SPRINT: Extending RUN to Schedule Sporadic Tasks

Andrea Baldovin
Geoffrey Nelissen
Tullio Vardanega
Eduardo Tovar

CISTER-TR-140905
10-08-2014

SPRINT: Extending RUN to Schedule Sporadic Tasks

Andrea Baldovin, Geoffrey Nelissen, Tullio Vardanega, Eduardo Tovar

CISTER Research Center

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: baldovin@math.unipd.it, grrpn@isep.ipp.pt, , emt@isep.ipp.pt

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

Abstract

The RUN algorithm has proven to be a very effective techniquefor optimal multiprocessor scheduling, thanks to thelimited number of preemptions and migrations incurred bythe scheduled task set. This permits to achieve high systemutilisation rates typical of global scheduling approacheswithout paying too much penalty due to excessive preemptionand migration overheads. Unfortunately, the adoptionof RUN in real-world applications is limited by the missingsupport to sporadic task sets: we address this problem byproposing SPRINT (SPoradic Run for INdependent Tasks).SPRINT is proven correct for the vast majority of task setsand successfully scheduled all those randomly generated duringour experiments. Yet, its behaviour is not defined forsome specific task sets, which are however extremely rare [1].Interestingly, experimental results show that the favourableproperty of causing a small number of preemptions and migrationsachieved by RUN is preserved with SPRINT.

SPRINT: Extending RUN to Schedule Sporadic Tasks

Andrea Baldovin[†], Geoffrey Nelissen[§], Tullio Vardanega[†], Eduardo Tovar[§]

[†]Department of Mathematics
University of Padova
{baldovin, tullio.vardanega}@math.unipd.it

[§]CISTER/INESC-TEC, ISEP
Polytechnic Institute of Porto
{grrpn, emt}@isep.ipp.pt

ABSTRACT

The RUN algorithm has proven to be a very effective technique for optimal multiprocessor scheduling, thanks to the limited number of preemptions and migrations incurred by the scheduled task set. This permits to achieve high system utilisation rates typical of global scheduling approaches without paying too much penalty due to excessive preemption and migration overheads. Unfortunately, the adoption of RUN in real-world applications is limited by the missing support to sporadic task sets: we address this problem by proposing SPRINT (SPoradic Run for INdependent Tasks). SPRINT is proven correct for the vast majority of task sets and successfully scheduled all those randomly generated during our experiments. Yet, its behaviour is not defined for some specific task sets, which are however extremely rare [1]. Interestingly, experimental results show that the favourable property of causing a small number of preemptions and migrations achieved by RUN is preserved with SPRINT.

1. INTRODUCTION

The Reduction to UNiprocessor (RUN) algorithm [2, 1] is an optimal technique for multicore scheduling that has recently attracted the attention of researchers for its out-of-the-box approach to the problem. Together with U-EDF [3, 4] and in contrast with previous scheduling approaches for multiprocessor systems such as PD^2 [5], BF^2 [6, 4], DP-Wrap [7] or LRE-TL [8], RUN is one of the very few scheduling algorithms to achieve optimality without explicitly resorting to a notion of proportionate fairness as captured in the seminal work of Baruah et al. [9]. Indeed, RUN does not try to mimic the *fluid schedule* by explicitly assigning execution time proportional to their utilisation to each task. The simulation results of RUN [1], backed up by those of U-EDF [3], tend to show that relaxing the notion of fairness significantly reduces the number of preemptions and migrations suffered by the tasks during the schedule[1]. These two metrics are a trustworthy indicator of the interference caused by operating system activities to the executing applications, and eventually give a hint on how the schedulability of an application running on a real platform is impacted by the choice of a specific algorithm.

RUN is a two-phase algorithm. First, off line, the multiprocessor problem of scheduling a set of n tasks on m identical processors is reduced to a set of uniprocessor scheduling problems. This step relies on the concept of *dual schedule* together with bin packing techniques. It results in a *reduction tree* (see Figure 2 for an example) which will be used during the on-line phase to take appropriate scheduling decisions.

The second phase takes place at run time: tasks on the leaves of the reduction tree are globally scheduled among the m processors by traversing the tree downwards from the root to the leaves. The scheduling decisions for passing from a level l to the next level $l-1$ of the reduction tree are taken according to a uniprocessor scheduling algorithm, the optimality of which ensures the optimality of RUN. Therefore, the earliest-deadline first (EDF) algorithm [10] is usually chosen, although no restriction is imposed in principle on this choice. In fact, because of the wide range of choices in both the off-line and the on-line part, RUN could be imagined as a framework to define different multiprocessor scheduling algorithms, each distinguished by the way the reduction process is performed off-line and the scheduling choices taken at run time.

Although its authors classified RUN as semi-partitioned, the algorithm presents the traits of different categories: the use of aggregates of tasks (packing) to reduce the size of the problem certainly recalls hierarchical algorithms. The idea of packing itself of course borrows from purely partitioned approaches, even though partitioning is not performed per processor but rather among servers. Finally, as in global scheduling techniques, tasks are not pinned to processors and are free to migrate to any processor when scheduling decisions are taken.

More than just reducing the number of incurred preemptions and migrations, RUN presents some additional properties which turn out to be useful to support system design, and make its adoption appealing from an industrial perspective. Firstly, since the computation of the reduction tree is performed offline and the reduction process is guaranteed to converge, little re-engineering effort is required on the occurrence of system changes involving modifications to the task

[1]The authors of RUN proved that the average number of preemptions per job is upper bounded by $\lceil \frac{3p+1}{2} \rceil$ when p reduction operations are required (see Section 3), and they observed an average of less than 3 preemptions per job in their simulation results.

RTNS 2014 , October 8 - 10 2014, Versailles, France
Copyright 2014 ACM 978-1-4503-2727-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2659787.2659828.

set. This feature is highly desirable in settings where incrementality is a necessary requirement of the development and qualification processes, as it is often the case of industrial real-time systems. Secondly, the divide-et-impera approach taken by RUN may help enable *compositional* system design and verification, since servers mimic functionally cohesive and independent subsystems which may be allocated to a dedicated subset of system resources and analysed in isolation. Although this could be more easily achieved by strict partitioning, RUN provides it while achieving higher schedulable utilisation at run time, thus avoiding over provisioning of system resources.

Unfortunately, the main issue with RUN is that it is intended only for the scheduling of periodic task sets. This is a major limitation that hinders its applicability in a real-world scenario, since supporting asynchronous events like interrupts and sporadic task activations is a strong requirement for a scheduling algorithm to be industrially relevant.

In this paper we propose an extension to RUN to cope with this problem, i.e. supporting the scheduling of sporadic task sets. The paper is organized as follows. Section 2 presents the system model and notation we will use through the rest of the work. In Section 3 we recall the main ideas at the basis of RUN and its scheduling process, both in the off-line and on-line phases. The core of our contribution is presented in Section 4 in which we introduce SPRINT (SPoradic Run for INdependent Tasks), an extension of RUN which enables the scheduling of sporadic task sets that require less than two reduction levels in the reduction tree (which is in fact the case for the vast majority of task sets). While not formally proving the correctness of SPRINT due to space limitations (the proof can be consulted in [11]), each design choice is motivated in Section 4 and experimental evidence of the performance of SPRINT are provided in Section 5. Finally, in Section 6, we draw some final considerations on our contribution.

2. SYSTEM MODEL

We address the problem of scheduling a set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of n independent sporadic tasks with implicit deadlines on a platform composed of m identical processors. Each task $\tau_i \overset{\text{def}}{=} \langle C_i, D_i, T_i \rangle$ is characterized by a worst-case execution time C_i, a relative deadline D_i, and a *minimum* inter-arrival time $T_i = D_i$. A task τ_i releases a (potentially infinite) sequence of jobs. Each job $J_{i,q}$ of τ_i that arrives at time $a_{i,q}$ must execute for at most C_i time units before its deadline occurring at time $a_{i,q} + D_i$. The earliest possible arrival time of the next job $J_{i,q+1}$ of τ_i is at time $a_{i,q} + T_i$. Tasks are supposed independent, i.e. they do not have precedence, exclusion or concurrency constraints and they do not share any resource (software or hardware) at the exception of the processors.

