the schedulabity test of the slot-based scheduling algorithm without any constraints. The time required to execute $\tau^{ns}[p]$ is given by:

$$dbf_{\tau^{ns}[p]}(L,p) = \sum_{i \in \tau^{ns}[p]} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot C_i \quad (4)$$

The time supplied for the execution of non-split tasks on processor p over a time interval of length L is lower-bounded by:

$$sbf_{\tau^{ns}[p]}(L,p) = \left\lfloor \frac{L}{S} \right\rfloor \cdot N[p] + \max\left(0, \left(L - \left\lfloor \frac{L}{S} \right\rfloor \cdot S\right) - (x[p] + y[p])\right) \quad (5)$$

Analogously, the processor demand, over a time interval of length L, by a task τ_i split between processors p and $p + 1$, is computed as follows:

$$dbf_{\tau_i}(L,p) = \max\left(0, \left\lfloor \frac{L - (y[p] + x[p+1])}{S} \right\rfloor \cdot (y[p] + x[p+1])\right) \quad (6)$$

Intuitively, this corresponds to modelling the two reserves of the split task as a single task with $C = D = y[p] + x[p + 1]$ (i.e. arriving with zero laxity) and a period of S, which migrates between processors p and $p + 1$ during execution. For a task τ_i split between processors p and $p + 1$ a lower bound for the supply of processor time (from both processors, in an alternating manner) is given by:

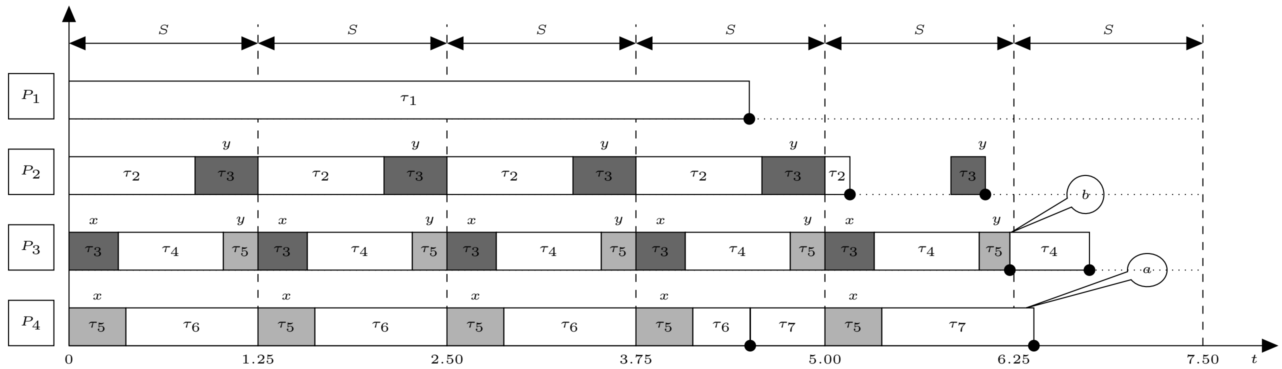


Fig. 2. Execution timeline

$$sbf_{\tau_i}(L) = \left\lfloor \frac{L}{S} \right\rfloor \cdot (y[p] + x[p+1]) + \max\left(0, \left(L - \left\lfloor \frac{L}{S} \right\rfloor \cdot S\right) - (S - (y[p] + x[p+1]))\right) \quad (7)$$

B. Release Jitter ($RelJ$)

Before describing the release jitter let us explain the release procedure in our implementation. This procedure is supported by a red-black binary tree and a high resolution timer per processor. All non-ready jobs are stored in the release binary tree sorted by the arrival time and the timer is set up to expire with the earliest arrival time. This way, the release of each job is done by the processor that the job is assigned to. Note that there is always a drift between the time instant when the timer should expire and the time instant at which the timer fires.

$relJ_{i,j}$ (*release Jitter* of job $\tau_{i,j}$), denotes the difference in time from when the job $\tau_{i,j}$ should arrive (the arrival time $a_{i,j}$) until it is enqueued into the ready queue, the release time ($r_{i,j}$). Then the timer callback enqueues job into the ready queue, and this is the release time ($r_{i,j}$) of job $\tau_{i,j}$. Fig. 3(a) ilustrates the $relJ_{i,j}$ and its relation with other parameters.

