Technical Report

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn Andersson

Konstantinos Bletsas

HURRAY-TR-080402

Version: 0

Date: 04-17-2008

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn Andersson, Konstantinos Bletsas

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: bandersson@dei.isep.ipp.pt, ksbs@isep.ipp.pt

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

Abstract

Consider the problem of scheduling n sporadic tasks so as to meet deadlines on m identical processors. A task is characterised by its minimum interarrival time and its worst-case execution time. Tasks are preemptible and may migrate between processors. We propose an algorithm with limited migration, configurable for a utilisation bound of 88% with few preemptions (and arbitrarily close to 100% with more preemptions).

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn Andersson and Konstantinos Bletsas
IPP-HURRAY! Research Group, Polytechnic Institute of Porto (ISEP-IPP),
Rua Dr. António Bernardino de Almeida 431, 4200-072 Porto, Portugal
bandersson@dei.isep.ipp.pt, ksbs@isep.ipp.pt

Abstract

Consider the problem of scheduling n sporadic tasks so as to meet deadlines on m identical processors. A task is characterised by its minimum interarrival time and its worst-case execution time. Tasks are preemptible and may migrate between processors. We propose an algorithm with limited migration, configurable for a utilisation bound of 88% with few preemptions (and arbitrarily close to 100% with more preemptions).

1. Introduction

Consider the problem of preemptively scheduling n sporadically arriving tasks on m identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor likewise in the range $1..m$. A task τ_i generates a (potentially infinite) sequence of jobs. The arrival times of these jobs cannot be controlled by the scheduling algorithm and are *a priori* unknown. We assume that the time between two successive jobs by the same task τ_i is at least T_i. Every job by τ_i requires at most C_i time units of execution over the next T_i time units after its arrival. We assume that T_i and C_i are real numbers and $0 \leq C_i \leq T_i$. A processor executes at most one job at a time and no job may execute on multiple processors simultaneously. The utilisation is defined as $U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} \frac{C_i}{T_i}$. The utilisation bound $\mathrm{UB_A}$ of an algorithm A is the maximum number such that all tasks meet their deadlines when scheduled by A, if $U_s \leq \mathrm{UB_A}$.

Multiprocessor scheduling algorithms are often categorised as *partitioned* or *global*. Global scheduling stores tasks which have arrived but not finished execution in one queue, shared by all processors. At any moment, the m highest-priority tasks among those are selected for execution on the m processors. In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same processor. Tasks may not migrate from one processor to another. The multiprocessor scheduling problem is thus transformed to many uniprocessor scheduling problems. This simplifies scheduling and schedulability analysis as the wealth of results in uniprocessor scheduling can be reused. Unfortunately, all partitioned scheduling algorithms have utilisation bounds of 50% or less. Global scheduling may achieve a utilisation bound of 100% using the *pfair* [5, 1] family of algorithms. Yet, this great utilisation bound comes at a price; all task parameters must be multiples of a time quantum and in every quantum a new task is selected for execution. Preemption counts can thus be high [9]. Like Baker (see [4, p. 12]), we desire a high utilisation bound without too many preemptions.

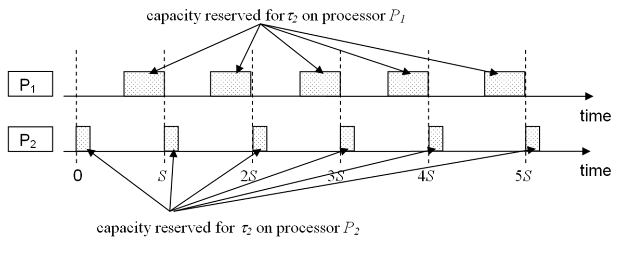
This paper proposes an algorithm for scheduling n sporadic tasks on m processors. It assigns at most $m-1$ tasks to two processors (carefully dispatched so that they never execute on both simultaneously) and the rest to only one processor. This design circumvents the limitation (i.e. the utilisation bound of 50%) of partitioned scheduling yet retains its advantages in that (i) the time complexity of dispatching is independent of processor count and (ii) most tasks do not migrate at all and those few that do, only do so between two processors each and (iii) few preemptions are generated.