The utilization of a task τ_i is defined as $U(\tau_i) \overset{\text{def}}{=} \frac{C_i}{T_i}$. Informally, it represents the percentage of processor time the task may use by releasing one job every T_i time units and executing each such job for C_i time units. The system utilization $U(\mathcal{T})$ is the sum of all task utilizations (i.e., $U(\mathcal{T}) \overset{\text{def}}{=} \sum_{\tau_i \in \mathcal{T}} U(\tau_i)$), which represents the minimum computational capacity that must be provided by the platform to meet all task deadlines. This means that a necessary condition to respect all job deadlines is to have a number of processors $m \geq U(\mathcal{T})$. Furthermore, since an *optimal scheduling algorithm* for independent sporadic tasks with implicit deadlines needs *exactly* $m = \lceil U(\mathcal{T}) \rceil$ processors to respect all job deadlines, we say that $\mathcal{T}$ is *feasible* on $m = \lceil U(\mathcal{T}) \rceil$. For this reason, in case $U(\mathcal{T}) < m$, we can safely insert idle (dummy) tasks to make up the difference, similarly to the original RUN algorithm.

We say that a job $J_{i,q}$ is *active* at time t if it has been released no later than t and has its deadline after t, i.e. $a_{i,q} \leq t < a_{i,q} + D_i$. If a task τ_i has an active job at time t then we say that τ_i is active and we define $a_i(t)$ and $d_i(t)$ as the arrival time and absolute deadline of the currently active job of τ_i at time t. Since we consider tasks with implicit deadlines (i.e., $D_i = T_i$), at most one job of each task can be active at any time t. Therefore, we use the terms "tasks" and "jobs" interchangeably in the remainder of this paper with no ambiguity. The set of active tasks in the system at time t is indicated by $\mathcal{A}(t)$.

3. REVIEW OF RUN

As a first step to build its reduction tree offline, RUN wraps each individual task τ_i into a higher-level structure S_i called *server* with the same utilisation, period and deadline. Then it resorts to the concepts of *dual schedule* and *supertasking* [12, 13, 14], whose reciprocal interactions are recalled in the next section.

3.1 Off-line phase

The simple observation behind RUN is that scheduling a task's execution time is equivalent to scheduling its idle time. This approach, named *dual scheduling*, had already been investigated in a few previous works [15, 16, 7, 17]. The dual schedule of a set of tasks $\mathcal{T}$ consists in the schedule produced for the dual set $\mathcal{T}^*$ defined as follows:

Definition 1 (Dual task). *Given a task τ_i with utilization $U(\tau_i)$, the dual task τ_i^* of τ_i is a task with the same period and deadline of τ_i and utilisation $U(\tau_i^*) \overset{\text{def}}{=} 1 - U(\tau_i)$.*

Definition 2 (Dual task set). *$\mathcal{T}^*$ is the dual task set of the task set $\mathcal{T}$ if (i) for each task $\tau_i \in \mathcal{T}$ its dual task τ_i^* is in $\mathcal{T}^*$, and (ii) for each $\tau_i^* \in \mathcal{T}^*$ there is $\tau_i \in \mathcal{T}$.*

Scheduling the tasks in $\mathcal{T}^*$ is equivalent to schedule the idle times of the tasks in $\mathcal{T}$, therefore a schedule for $\mathcal{T}$ can be derived from a schedule for the dual task set $\mathcal{T}^*$. Indeed, if τ_i^* is running at time t in the dual schedule produced for $\mathcal{T}^*$, then τ_i must stay idle in the actual schedule of $\mathcal{T}$ (also called *primal* schedule). Inversely, if τ_i^* is idle in the dual schedule, then τ_i must execute in the primal schedule.

Example 1. *Figure 1 shows an example of the correspondence between the dual and the primal schedule of a set $\mathcal{T}$ composed of three tasks executed on two processors. In this example tasks τ_1 to τ_3 have utilization of $\frac{2}{3}$ each, implying that their dual tasks τ_1^* to τ_3^* have individual utilization of $\frac{1}{3}$. The dual task set is therefore schedulable on one (logical) processor, which makes it a simpler problem, while the primal tasks τ_1 to τ_3 need two processors. Whenever a dual task τ_i^* is running in the dual schedule, the primal task τ_i remains idle in the primal schedule; conversely, when τ_i^* is idling in the dual schedule then τ_i is running in the primal schedule.*

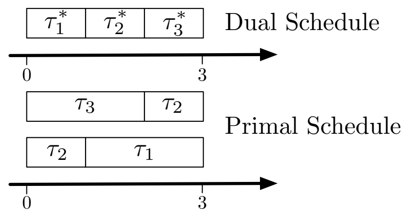


Figure 1: Correspondence between the primal and the dual schedule on the three first time units for three tasks τ_1 to τ_3 with utilizations of $\frac{2}{3}$ and deadlines equal to 3.

RUN takes benefit from the duality principle presented above by using it during the reduction process, i.e. applying the following DUAL operation at each level of the reduction tree:

Definition 3 (Dual operation/Dual server). *Given a server S, the* DUAL *operation defines a server S^* with utilisation $U(S^*) = 1 - U(S)$. S^* is called the* dual server *of S and shares the same deadline as S.*

Note however that the number of processors does not always diminish in the dual schedule. This actually depends on the utilization of the tasks in the particular task set. Lemma 1 (proven in [1]) explicits the relation existing between the utilisation of a task set and the utilisation of its dual:

Lemma 1. *Let $\mathcal{T}$ be a set of n periodic tasks. If the total utilization of $\mathcal{T}$ is $U(\mathcal{T})$, then the utilization of $\mathcal{T}^*$ is $U(\mathcal{T}^*) \overset{\text{def}}{=} n - U(\mathcal{T})$.*

Hence, if $U(\mathcal{T})$ is an integer, then $\mathcal{T}$ and $\mathcal{T}^*$ are feasible on $m = U(\mathcal{T})$ and $m^* = n - U(\mathcal{T})$ processors, respectively. Consequently, in systems where n is small enough to get

$$(n - U(\mathcal{T})) < m \tag{1}$$

the number of processors can be reduced — and so the complexity of the scheduling problem — by scheduling the dual task set $\mathcal{T}^*$ instead of $\mathcal{T}$. For instance, in Example 1, we have $U(\mathcal{T}) = 2$. $\mathcal{T}$ is therefore feasible on two processors. And because, $n = 3$, we get $U(\mathcal{T}^*) = n - U(\mathcal{T}) = 1$, thereby implying that the dual task set $\mathcal{T}^*$ is feasible on one processor only.

Therefore, in order to enforce Expression 1 being true and benefit from the duality, RUN uses a bin-packing heuristic to pack servers into higher level servers as defined by the following operation:

Definition 4 (Pack operation/Packed server). *Given a set of servers $\{S_1, S_2, \ldots, S_n\}$, the* PACK *operation defines a server S with utilisation $U(S) = \sum_{i=1}^{n} U(S_i)$. S is called the* packed server *of $\{S_1, S_2, \ldots, S_n\}$.*

In the remainder of this paper, we will use the notation $S \in B$ to specify that server S is packed into server B or more generally that server S is part of the subtree rooted in B.

The DUAL and PACK operations presented above are all the ingredients needed to build the RUN reduction tree and eventually transform the scheduling problem on m processors into an equivalent uniprocessor one. The process of constructing the tree consists in fact in alternatively packing existing servers to satisfy Equation 1, and then applying the duality to the newly obtained set of servers.

A reduction level in the reduction tree of RUN is therefore defined as follows:

Definition 5 (Reduction level). *The packing of the initial servers $\{S_1, S_2, \ldots, S_n\}$ is named $\mathcal{S}^0$. The successive application of the* DUAL *and the* PACK *operation to the set of servers $\mathcal{S}^l$ at level l defines a new set of servers $\mathcal{S}^{l+1}$ at level $l+1$ in the reduction tree. The intermediate level between l and $l+1$ (i.e. when the* DUAL *operation has been applied but the* PACK *operation has not) is indicated by l^* (see Figure 2 as an example).*

By recursively applying the DUAL and PACK operations, in [2] the authors proved that the number of processors needed to schedule the set of obtained servers eventually reaches one. Hence, the initial multiprocessor scheduling problem can be reduced to a uniprocessor scheduling one. More formally, this means that $\exists l, S_k^l : (|\mathcal{S}^l| = 1) \wedge (U(S_k^l) = 1)$, where $\mathcal{S}^l$ is the set of servers at level l and $S_k^l \in \mathcal{S}^l$. In fact, at every application of the reduction operation (i.e. at each level of the reduction tree) the number of servers is reduced by approximately one half, i.e. $|\mathcal{S}^{l+1}| \leq \left\lceil \frac{|\mathcal{S}^l|}{2} \right\rceil$.