We define $RelJ$ as an upper bound on the value of any $relJ_{i,j}$ $\forall i, j$. In that case, $RelJ$ effectively "adds" to the execution of a task τ_i. Therefore we amend the derivation of the dbf to:

$$dbf_{\tau^{ns}[p]}(L, p) = \sum_{i \in \tau^{ns}[p]} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot (C_i + RelJ) \quad (8)$$

$$dbf_{\tau_i}(L, p) = \max\left(0, \left\lfloor \frac{L - (y[p] + x[p+1] + RelJ)}{S} \right\rfloor \cdot (y[p] + x[p+1] + RelJ)\right) \quad (9)$$

C. Reserve Jitter ($ResJ$)

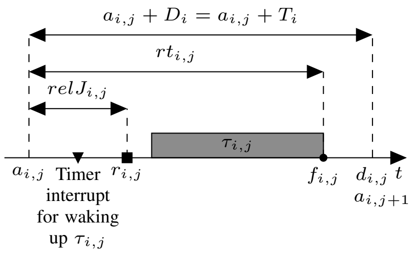
Another type of jitter, specific to slot-based task-splitting scheduling algorithms, concerns the (implementation-related) deviations from strict periodicity in the starting/ending of the reserves. Fig. 3(b) shows $resJ_{i,j}$, which represents the *reserve jitter* of job $\tau_{i,j}$ and denotes the discrepancy between the time when the job $\tau_{i,j}$ should (re)start executing (at the beginning of the reserve A, where A could be $x[p]$, $N[p]$ or $y[p]$) and when it actually (re)starts. It should be mentioned that the timers are set up to fire when the reserve should begin but 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 currently executing task to be preempted, which triggers the invocation of the dispatcher. Then the dispatcher selects a new task for execution, according to the dispatching algorithm. Note that, when the timer callback executes, as mentioned, it sets the current task to be preempted and also sets up the beginning of the next reserve. In order to avoid cumulative drift, this is done considering the theoretical (ideal) behaviour, therefore the expiration time for the timer is using as a point of reference the time that the timer callback should have *ideally* executed (i.e. not considering any delays). In practice, this means that the execution time of each reserve is reduced by $resJ_{i,j}$, consequently the amount of time that could be supplied for execution decreases.

Let us define $ResJ$ as an upper bound on the value of any $resJ_{i,j}$ $\forall i, j$. Then, for the set of non-split tasks on a processor p, the sbf is given by:

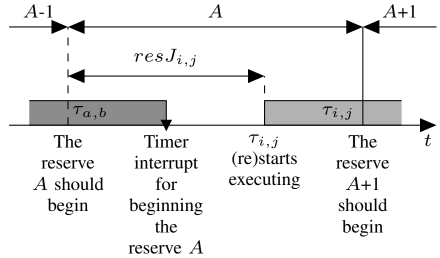
$$sbf_{\tau^{ns}[p]}(L, p) = \left\lfloor \frac{L}{S} \right\rfloor \cdot (N[p] - ResJ) + \max\left(0, \left(L - \left\lfloor \frac{L}{S} \right\rfloor \cdot S\right) - (x[p] + y[p] + ResJ)\right) \quad (10)$$

Similarly, for a task τ_i split between processors p and p+1:

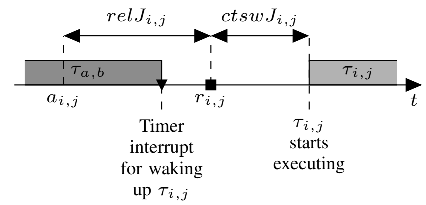
$$sbf_{\tau_i}(L, p) = \left\lfloor \frac{L}{S} \right\rfloor \cdot (y[p] + x[p+1] - ResJ) + \max\left(0, \left(L - \left\lfloor \frac{L}{S} \right\rfloor \cdot S\right) - (S - (y[p] + x[p+1] + ResJ))\right) \quad (11)$$



(a) Release jitter and relation with other job parameters.



(b) Reserve jitter.



(c) Context switch jitter.

Fig. 3. Jitter sources.

D. Context Switch ($CtswJ$)

Context switching is the procedure that swaps the currently executing job with another job, of higher priority. Since we have already accounted (via the term $ResJ$) for the context-switching overheads at the reserve boundary, here we only need (additionally) consider the overheads of the context switches generated by the EDF scheduling decisions. As a rough (pessimistic) estimate, the number of context switches over a time interval of length L is upper bounded by twice the number of job releases during that interval. This is because, under EDF, context switches occur either when some job is released or when it completes – but not every job release will cause a context switch. Fig. 3(c) illustrates the *context switch jitter* ($ctswJ_{i,j}$) of job $\tau_{i,j}$.