The algorithm is configurable with a parameter δ, controlling the frequency of migration of tasks assigned to two processors. The utilisation bound of the algorithm is

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

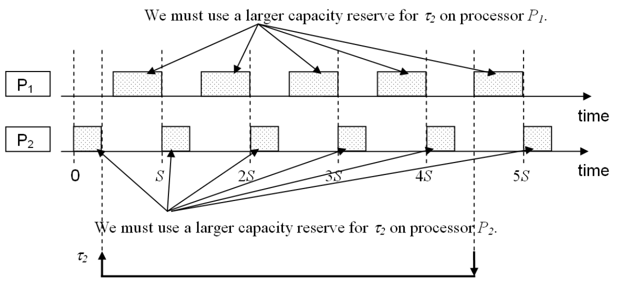
(88% for δ=4). Let TMIN denote the minimum of all T_i. Let $njobs_p(t)$ denote the maximum number, over an interval of length t, of arrivals of jobs assigned (split or non-split) on processor p. During the same interval the algorithm generates at most $3 \cdot \delta \cdot \lceil t/\mathrm{TMIN} \rceil + 2 + njobs_p(t)$ preemptions on processor p.

This paper is organised as follows. Section 2 explores the design of a scheduling algorithm with high utilisation bound and few preemptions leading to our design (Section 3) and proof of its performance. Section 4 explores special cases where the new algorithm can achieve a utilization bound of 100%. Section 5 concludes.



capacity reserved for τ_2 on processor P_1
P1
time
P2
0
S
$2S$
$3S$
$4S$
$5S$
time
capacity reserved for τ_2 on processor P_2

(a) The reserves on processors for the task that is assigned to two processors.



We must use a larger capacity reserve for τ_2 on processor P_1.
P1
time
P2
0
S
$2S$
$3S$
$4S$
$5S$
time
We must use a larger capacity reserve for τ_2 on processor P_2.
τ_2

(b) In order to ensure that the split tasks meet deadlines for all possible arrivals, it is necessary to increase the size of the reserves.

Figure 1. How to perform run-time dispatching of tasks that are assigned to two processors.

2. Background

A task with outstanding computation at time t, is said to be preempted at time t if it executes on processor p just before t but not just after t. With this definition, a job that starts executing is not preempted, nor is one that finishes executing. Also, if a job is executing both just before and just after t but on different processors (hence migrates), with our definition there is preemption at time t. We believe that this captures the notion of preemption used in the research community.

Preemptions are important for meeting deadlines in real-time scheduling on both uniprocessors and multiprocessors. In fact, every non-preemptive scheduling algorithm has a utilisation bound of 0% (see Example 2 in [2]). Although preemption is useful, it is not to be overused as it has associated operating system overheads. Exact preemption costs are application- and architecture-dependent and we will not discuss those (see [7] for excellent coverage).

We will express an upper bound on the count of preemptions in a time interval as a function of its duration and the number of job arrivals within it. By this metric, one can show (see Example 4 in [2]) that pfair scheduling is inherently prone to high preemption counts (as also shown by Devi and Anderson [9]). A promising technique for reducing preemptions is the enforcement of the pfair constraint only upon job arrivals. The algorithm BF [11] and variants [8, 10] do that. Under the EKG algorithm [3], configurable for a utilisation bound of 100%, only a subset of tasks need satisfy the pfair constraint when jobs arrive.

Unfortunately, BF, its variants and EKG were only designed for strictly periodic tasks, whose job arrival times are foreknowable. This permits the calculation of the time budget to be granted to a task over the interval until the next arrival of some job. For sporadic tasks, where the time of next arrival is unknown, this technique cannot be used.

We will thus reason about how to design an algorithm for sporadic tasks; this reasoning takes as starting point partitioned scheduling and the new algorithm based on it is presented in Section 3.

It is well-known that partitioned scheduling has a utilisation bound of at most 50%. For illustrative purposes, the argument is repeated in Example 1.