3.2 On-line phase

The *on-line* phase of RUN consists in deriving the schedule for $\mathcal{T}$ from the schedule constructed with EDF at the uppermost reduction level (i.e. for the equivalent uniprocessor system). During runtime, each server S is characterized by a *current deadline* and a given *budget.*

Definition 6 (Server deadline in RUN). *At any time t, the deadline of server S_k^l on level l is given by*

$$d_k^l(t) \overset{\text{def}}{=} \min_{S_i^{l-1} \in S_k^l} \{d_i^{l-1}(t)\}$$

The deadline of the dual server $S_k^{l^*}$ on level l^* is given by

$$d_k^{l^*}(t) \overset{\text{def}}{=} d_k^l(t)$$

Deadline and budget are related since whenever a server S_k^l reaches its deadline, it replenishes its budget by an amount proportional to its utilisation $U(S_k)$, as follows:

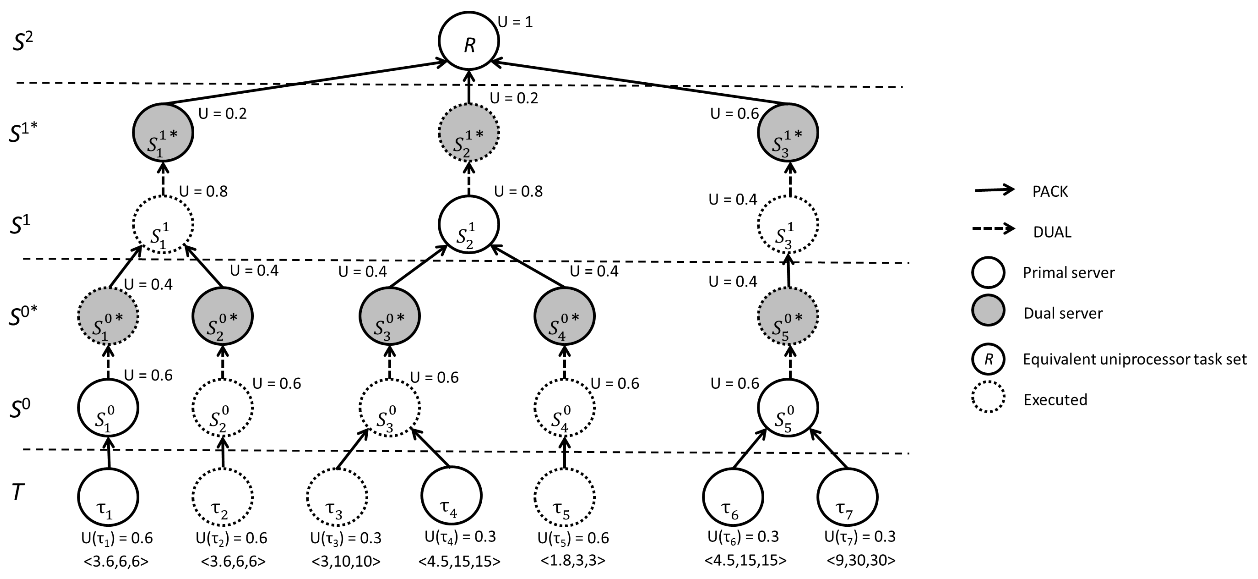
Definition 7 (Budget replenishment in RUN). *Let $R(S_k^l) \overset{\text{def}}{=} \{r_0(S_k^l), \ldots, r_n(S_k^l), \ldots\}$ be the time instants at which S_k^l replenishes its budget, with $r_0(S_k^l) = 0$ and $r_{n+1}(S_k^l) = d_k^l(r_n(S_k^l))$. At any instant $r_n(S_k^l) \in R(S_k^l)$ server S_k^l is assigned an execution budget $bdgt(S_k^l, r_n(S_k^l)) \overset{\text{def}}{=} U(S_k^l) \times \left(r_{n+1}(S_k^l) - r_n(S_k^l)\right)$.*

The budget of a server is decremented with its execution. That is, assuming that the budget of S_k^l was not replenished within the time interval $[t_1, t2]$, then

$$bdgt(S_k^l, t2) = bdgt(S_k^l, t1) - exec(S_k^l, t_1, t_2) \tag{2}$$

where $exec(S_k^l, t_1, t_2)$ is the time S_k^l executed during $[t_1, t2]$.

The schedule for $\mathcal{T}$ is finally built by applying the two following rules at each reduction level:



S^2
S^{1*}
S^1
S^{0*}
S^0
T
R
U = 1
U = 0.2
U = 0.2
U = 0.6
U = 0.8
U = 0.8
U = 0.4
U = 0.4
U = 0.4
U = 0.4
U = 0.4
U = 0.4
U = 0.6
U = 0.6
U = 0.6
U = 0.6
U = 0.6
$U(\tau_1) = 0.6$ <3.6,6,6>
$U(\tau_2) = 0.6$ <3.6,6,6>
$U(\tau_3) = 0.3$ <3,10,10>
$U(\tau_4) = 0.3$ <4.5,15,15>
$U(\tau_5) = 0.6$ <1.8,3,3>
$U(\tau_6) = 0.3$ <4.5,15,15>
$U(\tau_7) = 0.3$ <9,30,30>
PACK
DUAL
Primal server
Dual server
Equivalent uniprocessor task set
Executed

Figure 2: RUN reduction tree for a task set of 7 tasks.

Rule 1. *If a server S_k^l at reduction level l is running at time t, then the component server $S_j^{l-1^*} \in S_k^l$ with the earliest deadline is executed at level $(l-1)^*$ with budget $bdgt(S_j^{l-1^*}, t) > 0$. If a server S_k^l at reduction level l is not running at time t, then none of its component servers $S_j^{l-1^*} \in S_k^l$ at reduction level $(l-1)^*$ is executed.*

Rule 2. *If server $S_j^{l-1^*}$ is not running at level $(l-1)^*$, then server S_j^{l-1} is executed in the primal schedule at reduction level $l-1$. And inversely, if server $S_j^{l-1^*}$ is running at level $(l-1)^*$, then S_j^{l-1} is kept idle in the primal schedule at reduction level $l-1$.*

The server at the root of the reduction tree is assumed to always be running.

Example 2. *Let $\mathcal{T}$ be composed of 7 tasks τ_1 to τ_7 such that $U(\tau_1) = U(\tau_2) = U(\tau_5) = 0.6$ and $U(\tau_3) = U(\tau_4) = U(\tau_6) = U(\tau_7) = 0.3$. One possible reduction tree of $\mathcal{T}$ is provided in Figure 2. Let us assume that each of the seven tasks $\tau_i \in \mathcal{T}$ has an active job at time $t = 0$ characterized by its execution time, deadline and period (<c,d,t> respectively in Figure 2). According to Definition 6, each server $S_i^0 \in \mathcal{S}^0$ (with $1 \leq i \leq 5$) on the first reduction level inherits the deadline $d_i(t)$ of the corresponding task(s) τ_i. At the second reduction level $\mathcal{S}^1$, the sets of deadlines of S_1^1 and S_2^1 are $\{d_1(t), d_2(t)\}$ and $\{d_3(t), d_4(t), d_5(t)\}$ respectively, while S_3^1 inherits the deadlines of S_5^0, i.e. $\{d_6(t), d_7(t)\}$. Because a dual server has the same deadline of the corresponding primal server, if we execute EDF on the set of dual servers at level 1^* (Rule 1), $S_2^{1^*}$ is chosen to be executed at time t (see Figure 2). According to Rule 2, this means that both S_1^1 and S_3^1 should be running in the primal schedule at level 1. Applying EDF in each of these servers, we get that $S_1^{0^*}$ and $S_5^{0^*}$ must be running in the dual schedule of reduction level 0^*. Therefore, S_1^0 and S_5^0 must stay idle while S_2^0, S_3^0, S_4^0 must be executed in the primal schedule of reduction level 0. Consequently, it results that τ_2, τ_3 and τ_5 must execute at time t.*

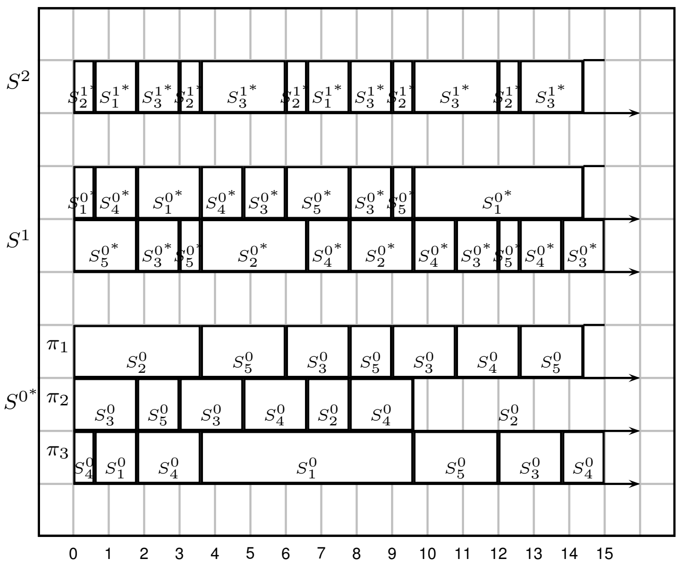


Figure 3: Possible RUN schedule for the task set in Example 3.