According to the dispatching algorithm, non-split tasks may be preempted, because they are scheduled according to EDF and usually more than one task shares the reserve $N[p]$. However, a split task executes at the highest priority within its reserves, therefore it cannot be preempted by other (i.e. non-split) tasks. As mentioned in Section II, non-split tasks could still execute within $x[p]$ and $y[p]$ reserves whenever the respective split tasks are not ready to execute. Any preemptions suffered by non-split tasks during their execution inside $x[p]$ and $y[p]$ reserves need not be accounted for because they occur within time intervals that have been excluded from the calculation of the respective sbf.

Let us define $CtswJ$ as an upper bound on the value of any $ctswJ_{i,j}$ $\forall i, j$. Then, for $\tau^{ns}[p]$, the set of non-split tasks on processor p, the dbf is computed as:

$$dbf_{\tau^{ns}[p]}(L,p) = \sum_{i \in \tau^{ns}[p]} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot (C_i + RelJ + 2 \cdot CtswJ) \quad (12)$$

Similarly, for a task τ_i split between processors p and p+1:

$$dbf_{\tau_i}(L,p) = \max\Bigg(0, \left\lfloor \frac{L - (y[p] + x[p+1] + RelJ + 2 \cdot CtswJ)}{S} \right\rfloor \cdot (y[p] + x[p+1] + RelJ + 2 \cdot CtswJ)\Bigg) \quad (13)$$

E. Interrupts

Let us assume that there is a limited number of interrupts (denoted by n^{int}) on the system and they are the events with the highest priority; that is, whenever an interrupt is fired, the processor stops what is doing (for instance, stops the execution of a job) to execute the Interrupt Service Routine (ISR) of that interrupt. This way, the interrupts increase the time required to execute a job. Hence, we will add the time consumed by interrupts to the dbf equation. Let us define T_i^{int} as the inter-arrival time of interrupt i and C_i^{int} as the worst-case execution time to execute the respective ISR. Let $\Lambda_{int[p]}$ be an upper bound on the amount of time spent executing the ISRs on processor p over a time interval of length L. Then:

$$\Lambda_{int[p]}(L,p) = \sum_{i \in n^{int}[p]} \left\lceil \frac{L}{T_i^{int}} \right\rceil \cdot C_i^{int} \quad (14)$$

Therefore, we amend the derivation of the dbf for the set of non-split tasks on processor p ($\tau^{ns}[p]$) to:

$$dbf_{\tau^{ns}[p]}(L,p) = \sum_{i \in \tau^{ns}[p]} \left\lfloor \frac{L}{S} \right\rfloor \cdot (C_i + RelJ + 2 \cdot CtswJ) + \Lambda_{int[p]} \quad (15)$$

Similarly, for a task τ_i split between processors p and p+1 :

$$dbf_{\tau_i}(L,p) = \Lambda_{int[p]} + \max\Bigg(0, \left\lfloor \frac{L - (y[p] + x[p+1] + RelJ + 2 \cdot CtswJ)}{T_i} \right\rfloor \cdot (y[p] + x[p+1] + RelJ + 2 \cdot CtswJ)\Bigg) \quad (16)$$

F. Heavy tasks

Heavy tasks (tasks whose utilisation u_i exceeds SEP) execute on a dedicated processor, facing no contention. Consequently, there is no need to divide time into timeslots. Hence, the sbf does not need to incorporate the $ResJ$. Therefore the sbf is given by Equation 3. Similarly, since a heavy task is the only task on its processor, it is never preempted by any other task. Hence, the dbf for a heavy task τ_i is computed as:

$$dbf_{\tau_i}(L,p) = \left\lfloor \frac{L}{S} \right\rfloor \cdot (C_i + RelJ + CtswJ) + \Lambda_{int[p]} \quad (17)$$

IV. Evaluation

We have conducted a set of experiments (with random task sets) that took approximately eight hours in a quad-core machine operating at 2.67 GHz. Adopting the real-time theory principles; that is, considering the worst case values (maximum execution time and minimum inter-arrival time values) we have collected the values of $RelJ$, $ResJ$ and $CtswJ$ metrics (see Table III). We also collected the values (worst case execution and the inter-arrival time) of the Tick interrupt and the interrupt 20 (irq20). The latter is related to the hard disk. As it is known, Tick is a periodic timer interrupt used by the system to do a set of operations. One of them is the possibility to invoke the scheduler. The periodicity of that timer is defined by a Linux Kernel macro `HZ`. In our system, we have set `HZ` equal to 1000, which means a periodicity of approximately 1 ms. We have disabled the tickless and also the CPU frequency scaling Linux kernel features, nevertheless, the minimal inter-arrival time collected was 0.1690 ms. However, the average is approximatelly 1 ms. To evaluate the impact of interrupts we have set up a controlled experiment by reducing the number of interrupts. We ran the experiments using the runlevel 1 with network connection and the filesystem journal mechanism disabled. Nevertheless, the worst case execution time of the irq20 was 0.0652 ms and the minimal inter-arrival time of that interrupt was 0.1271 ms, which leads to a required utilization of 0.5125 ($\frac{0.0652}{0.1271}$). Since, this kind of interrupts can be configured to be managed by one specific processor, we suggest assigning all interrupts to a dedicated processor which would not execute any application task. However, this is not possible for Tick. In fact, each processor has its own Tick, and therefore we must incorporate its overhead into the schedulability tests.