Example 1. *Consider* $n{=}m{+}1$ *tasks with* $T_i{=}1$ *and* $C_i{=}0.5{+}\epsilon$ *and partitioned scheduling. As* $n{>}m$*, one processor is assigned two or more tasks hence its utilisation is at least* $1{+}2\epsilon{>}100\%$*. If* $m{\to}\infty$ *and* $\epsilon{\to}0$*, the task set has* $U_s{\to}0.5$ *and a deadline is missed. The utilisation bound of every partitioned scheduling algorithm is thus 50% or less.*

This example shows that deadlines can be missed because a task could not be assigned to a processor although there was plenty of idle time in the overall system. The idle time was spread out on different processors and could not be used. However, if a task is given some utilisation on one processor and some on another processor (without executing on both simultaneously) then it is possible to assign tasks such that deadlines are met even with the utilisation on every processor reaching 100%. This approach was used in EKG where many tasks utilise only one processor but a few utilise two. This was ensured by enforcing that the amount of execution of any such task between two consecutive arrivals divided by the time between those is exactly equal to its utilisation on that processor. For sporadic tasks this scheme is impossible as the time of the next arrival is unknown. One may however subdivide the timeline into fixed-length timeslots of duration S equal to TMIN/δ (where TMIN is the shortest interarrival time of all tasks and δ is a positive integer set by the designer) and ensure that each task utilising two processors is given execution proportional to its utilisation in these timeslots (possibly adjusted for worst-case arrival phasings relative to timeslot boundaries). This approach requires no knowledge of future arrival times. See Figure 1.

```
1.  for p:=1 to $m$ do
2.      U[p] := 0
3.      lo_split[p] := 0
4.      hi_split[p] := 0
5.  end for
6.  Let $\tau^{heavy}$ denote the set of tasks such that $C_i/T_i >$ SEP
7.  Let $\tau^{light}$ denote the set of tasks such that $C_i/T_i \leq$ SEP
8.  L := $|\tau^{heavy}|$
9.  Order tasks such that $\tau_i$ with $i$ in 1..L are all in $\tau^{heavy}$
        and $\tau_i$ with $i$ in $L$+1..$n$ are all in $\tau^{light}$
10. if $L \geq m$
11.     declare FAILURE
12. else
13.     for $i$:=1 to L do
14.         p := i
15.         U[p] := U[p] + $C_i/T_i$
16.         $\tau_i$.processor_id1 := p
17.         $\tau_i$.processor_id2 := p
18.     end for
19.     p := L + 1
20.     for i := L+1 to $n$ do
21.         if U[p]+$C_i/T_i \leq$ SEP then
22.             U[p] := U[p] + $C_i/T_i$
23.             $\tau_i$.processor_id1 := p
24.             $\tau_i$.processor_id2 := p
25.         else
26.             if p+1 $\leq$ m then
27.                 hi_split[p] := SEP $-$ U[p]
28.                 lo_split[p+1] := $C_i/T_i$ $-$ hi_split[p]
29.                 $\tau_i$.processor_id1 := p
30.                 $\tau_i$.processor_id2 := p+1
31.                 $U[p]$ := $U[p]$ + hi_split[p]
32.                 $U[p$+1] := $U[p$+1] + lo_split[p+1]
33.                 $p$ := $p$+1
34.             else
35.                 declare FAILURE
36.             endif
37.         end if
38.     end for
39.     declare SUCCESS
40. end if
```

Figure 2. The algorithm for assigning tasks to processors.

3. The new algorithm

We outline our algorithm, then prove its utilisation and preemption count bounds as functions of a parameter δ (set by the designer), balancing utilisation vs preemptions.

3.1 Outline

Recall that n sporadic tasks are to be scheduled on m identical processors. Let TMIN be defined as

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

Our algorithm divides time into slots of length

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

Tasks whose utilisation exceeds some threshold SEP (corresponding to the utilisation bound of our algorithm, whose derivation for a given δ is explored later in the paper) are each granted a dedicated processor – hence never miss deadlines. Remaining tasks are assigned next-fit to remaining processors such that their utilisation is exactly SEP. To achieve this though, whenever assigning a task would cause the utilisation of a processor to exceed SEP, "task splitting" is performed as follows:

Said task (say τ_i) is "split" between that processor (indexed p) and the next one ($p+1$) (i.e. may utilise both but not simultaneously). This is accomplished by reserving the first x time units of each timeslot of length S for τ_i to execute on processor $p+1$ and the last y time units for it to execute on processor p. Reserves x, y for each split task must be appropriately sized so as to (i) rule out any overlap and (ii) ensure the schedulability of the split task and also that (iii) on each processor utilised up to SEP, all non-split tasks (scheduled under EDF whenever their host processor is not executing split tasks) also meet their deadlines.