4. A GENERALIZATION OF RUN TO HANDLE SPORADIC TASKS

4.1 Motivating Example

Providing support to sporadic activations is arguably a desirable property for any industrially-relevant system, since this would make it possible to accommodate the asynchronous events triggered by the interaction with the external world. Unfortunately, RUN and its optimality results only apply to the scheduling of periodic task sets. In the next motivating example, a setting is shown where a feasible task set cannot be scheduled with RUN as a consequence of the the sporadic nature of some of its tasks.

Example 3. *Consider the task set composed by 7 tasks with implicit deadlines in the reduction tree shown in Figure 2. The beginning of a possible schedule constructed by RUN is given in Figure 3, where server S_2^0 corresponding to task τ_2 is allocated to processor π_1 at time $t = 0$; however, if τ_2 were a sporadic task with a job released at time 0.3, then it would be impossible for RUN to execute τ_2 for the* 3.6 *required time units by the instant 6.3, thus causing its deadline miss. In this case the problem might be solved with RUN by choosing the schedule where the allocations of S_1^0 and S_2^0 are switched, although there is no way to capture this requirement in the original algorithm.*

Example 3 above highlights the main problem with RUN: the algorithm assumes that, whenever a server is scheduled, its workload is available for execution (thus its budget is not null). This is a consequence of job releases occurring exclusively upon server deadlines when tasks are periodic. After jobs are released and new deadlines are propagated up to the root, a new schedule for $\mathcal{T}$ is computed by traversing the reduction tree top-down, i.e. by first finding a solution to the uniprocessor problem of scheduling jobs released by servers at level n. By applying the two rules shown in Section 3.2 the process proceeds downwards eventually producing a schedule for the actual tasks on the leaves. However, no clue is given to the scheduling process on how to select servers at intermediate levels and the corresponding branches in the tree to favour/ignore the scheduling of specific tasks, which may not have been activated yet. It is therefore interesting to investigate a possible generalization of the algorithm, which could account for variable arrival times and handle sporadic task sets.

It was proven in [2] that the structure of the reduction tree based on the notion of dual schedule is the key enabler for the low number of observed preemptions and migrations. We decided therefore to maintain this core idea while relaxing any assumption on job arrival times, giving rise to SPRINT, an algorithm for scheduling sporadic task sets, which is presented in the next section. Only the online phase of RUN is affected by SPRINT modifications. The offline part — i.e. the construction of the reduction tree — is identical in SPRINT and RUN.

4.2 SPRINT

The algorithm we present is not different from RUN in its basic mechanism: servers are characterized by *deadlines*, and at the firing of each such deadlines the *execution budget* of a server is replenished to provide additional computational resources for new job executions. However, the way those quantities (i.e., deadlines and budgets) are computed changes to accommodate possible sporadic releases. Additionally, we need to enforce a new *server priority* mechanism for the scheduling of component servers to avoid the waste of processor time when the transient load of the system is low. We now show how these key modifications differentiate SPRINT from RUN. Note that SPRINT can be currently applied to reduction trees with at most two reduction levels, therefore it cannot be considered optimal. However, the simulation results provided in Section 5, as well as previous studies performed in [1], show that task sets needing more than two reduction levels are extremely rare (less than one case over 600 with 100 processors and thousands of tasks). For this reason we argue that SPRINT is capable of scheduling the vast majority of task sets.

4.2.1 Deadlines of servers

Server deadlines play a central role in governing the scheduling process of RUN, as they (i) directly influence the budget replenishment of the servers and (ii) map to priorities under the EDF scheduling policy used by Rule 1. It is therefore critical to carefully assign deadlines so that proper scheduling decisions are taken online. This principle still holds in SPRINT, with the further complication that the deadlines of servers at level 0 should account for the possible sporadic nature of the individual tasks wrapped into them.

Let $r_n(S_k^0)$ be the release of a new job of server S_k^0 at level 0. In RUN the job released by S_k^0 would be assigned a deadline equal to the minimum deadline of the tasks packed in S_k^0 (see Definition 6). However, in SPRINT, because of the sporadic nature of tasks, some task τ_i may exist in S_k^0 which is not active at time $r_n(S_k^0)$ and therefore has no defined deadline. Nonetheless, we need to compute a deadline $d_k^0(r_n(S_k^0))$ for the job released by server S_k^0 at time $r_n(S_k^0)$. While computing this deadline, we want to preserve an important property of RUN: the firing of a deadline of any job released by any task τ_i always corresponds to the firing of a deadline in any server S_p^l such that $\tau_i \in S_p^l$. Furthermore, for the sake of simplicity, we do not want to update the deadline of S_k^0 at any instant other than the release of one of its job.

In order to meet those two requirements, for all the tasks $\tau_i \in S_k^0$ that are inactive at time $r_n(S_k^0)$, we consider their earliest possible deadline assuming that they release a job right after $r_n(S_k^0)$. That is, for each inactive task $\tau_i \in S_k^0$, we assume an artificial deadline $r_n(S_k^0) + D_i$.

Thanks to the discussion above, the deadline of a server in SPRINT can be defined as follows:

Definition 8 (Server deadline in SPRINT). *At any time t the deadline of a server S_k^0 on level $l = 0$ is given by*

$$d_k^0(t) \overset{\text{def}}{=} \min_{\tau_i \in S_k^0} \{d_i(r_k^0(t)) | \text{ if } \tau_i \in \mathcal{A}(r_k^0(t)); \quad r_k^0(t) + D_i | \text{ if } \tau_i \notin \mathcal{A}(r_k^0(t))\}$$

where $r_k^0(t)$ is the latest arrival time of a job of server S_k^0, i.e., $r_k^0(t) \overset{\text{def}}{=} \max_{r_n(S_k^0) \in \mathcal{R}(S_k^0)} \{r_n(S_k^0) \mid r_n(S_k^0) \le t\}$.

At any time t the deadline of any other server S_k^l on a level $l > 0$ is defined as in RUN. That is,

$$d_k^l(t) \overset{\text{def}}{=} \min_{S_i^{l-1} \in S_k^l} \{d_i^{l-1}(t)\}$$

It is worth noticing that, as expected, this definition preserves the property of RUN that the firing of a deadline of any job released by any task τ_i always corresponds to the firing of a deadline in any server S_k^l such that $\tau_i \in S_k^l$. Note however that the converse, although true in RUN, is not valid in SPRINT anymore, i.e. a deadline may be fired by a job of server S_k^l without any corresponding deadline in the task set $\mathcal{T}$.

4.2.2 Reduction level 0

As a more challenging modification to RUN, SPRINT must redefine the budget replenishment rules for its servers. This is needed because the execution budget assigned to a server S_k^l at any level of the reduction tree must reflect the execution time demand of the component tasks at the leaves of the subtree rooted in S_k^l. This time demand may now vary at any point in time as a consequence of sporadic releases and must be preserved upon the reduction process performed along the tree. While in RUN just one rule is needed to compute the budget of any server in the tree, in SPRINT we need to distinguish different budget replenishment rules corresponding to the different levels at which a server is located in the reduction tree.