Metric	$RelJ$	$RelJ$	$CtswJ$	Tick	irq 20
C	0.0153	0.0110	0.0059	0.0117	0.0652
T	-	-	-	0.1690	0.1271

TABLE III
OVERHEAD TIME VALUES (ms)

Let us apply the schedulability test to the task set presented in Section II. Let us assume that the system is composed by one more processor (five processors), which is responsible for managing all interrupts. Fig. 4 plots the schedulability test of the $\tau^{ns}[P_2]$ in a time interval L (length 50 ms) and as it can be seen there are two points (time 42 and 48) where the dbf is higher than sbf, and therefore the system is not schedulable.

According to the original scheduling algorithm TMIN is computed as the minimal interarrival time of all tasks (TMIN $= \min(T_1, T_2, \cdots, T_n)$) and the timeslot length as $S = \frac{\text{TMIN}}{\delta}$. For executing heavy tasks, there is no need to divide the time into timeslots. Then, the T_i of the heavy

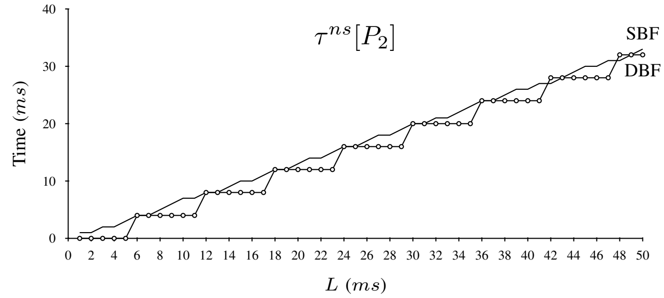


Fig. 4. Schedulability test fail.

tasks need not be considered, when computing TMIN. Consequently, S could be potentially larger, if the T_i of these tasks were the smallest. Note that, a larger timeslot reduces the impact of the overhead $ResJ$ on the scheduling algorithm. For instance, in the task set under study if we exclude T_1 thus,TMIN was equal to $\frac{T_2}{\delta} = \frac{6}{4} = 1.5000$ and as it can be seen from Fig. 5 the schedulabity test for $\tau^{ns}[P_2]$ in a time interval L (length 50 ms) succeeds.

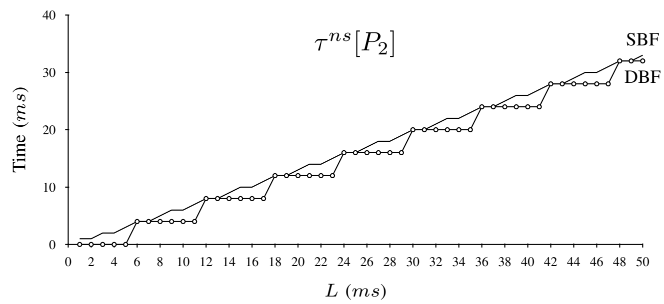


Fig. 5. Schedulability test succeed.

V. New Task Assignment Algorithm

Taking these overheads into account we defined a new task assigning algorithm (see Fig. 6). First, we classify tasks as heavy (if u_i exceeds SEP) or *light* (otherwise). Next, we order tasks such that τ_i with i in $1..L$ are all heavy and τ_i with i in $L{+}1..n$ are all light. L is the number of heavy tasks. A failure must be declared, if L exceeds the number of processors (m) or if $L{=}m$ and there is at least one light task to be assigned. After this, we assign the L heavy tasks to L processors. Note that, to each processor only one task is assigned. However, a failure is declared if the dbf of any task (subset) exceeds the sbf. We proceed computing the TMIN (using only light tasks) and the timeslot length (S). Finally, we assign the light tasks to the processors, in a manner similar to next-fit bin packing: If upon assigning a task to the current processor p, its utilization would not exceed SEP, then the task is assigned as non-split task. Otherwise, the task is split between processors p and p+1. Note that, the schedulability is guarranteed by invoking the schedulability test for each task assignment (for both split and non-split tasks); if the test fails, failure is declared.