The offline algorithm for task partitioning and splitting is described in detail in pseudocode in Figure 2. As for the online dispatching algorithm, it is described in high-level pseudocode in Figure 3.

Since dedicated processors for tasks with utilisation above SEP are likewise utilised above SEP and the utilisation of those for remaining tasks is SEP, then if the algorithm ensures that remaining tasks (split or not) are schedulable, it follows that its utilisation bound is at least SEP.

We proceed with the derivation of optimal reserve sizing (given δ) such that any split task will always be schedulable under our algorithm and, subject to this finding, next determine what the maximum value for SEP (given δ) is, for which any task set with utilisation up to SEP will always be schedulable under our algorithm (i.e. its utilisation bound). Last, we derive worst-case preemption counts over any time interval as a function of its length, for a given δ.

3.2 Observations

We observe the following:

```
while TRUE do
  if first_iteration() then //some variables set just once
    p:=get_host_processor();
    hi_task:=get_hi_task(p); //split task run at timeslot end
    lo_task:=get_lo_task(p); //split task run at timeslot start
    y:=optimally_size_reserve(hi_task,p,S); //using Eq. 3
    x:=optimally_size_reserve(lo_task,p,S); //using Eq. 4
  end if

  t:=(current_time()-t_boot) mod S;   //since timeslot start

  if 0≤t<x then //within reserve for low split task
    if has_arrived_but_not_completed(lo_task) then
      execute_lo_task(p);
    else
      execute_non_split_task_with_earliest_deadline();
    end if
  end if

  if x≤t<S-y then //not within a reserve of a split task
    execute_non_split_task_with_earliest_deadline();
  end if

  if S-y≤t<S then //within reserve for high split task
    if has_arrived_but_not_completed(hi_task) then
      execute_hi_task(p);
    else
      execute_non_split_task_with_earliest_deadline();
    end if
  end if

end while
```

Figure 3. A high-level overview of the dispatching algorithm, which runs on every non-dedicated processor

Let x, y be the reserves of some split task τ_i on processors $p+1$, p respectively. Depending on the phasing of the arrival and deadline of τ_i relative to timeslot boundaries, the fraction of time available for τ_i between its arrival and deadline may differ from $\frac{x+y}{S}$. Yet, as reserves are made available periodically and the interarrival time of any task is at least $\delta \cdot S$, said fraction is in any case lower bounded by

$$\min_{\substack{\kappa=\delta \\ \kappa\in\mathbb{N}}}^{\infty} \frac{\kappa \cdot (x+y)}{(\kappa+1)\cdot S-(x+y)} = \frac{\delta\cdot(x+y)}{(\delta+1)\cdot\frac{\text{TMIN}}{\delta}-(x+y)}$$

This is because, any two consecutive windows of execution (consisting of adjacent reserves y and x merged) for the split task are separated by intervals, $S{-}x{-}y$ time units long, of split task idleness, and any interval fully containing κ of the former, may fully contain up to $\kappa{+}1$ of the latter.

Were reserves x and y to be sized such that $\frac{x+y}{S}{=}\frac{C_i}{T_i}$, by inspection, the above lower bound would be less than $\frac{C_i}{T_i}$, thus deadlines could potentially be missed. It is thus necessary to inflate reserves by some factor α so as to always meet deadlines irrespective of arrival phasings relative to timeslot boundaries. These then become:

$$y = S\cdot(\alpha + hi\_split(i)) \tag{3}$$

$$x = S\cdot(\alpha + lo\_split(i)) \tag{4}$$

where hi_split, lo_split denote the respective "uninflated" reserves[1], for which the following holds:

$$hi\_split(i) + lo\_split(i) = \frac{C_i}{T_i} \tag{5}$$

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

$$\frac{\delta\cdot(x+y)}{(\delta+1)\cdot\frac{\text{TMIN}}{\delta}-(x+y)} \geq \frac{C_i}{T_i} \tag{6}$$