Following the reasoning of Section 3.2, let $R(S_k^0) \overset{\text{def}}{=} \{r_0(S_k^0), \dots, r_n(S_k^0), \dots\}$ be the sequence of time instants at which the budget of S_k^0 is replenished. We still have $r_0(S_k^0) \overset{\text{def}}{=} 0$ and $r_{n+1}(S_k^0) \overset{\text{def}}{=} d_k^0(r_n(S_k^0))$. While, in RUN, the budget allocated to S_k^0 at time $r_n(S_k^0)$ is proportional to the utilisation of all the tasks in S_k^0, in SPRINT, the budget of S_k^0 should account only for the *active* tasks in S_k^0 at time $r_n(S_k^0)$. The budget of S_k^0 is therefore computed as follows:

$$bdgt(S_k^0, r_n(S_k^0)) \overset{\text{def}}{=} \sum_{\tau_j \in S_k^0 \cap \mathcal{A}(r_n(S_k^0))} U(\tau_j) \times \left(r_{n+1}(S_k^0) - r_n(S_k^0)\right) \qquad (3)$$

The provisioned budget for the execution of a server in SPRINT may therefore be smaller than in RUN. Yet, because of their sporadic nature, the tasks packed in S_k^0 may also release some jobs at any time instant t in-between two replenishment events $r_n(S_k^0)$ and $r_{n+1}(S_k^0)$. In this event, the budget of S_k^0 should be incremented to account for the new workload to be executed. More generally, if a set of tasks becomes active in a server S_k^0 at time t_a such that $r_n(S_k^0) < t_a < r_{n+1}(S_k^0)$, the budget of S_k^0 should be incremented of an amount proportional to the cumulative utilisation of all released jobs. Formally,

$$bdgt(S_k^0, t_a) \overset{\text{def}}{=} bdgt(S_k^0, t_a^-) + \sum_{\tau_i \in S_k^0 \cap Rel(t_a)} U(\tau_i) \times (d_k^0(t_a) - t_a) \qquad (4)$$

where $Rel(t_a)$ is the set of tasks releasing a job at time t_a and $bdgt(S_k^0, t_a^-)$ is the remaining execution budget of S_k^0 right before the arrival of those jobs.

The computation of the budget for the dual server $S_k^{0^*}$ is also impacted by the sporadic behavior of the tasks. Indeed, by definition of dual server, the primal server S_k^0 executes only when $S_k^{0^*}$ is idle, and conversely S_k^0 is kept idle when $S_k^{0^*}$ executes (see Rule 2). Therefore, in order to respect the deadline $d_k^0(t)$ of S_k^0, as a minimal requirement we need to ensure that $bdgt(S_k^{0^*}, t) \le (d_k^0(t) - t) - bdgt(S_k^0, t)$ at any time t. Since in RUN the budget assigned to a server S_k^0 may only vary at instants $r_n(S_k^l) \in R(S_k^l)$ (see Definition 7), it is sufficient to respect the equality $bdgt(S^{0^*}, t) \overset{\text{def}}{=} (d_k^0(t) - t) - bdgt(S_k^0, t)$ at any time t. In SPRINT instead the budget of S_k^0 may increase as soon as an inactive task $\tau_i \in S_k^0$ releases a new job (Equation 4). Therefore, whenever the budget $bdgt(S_k^0, t)$ increases at any time t due to the release of a new job of S_k^0 or a job of an inactive task $\tau_i \in S_k^0$, the budget of $S_k^{0^*}$ needs to be updated according to the following equation:

$$bdgt(S_k^{0^*}, t) = (d_k^0(t) - t) - bdgt(S_k^0, t) - \sum_{\tau_i \in S_k^0, \tau_i \notin \mathcal{A}(t)} U(\tau_i) \times (d_k^0(t) - t) \qquad (5)$$

where the last term accounts for the maximum workload by which the budget of S_k^0 could be inflated as a consequence of potential future job releases by the inactive tasks in S_k^0. In Equation 5 the maximum workload (currently active and potentially released) of the corresponding primal server is subtracted from the budget of the dual to prevent its execution for more than allowed.

To summarize, in SPRINT the computation of the budgets of any servers S_k^0 and $S_k^{0^*}$ at level $l = 0$ must comply with the two following rules:

Rule 3 (Budget replenishment at level 0). *At any instant $r_n(S_k^0) \in R(S_k^0)$ as defined in Section 3, servers S_k^0 and $S_k^{0^*}$ are assigned execution budgets*

$$\begin{cases} bdgt(S_k^0, r_n(S_k^0)) = \displaystyle\sum_{\tau_j \in S_k^0 \cap \mathcal{A}(r_n(S_k^0))} U(\tau_j) \times \left(r_{n+1}(S_k^0) - r_n(S_k^0)\right) \\ bdgt(S_k^{0^*}, r_n(S_k^0)) = (r_{n+1}(S_k^0) - r_n(S_k^0)) \\ \qquad - \displaystyle\sum_{\tau_i \in S_k^0} U(\tau_i) \times (r_{n+1}(S_k^0) - r_n(S_k^0)) \end{cases}$$

where $\mathcal{A}(r_n(S_k^0))$ is the set of active tasks at time $r_n(S_k^0)$.

Rule 4 (Budget update at level 0). *At any instant t (such that $r_n(S_k^0) < t < r_{n+1}(S_k^0)$) corresponding to the release of one or more jobs by one or more tasks in server S_k^0, the execution budgets of servers S_k^0 and $S_k^{0^*}$ are updated as follows:*

$$\begin{cases} bdgt(S_k^0, t) = bdgt(S_k^0, t^-) \\ \qquad + \displaystyle\sum_{\tau_j \in S_k^0 \cap Rel(t)} U(\tau_j) \times (d_k^0(t) - t) \\ bdgt(S_k^{0^*}, t) = (d_k^0(t) - t) - bdgt(S_k^0, t) \\ \qquad - \displaystyle\sum_{\tau_i \in S_k^0, \tau_i \notin \mathcal{A}(t)} U(\tau_i) \times (d_k^0(t) - t) \end{cases}$$

where $Rel(t)$ is the set of tasks releasing a job at time t and $bdgt(S_k^0, t^-)$ is the remaining execution budget of S_k^0 right before the arrival of those jobs at time t.

The following example shows how server budget can be computed in the presence of both active and inactive tasks, resuming from the motivating Example 3 in Section 4.1.

Example 4. *Let us consider server S_3^0 in the reduction tree depicted in Figure 2 and the possible schedule in Figure 3; additionally let τ_4 be sporadic and initially inactive at time $t = 0$. Since $d_3(0) \le t + D_4$, the deadline at time 0 of server S_3^0 in which τ_3 and τ_4 are packed is $d_3^0(0) = d_3(0) = 10$, corresponding to the deadline of task τ_3. Server S_3^0 is assigned budget proportional to the active tasks packed in it, i.e. $bdgt(S_3^0, 0) = \sum_{\tau_j \in S_3^0 \cap \mathcal{A}(0)} U(\tau_j) \times (d_3^0 - 0) = 0.3 \times 10 = 3$. Supposing now that τ_4 becomes active (and thus releases a job) at time $t_1 = 0.6$, the budget of server S_3^0 should be raised to satisfy the execution demand of τ_4. The amount of such increment is given by $\Delta bdgt(S_3^0, t_1) = \sum_{\tau_j \in S_3^0 \cap Rel(t_1)} U(\tau_j) \times (d_3^0(t_1) - t_1) = U(\tau_4) \times (d_3^0 - t_1) = 0.3 \times (10 - 0.6) = 2.82$. However, since $d_4(t_1) = 15.6 > d_3(t_1)$, τ_3 keeps on executing. At time $t_2 = 10$, τ_3 will release a new job with* *deadline $d_3 = 20$ (which is later than S_3^0 current deadline, $d_3^0(t_2) = 15.6$) and the budget of S_3^0 will be reset to $bdgt(S_3^0, 10) = (U(\tau_3)+U(\tau_4)) \times (d_3^0(t_2)-t_2) = (0.3+0.3) \times (15.6-10) = 3.36$ since both τ_3 and τ_4 will be active at t_2. Thus, the budget assigned overall to S_3^0 for the execution of τ_4 is given by the sum of the budgets assigned in the two intervals, i.e. $bdgt(\tau_4, [t_1, d_4(t_1)]) = bdgt(\tau_4, [t_1, t_2]) + bdgt(\tau_4, [t_2, d_4(t_1)]) = U(\tau_4) \times (t_2 - t_1) + U(\tau_4) \times (d_4(t_1) - t_2) = 0.3 \times 9.4 + 0.3 \times 5.6 = 4.5 = C_4$.*

We now prove that scheduling the packed servers at level 0 is equivalent to scheduling the task set $\mathcal{T}$.