1. **for** $p := 1$ to m **do**
2. $U[p] := 0$;
3. $u_{ns}[p] := 0$;
4. $u_{lo}[p] := 0$;
5. $u_{hi}[p] := 0$;
6. $\tau^{ns}[p] := 0$;
7. **end for**
8.
9. Let τ^{heavy} denote the set of tasks such that $C_i/T_i > \text{SEP}$
10. Let τ^{light} denote the set of tasks such that $C_i/T_i \leq \text{SEP}$
11. $L := |\tau^{heavy}|$;
12. Order tasks such that τ_i with i in $1..L$ are all in τ^{heavy}
13. and τ_i with i in $L+1..n$ are all in τ^{light}
14.
15. **if** $(L > m)$ OR $((L = m)$ AND $(n \geq m + 1))$
16. declare FAILURE;
17. **end if**
18.
19. **for** $i := 1$ to L **do**
20. $p := i$;
21. **if** $(dbf_{\tau[p]}(L, p) \leq sbf_{\tau[p]}(L, p), \forall L$) **then**
22. $U[p] = C_i/T_i$;
23. $\tau_i.processor\_id1 := p$;
24. $\tau_i.processor\_id2 := p$;
25. **else**
26. declare FAILURE;
27. **end if**;
28. **end for**
29.
30. $p := L + 1$;
31. TMIN:=$\min(T_{L+1}, T_{L+2}, ..., T_n)$;
32. S=TMIN/δ;
33. **for** $i := L + 1$ to n **do**
34. **if** $(u_{lo}[p] + u_{ns}[p] + C_i/T_i \leq \text{SEP})$ **then**
35. $\tau^{ns}[p] := \tau^{ns}[p] + \tau_i$;
36. **if** $(dbf_{\tau^{ns}[p]}(L, p) \leq sbf_{\tau^{ns}[p]}(L, p), \forall L$) **then**
37. $\tau_i.processor_i d1 := p$;
38. $\tau_i.processor_i d2 := p$;
39. $u_{ns}[p] := u_{ns}[p] + C_i/T_i$;
40. **else**
41. $\tau^{ns}[p] := \tau^{ns}[p] - \tau_i$;
42. **if** $(p = m)$
43. declare FAILURE;
44. **end if**;
45. $u_{hi}[p] = \max(u)$ such that:
46. predicate
47. $u_{hi}[p] \geq 0$ AND
48. $u_{hi}[p] \leq C_i/T_i$ AND
49. $u_{lo}[p + 1]$:=$C_i/T_i - u_{hi}[p]$ AND
50. $\tau_i.processor\_id1 := p$ AND
51. $\tau_i.processor\_id2 := p + 1$ AND
52. $(dbf_{\tau^{ns}[p]}(L, p) \leq sbf_{\tau^{ns}[p]}(L, p), \forall L)$ AND
53. $(dbf_{\tau_i}(L, p) \leq sbf_{\tau_i}(L, p), \forall L$)
54. end predicate
55. $u_{lo}[p + 1]$:=$C_i/T_i - u_{hi}[p]$;
56. $\tau_i.processor\_id1 := p$;
57. $\tau_i.processor\_id2 := p + 1$;
58. $p := p + 1$;
59. **end if**
60. **else**
61. same as lines 42 to 58
62. **end if**
63. **end for**
64. declare SUCCESS;

Fig. 6. The new algorithm for assigning tasks to processors.

VI. Conclusion

To our best knowledge this is the first approach to define a schedulability test taking into account real-world overheads for slot-based task-splitting. Using an implementation of slot-based scheduling algorithm [2] in Linux kernel 2.6.34 we have identified the most important overheads that such scheduling algorithm incurs. We modeled all these overheads and defined a new schedulability test taking into account these overheads as well as a new task assignment to processor algorithm.

In future work, we will consider dependent tasks to evaluate the impact of real-time synchronization protocols on slot-based task-splitting scheduling algorithms.

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. 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.

[11] P. B. Sousa, B. Andersson, and E. Tovar, "Challenges and design principles for implementing slot-based task-splitting multiprocessor scheduling," in *Work in Progress (WiP) session of the 31st IEEE Real-Time Systems Symposium (RTSS 10)*, San Diego, CA, USA, 2010. [Online]. Available: http://cse.unl.edu/rtss2008/archive/rtss2010/WIP2010/5.pdf

[12] ——, "Implementing slot-based task-splitting multiprocessor scheduling," in *6th IEEE International Symposium on Industrial Embedded Systems (SIES 11)*, Västerås, Sweden, 2011, pp. 256–265.

[13] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *11th Real-Time Systems Symposium*. IEEE Computer Society Press, 1990, pp. 182–190.