Lemma 2. *Let S_k^0 be a server at level $l = 0$ of the reduction tree and assume that S_k^0 complies with Rules 3 and 4 for computing its budget. If S_k^0 always exhausts its budget by its deadlines then all jobs released by the tasks in S_k^0 respect their deadlines.*

Proof. We provide a proof sketch. According to Definition 8, the deadline of any job released by any task in S_k^0 corresponds to one of the deadline of S_k^0. Therefore, from Rules 3 and 4, the budget allocated to S_k^0 between any instant corresponding to the release $a_{i,q}$ of a job $J_{i,q}$ of a task $\tau_i \in S_k^0$ and the deadline $d_{s,p}$ of any job released by the same or another task $\tau_s \in S_k^0$, is proportional to the utilisation of the tasks in S_k^0 that are active between those two instants. That is, the budget allocated to the server (i.e., the supply) is larger than or equal to the sum of the worst-case execution times of the jobs of the tasks in S_k^0 with both an arrival and a deadline in the interval (i.e., the demand). And because EDF is an optimal scheduling algorithm and the cumulative load is always ≤ 1, all those jobs respect their deadlines. $\square$

Properly assigning deadlines and budgets to servers is not sufficient to guarantee that the algorithm works. As mentioned at the beginning of this section, due to the fact that all tasks are not always active at any given time t, the priority rules for servers must also be adapted in SPRINT to avoid wasting time while there is still pending work in the system. Indeed, as shown by Example 3, blindly using EDF to schedule servers in the presence of sporadic tasks may lead to deadline misses. Because a sufficient condition for guaranteing the schedulability of the tasks in $\mathcal{T}$ is that all jobs of the servers at level 0 respect their deadlines (as proven by Lemma 2), then it is straightforward to conclude that there is no need to execute any server S_k^0 at level 0 for more than its assigned budget. To enforce this, we rely on the idea of *dual schedule*, ensuring that S_k^0 does not execute when $S_k^{0^*}$ is running. Therefore, we just need to enforce the execution of $S_k^{0^*}$ as soon as a server S_k^0 exhausts its budget (even in case $S_k^{0^*}$ already exhausted its own budget, i.e. $bdgt(S_k^0, t) = bdgt(S_k^{0^*}, t) = 0$): this can be achieved by assigning the highest priority to $S_k^{0^*}$. As a consequence, by virtue of Rule 2 presented in Section 3, $S_k^{0^*}$ will be favourably chosen to execute at level $l = 0^*$ (the only exception being if another server S_p^0 also completed its execution), thereby implying that S_k^0 will not execute (thanks to Rule 1). These observations are formalised in Rule 5:

Rule 5 (Server priorities at level $l = 0^*$). *If the budget of a server S_k^0 is exhausted at time t, i.e. $bdgt(S_k^0, t) = 0$, then the dual server $S_k^{0^*}$ is given the highest priority. Otherwise, if $bdgt(S_k^0, t) > 0$, the priority of $S_k^{0^*}$ is determined by its deadline $d_k^0(t)$ as defined in Definition 8.*

4.2.3 Reduction level 1

We can extend the reasoning above to determine how the execution budgets should be replenished and how the scheduling decisions should be taken at levels $l = 1$ and 1^* of the reduction tree. We first start with the observations in the two following lemmas.

Lemma 3. *If $S_i^{1^*}$ executes at time t then all servers $S_k^0 \in S_i^1$ execute at time t.*

Proof. This lemma is a consequence of the dual operation applied in Rule 2. If $S_i^{1^*}$ executes at time t then, by Rule 2, S_i^1 does not execute at time t. Consequently, by Rule 1, none of the component servers $S_k^{0^*} \in S_i^1$ executes either. This implies (by Rule 2) that all tasks $S_k^0 \in S_i^1$ execute at time t. $\square$

Lemma 4. *If $S_i^{1^*}$ does not execute at time t then all severs $S_k^0 \in S_i^1$ but one execute at time t.*

Proof. If $S_i^{1^*}$ does not execute at time t then, by Rule 2, S_i^1 executes at time t. Consequently, by Rule 1, one component server $S_p^{0^*} \in S_i^1$ executes at time t. Therefore, applying Rule 2, all tasks $S_k^0 \in \{S_i^1\} \setminus S_p^0$ execute at time t. $\square$

A direct consequence of Lemmas 3 and 4 is that there is no need for executing $S_i^{1^*}$ when at least one of the servers $S_k^0 \in S_i^1$ exhausted its budget. Therefore, $S_i^{1^*}$ is assigned the lowest priority to prevent its execution, as long as at least one server $S_k^0 \in S_i^1$ has budget $bdgt(S_k^0, t) = 0$. Hence, the following rule applies at level $l = 1^*$:

Rule 6 (Server priorities at level $l = 1^*$). *If the budget of a server S_k^0 is exhausted at time t, i.e. $bdgt(S_k^0, t) = 0$, then the server $S_i^{1^*}$ such that $S_k^0 \in S_i^{1^*}$ is given the lowest priority. Otherwise, if $bdgt(S_k^0, t) > 0$ for all $S_k^0 \in S_i^{1^*}$, the priority of $S_i^{1^*}$ is determined by its deadline $d_k^1(t)$ as specified in Definition 8.*

The following example shows how assigning priorities according to Rules 5 and 6 at levels 0^* and 1^* affects the scheduling of the servers at level 0.

Example 5. *Consider again the task set in Figure 2 and focus on the subtree rooted in $S_2^{1^*}$. We assume τ_3, τ_4 and τ_5 to be sporadic and releasing their first jobs at time instants 3, 0.1 and 0 respectively. The beginning of a possible schedule constructed by RUN is given in Figure 4. Since the budget of S_3^0 is null at time 0, according to Rule 5 server $S_3^{0^*}$ is assigned the highest priority among $\mathcal{S}^{0^*}$, whereas $S_2^{1^*}$ takes the lowest priority in $\mathcal{S}^{1^*}$ according to Rule 6. Consequently, SPRINT selects $S_1^{1^*}$ at level $\mathcal{S}^{1^*}$ and $S_3^{0^*}$ at $\mathcal{S}^{0^*}$, which prevents dispatching τ_3 and τ_4, that have not been released yet, in favour of S_4^0 and τ_5. Later at time 0.1, when τ_4 is released, server S_3^0 is given budget proportional to the execution demand of τ_4, $S_3^{0^*}$ takes nominal priority (that of S_3^0, according to Definition 8), and $S_2^{1^*}$ is assigned priority equal to the deadline of S_4^0 by virtue of Rule 6. This new assignment permits to select $S_2^{1^*}$ at level $\mathcal{S}^{1^*}$ and consequently enables the scheduling of S_3^0, thus τ_4, at level S^0. At time 3, τ_3 is also released and preempts τ_4 as a consequence of its earlier deadline. At the same time τ_5 completes and, assuming that its next arrival is later than time 3, servers $S_4^{0^*}$ is assigned the highest priority among $\mathcal{S}^{0^*}$, whereas $S_2^{1^*}$*

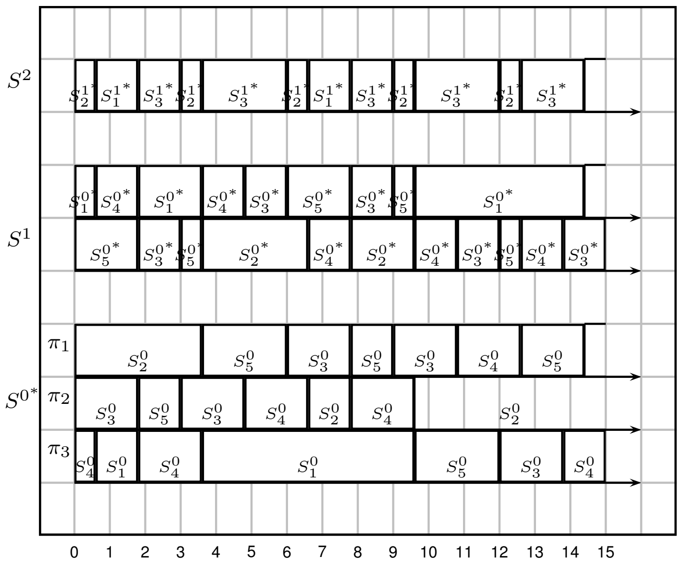


Figure 4: Possible RUN schedule for the task set in Example 5.

takes again the lowest priority in S^{1^*}. Similarly to the situation at time 0, preventing the selection of server $S_2^{1^*}$ implies scheduling the server with the highest priority in S^{0^*}, this time $S_4^{0^*}$, which in turn favours the scheduling of the active workload in S_3^0 over S_4^0 (and thus τ_5) in the primal schedule.

At levels 1 and 1^*, the budget replenishment policy applied at any instant $r_n(S_k^1) \in R(S_k^1)$ is no different from RUN. Hence, the algorithm still respects the following rule:

Rule 7 (Budget replenishment at level 1). *At any instant $r_n(S_k^1) \in R(S_k^1)$ servers S_k^1 and $S_k^{1^*}$ are assigned execution budgets*

$$\begin{cases} bdgt(S_k^{1^*}, r_n(S_k^1)) = U(S_k^{1^*}) \times \left(r_{n+1}(S_k^1) - r_n(S_k^1)\right) \\ bdgt(S_k^1, r_n(S_k^1)) = (r_{n+1}(S_k^1) - r_n(S_k^1)) - bdgt(S_k^{1^*}, r_n(S_k^1)) \end{cases}$$

One more rule is however needed to define the behaviour of the algorithm when a task releases a job at time t such that $r_n(S_k^1) < t < r_{n+1}(S_k^1)$. This rule is given below and uses the operator $[x]_y^z$ defined as $\min\{z, \max\{y, x\}\}$.

Rule 8 (Budget update at level 1). *At any instant t such that $r_n(S_k^1) < t < r_{n+1}(S_k^1)$, corresponding to the update of one or more jobs from one or more server $S_p^0 \in S_k^1$:*

- *if $bdgt(S_p^0, t^-) = 0$ and calling $t_0 \geq r_n(S_k^1)$ the instant at which $bdgt(S_p^0, t)$ became equal to 0, then the execution budgets of servers S_k^1 and $S_k^{1^*}$ are updated as follows:*

$$\begin{cases} bdgt(S_k^{1^*}, t) = \left[U(S_k^{1^*}) \times (d_k^1(t) - t)\right]_{bdgt(S_k^{1^*}, t_0) - (t - t_0)}^{bdgt(S_k^{1^*}, t_0)} \\ bdgt(S_k^1, t) = (d_k^1(t) - t) - bdgt(S_k^{1^*}, t) \end{cases}$$

where $bdgt(S_k^0, t^-)$ and $bdgt(S_k^{1^*}, t^-)$ are the remaining execution budgets of S_k^0 and $S_k^{1^*}$, respectively, right before the budget update occurring at time t;

- *otherwise, if $bdgt(S_p^0, t^-) > 0$ for all updated servers $S_p^0 \in S_k^1$, then $bdgt(S_k^{1^*}, t)$ and $bdgt(S_k^1, t)$ remain unchanged, i.e. $bdgt(S_k^{1^*}, t) = bdgt(S_k^{1^*}, t^-)$ and $bdgt(S_k^1, t) = bdgt(S_k^1, t^-)$.*

Due to space limitations, we cannot provide a formal proof in this paper that all servers $S_k^{1^*}$ of level $l = 1$ always respect their deadlines. However, as proven in [11], the following lemma holds:

Lemma 5. *If all servers of level 1^* comply with Rules 7 and 8 for computing their budgets, then all servers of level 1^* respect all their deadlines.*

4.2.4 Reduction level 2

By assumption, at most two reduction levels are present in SPRINT: consequently, level 2 can only be the root of the reduction tree. Since by definition the root has always utilisation equal to 100%, it is always executing and neither budget nor priority need to be computed for it.

Theorem 1. *SPRINT respects all the deadlines of all the jobs released by sporadic tasks with implicit deadlines when there are a maximum of two reduction levels in the reduction tree.*

PROOF. See [11]. □

5. EXPERIMENTAL RESULTS

We now proceed to compare SPRINT to state-of-the-art multicore scheduling algorithms. In particular, we are interested in counting the number of preemptions and migrations incurred by the task sets to be scheduled during those tests.

Please note that from a run-time complexity viewpoint, RUN needs to traverse the reduction tree upon each scheduling event. SPRINT has the same behaviour while performing a few supplementary summations and multiplications to properly adjust server utilisation, with minimal additional impact on system overhead. Since it has been demonstrated in [18] that RUN can be actually implemented with reasonable performance when compared to other existing partitioned and global algorithms[2], we assume that this result can be extended to SPRINT, which is based on RUN, to only focus on evaluation by simulation.

All our experiments relate SPRINT with Partitioned-EDF (P-EDF), Global-EDF (G-EDF) and U-EDF, by scheduling randomly generated sporadic task sets. Individual tasks are characterised by their minimum inter-arrival time, randomly chosen in the range of $[5, 100]$ time units. Sporadic releases are simulated by randomly picking an arrival delay for each job in a range of values depending on the specific scenario. Every point in the graphs presented in this section is the result of the scheduling of 1000 task sets. During the off line reduction process of SPRINT, no task set required more than 2 levels in its reduction tree and all tasks always respected their deadlines when using SPRINT.

In the first batch of experiments we studied SPRINT performance as a function of the varying system utilisation. We simulated a system with 8 processors and we randomly generated task utilizations between 0.01 and 0.99 until the

[2]The cited implementation has been made on the LITMUS^{RT} extension to the Linux kernel developed by UNC [19, 20].

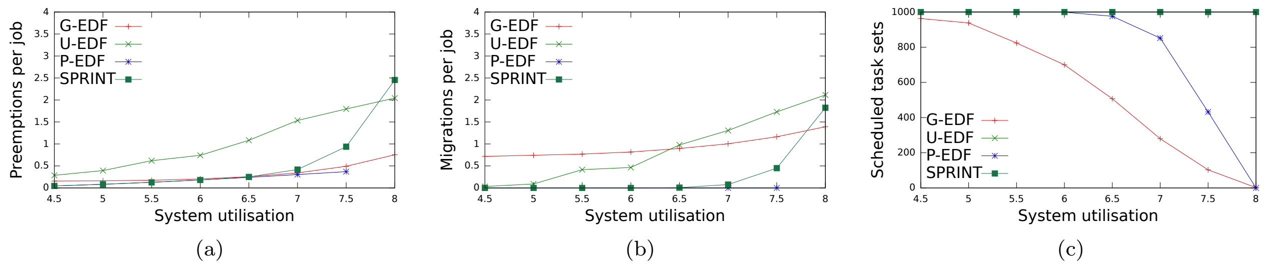


Figure 5: Comparative results for SPRINT with respect to G-EDF, P-EDF and U-EDF in terms of preemptions (a) and migrations (b) per job, and number of schedulable task sets (c) with increasing system utilisation.

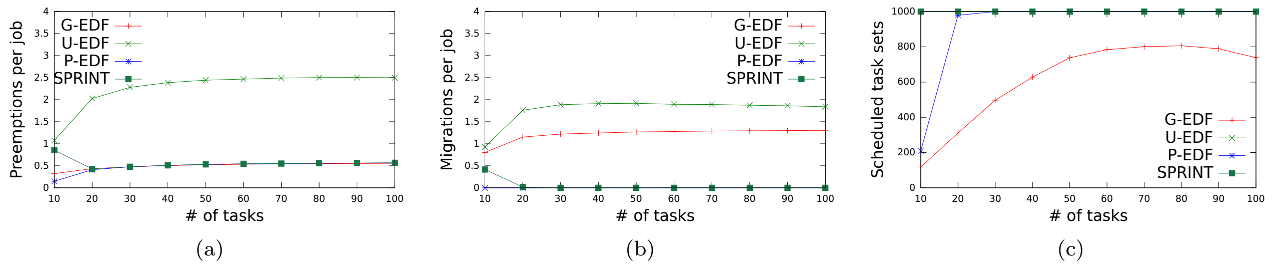


Figure 6: Comparative results for SPRINT with respect to G-EDF, P-EDF and U-EDF in terms of preemptions (a) and migrations (b) per job, and number of schedulable task sets (c) with increasing number of tasks.

targeted system utilization was reached, increasing it progressively from 55% to 100%. Sporadic delays suffered by the jobs were randomly chosen in the range of $[0, 100]$ time units. Figures 5(a) and 5(b) show the results obtained for SPRINT in terms of preemptions and migrations per job, respectively; in particular, we notice that the number of migrations incurred by SPRINT is always smaller than the number experienced under both G-EDF and U-EDF up to 95% utilisation ($U(\mathcal{T}) = 7.5$), whereas it approaches U-EDF performance afterwards. The number of preemptions is similar to the well-known results for P-EDF, at least up to 85% utilisation, i.e. $U(\mathcal{T}) = 7$. After that point however the number of scheduled task sets for P-EDF and G-EDF drops substantially, as evident from Figure 5(c)[3], until the extreme of 100% utilisation where not even a valid partitioning is found for P-EDF. As expected, the schedulability ratios for U-EDF and SPRINT remain 100%, while the number of incurred preemptions becomes similar due to a saturated system.

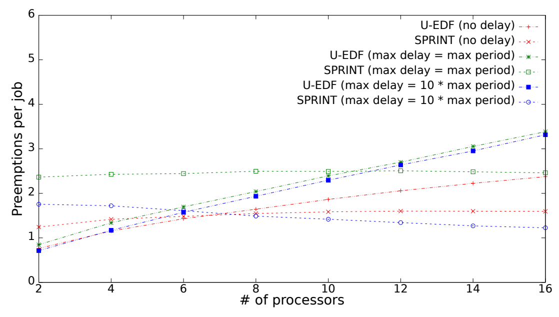
In the second experiment, we kept the system utilisation fixed to 90% (i.e. $U(\mathcal{T}) = 7.2$), in order to still observe some feasible task sets under G-EDF and P-EDF, and varied the number of tasks. Task utilizations were generated using the method proposed in [21] and sporadic delays were picked in the range $[0, 100]$ time units. In our expectations this experiment would challenge even more the relative performance of the algorithms, since growing the number of concurrent tasks in the system potentially increases the work to be performed by the scheduling algorithm. Figures 6(a) and 6(b) show that the number of preemptions for SPRINT is similar to that of P-EDF and G-EDF, while the number of migrations is even smaller (in fact null) than the migrations registered by G-EDF and U-EDF. However, with a small number of tasks, whose individual utilisation must be therefore large, P-EDF and G-EDF fail to schedule some task sets as a consequence of the impossibility of finding a good partitioning and of taking advantage of task migration, respectively (Figure 6(c)). U-EDF is instead comparable to SPRINT in terms of achieved schedulability, still paying some penalty due to a higher number of preemptions and migrations.

As a final experiment we observed the behaviour of SPRINT and U-EDF when the number of processors in the system increases, while keeping the system fully utilised. As expected, both the number of preemptions (Figure 7(a)) and migrations (Figure 7(b)) increase for U-EDF with the size of the system, whereas for SPRINT it remains constant on average, and always below the value of 3. This is in line with the results obtained for RUN and by virtue of the observation that no task set in our experiments requires more than 2 reduction levels. Note however, that some experiments show results were up to 5 preemptions per job are needed in average. This breaks the upper-bound on the average number of preemptions per job proven for RUN and is the logical result of the sporadic releases of the jobs.

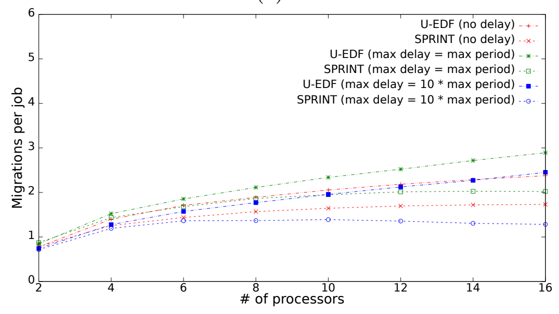
The same graphs also show how the behaviour of both SPRINT and RUN are affected by modifying the maximum delay that sporadic jobs could incur. To this end we defined three representative scenarios: (i) in the first one, jobs do not incur any delay (i.e., max delay $= 0$), which corresponds to having only periodic tasks and is therefore suitable to roughly compare SPRINT and U-EDF on a strictly periodic system; (ii) in the second, job delays are randomly picked in the range $[0, \mathit{max\_period}]$, so that there is at least one job released by each task every 200 time units; finally, (iii) in the third scenario, job delays are chosen in the range $[0, 10 \times \mathit{max\_period}]$. We notice on Figure 7 that scenario

[3]In that case we only count the number of preemptions and migrations incurred by the schedulable task sets.

(ii) is the most expensive both in terms of preemptions and migrations, for both algorithms. This is explained by the fact that in that scenario, jobs are released often enough to always keep the processors busy; additionally such releases are likely to happen out-of-phase with respect to each other, thereby generating more scheduling events. On the contrary, in setting (i), jobs are more likely to be released in phase, whereas in setting (iii), job releases are far less frequent, thus diluting the number of dispatched scheduling events.



(a)



(b)

Figure 7: Comparative results for SPRINT with respect to U-EDF in terms of preemptions (a) and migrations (b) per job, with different inter-arrival times and increasing number of processors.

6. CONCLUSIONS

In this paper we presented SPRINT, an extension to RUN to schedule sporadic task sets in multiprocessor systems. Although only applicable to those task sets whose reduction tree does not require more than two reduction levels, our algorithm presents a first yet solid investigation on how optimal multiprocessor scheduling of sporadic taks can be achieved by embracing a RUN-like philosophy. The benefits thereof can be leveraged by simply re-defining the priority and budget replenishment rules for servers, which need to be taken into account upon the occurrence of RUN's scheduling events. Experimental evidence confirmed that the low number of preemptions and migrations, and the schedulability results enjoyed with RUN are in fact preserved by SPRINT. We plan therefore to carry on with the study of SPRINT to make it suitable for the scheduling of any given task set, with no restriction imposed on the height of its reduction tree.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER), and by National Funds through FCT and the EU ARTEMIS JU funding, within project ref. ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO).

7. REFERENCES

[1] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach," *Real-Time Systems*, vol. 49, no. 4, pp. 436–474, 2013.

[2] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *Proceedings of the 32th IEEE Real-Time Systems Symposium (RTSS)*, pp. 104–115, 2011.

[3] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 13–23, 2012.

[4] G. Nelissen, *Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems*. PhD thesis, Université Libre de Bruxelles, 2013.

[5] A. Srinivasan and J. H. Anderson, "Optimal rate-based scheduling on multiprocessors," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 189–198, 2002.

[6] G. Nelissen, H. Su, Y. Guo, D. Zhu, V. Nélis, and J. Goossens, "An optimal boundary fair scheduling," *Real-Time Systems*, vol. 50, no. 4, pp. 456–508, 2014.

[7] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, pp. 389–429, 2011.

[8] S. Funk and V. Nadadur, "LRE-TL: An optimal multiprocessor algorithm for sporadic task sets," in *Proceedings of the 17th International Conference on Real-Time and Network Systems (RTNS)*, pp. 159–168, 2009.

[9] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[10] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[11] A. Baldovin, G. Nelissen, T. Vardanega, and E. Tovar, "SPRINT: Extending RUN to schedule sporadic tasks (appendix)," tech. rep., 2014. Available online at `http://tinyurl.com/l8wn8f7`.

[12] M. Moir and S. Ramamurthy, "Pfair scheduling of fixed and migrating periodic tasks on multiple resources," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pp. 294–303, 1999.

[13] P. Holman and J. H. Anderson, "Using supertasks to improve processor utilization in multiprocessor real-time systems," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 41–50, 2003.

[14] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 243–252, 2008.

[15] S. K. Baruah, J. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proccedings of the 9th International Parallel Processing Symposium (IPPS)*, pp. 280–288, 1995.

[16] G. Levin, C. Sadowski, I. Pye, and S. Brandt, "SNS: A simple model for understanding optimal hard real-time multiprocessor scheduling," Tech. Rep. ucsc-soe-11-09, UCSC, 2009.

[17] D. Zhu, X. Qi, D. Mossé, and R. Melhem, "An optimal boundary fair scheduling algorithm for multiprocessor real-time systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1411–1425, 2011.

[18] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting RUN into practice: implementation and evaluation," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[19] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pp. 111–126, 2006.

[20] `http://www.litmus-rt.org/`, 2014. LITMUS^{RT}, The Linux Testbed for Multiprocessor Scheduling in Real Time Systems.

[21] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 6–11, 2010.