Equations 2, 3, 4, 5 and Inequality 6 combined yield:

$$\alpha \geq \frac{U\cdot(1-U)}{2\cdot U+2\cdot\delta} \tag{7}$$

where $U = \frac{C_i}{T_i}$ is the utilisation of the task in question. For a given δ, what is of interest is the optimal (i.e. smallest) value for α (as a function of δ) which will satisfy schedulability for all permissible utilisations. The function

$$\alpha_\delta(U) = \frac{U\cdot(1-U)}{2U+2\delta} \tag{8}$$

has a maximum at $U_0 = \sqrt{\delta\cdot(\delta+1)}-\delta$, which is

$$\alpha_\delta(U_0) = \frac{1}{2} - \sqrt{\delta\cdot(\delta+1)}+\delta \tag{9}$$

The optimal value $\big(\alpha_\delta(U_0)\big)$ for α given δ is plotted in Figure 4 for various values of δ.

Finally, it must be that no temporal overlap is possible either (i) between the reserves on different processors of any given split task τ_i or (ii) between the reserves of different split tasks on the same processor p. For these cases to be satisfied, the following (necessary and sufficient) set of constraints (see Figure 5) applies:

$$(hi\_split(i)+\alpha)+(lo\_split(i)+\alpha) \leq 1\ \forall \text{ split } \tau_i \tag{10}$$

$$(hi\_split[p]+\alpha)+(lo\_split[p]+\alpha) \leq 1\ \forall p \tag{11}$$

[1]Since $hi\_split(i)$ utilises processor p, in which it is associated with the reserve placed at the end of a timeslot, we will occasionally use for it the equivalent notation $hi\_split[p]$, with angular brackets instead of parentheses. Similarly, for $lo\_split(i)$, utilising processor $p+1$, in which it is associated with the reserve towards the start of the timeslot, we will be using $lo\_split[p+1]$.

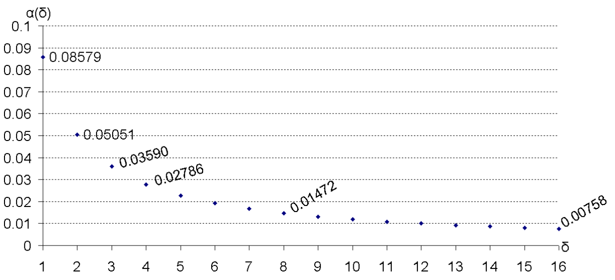


Figure 4. Optimal α as a function of parameter δ

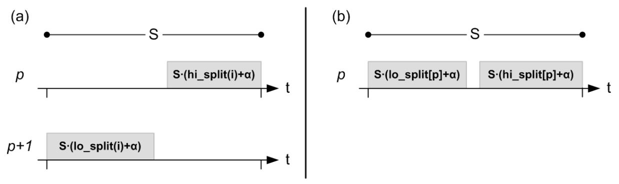


Figure 5. (a) Reserves of the same split task τ_i on processors p and $p+1$. (b) Reserves of different split tasks on the same processor p.

From our earlier assumptions though (see Equation 5), we have

$$hi\_split(i) + lo\_split(i) = \frac{C_i}{T_i} \leq \mathrm{SEP}\ \forall \text{ split } \tau_i$$

and

$$hi\_split[p] + lo\_split[p] = U[p] \leq \mathrm{SEP}\ \forall p$$

Thus, Inequalities 10 and 11 are satisfied if the following (sufficient) condition holds:

$$\mathrm{SEP} \leq 1 - 2 \cdot \alpha \tag{12}$$

Let us then request that the sufficient constraint expressed by Inequality 12 holds, while at the same time not restricting our reasoning to a specific value for SEP within that range.

Recapitulating then, if reserves of all split tasks are inflated by the optimal α, subject to the constraint expressed by Inequality 12, these tasks (which only execute within their reserves but when doing so have priority over non-split tasks) will always be schedulable. The question then arises: What is the maximum utilisation (SEP) for non-dedicated processors such that non-split tasks will likewise always be schedulable? Equivalently, what is the lowest processor utilisation for which it would be possible for non-split tasks to miss deadlines? This, we proceed to derive:

Assume that deadlines on processor p by one or more non-split tasks are in fact missed. Let t be the earliest time that a deadline miss occurs and $t - L < t$ be the earliest instant such that the processor will have been busy during the entire interval $[t - L,\ t)$. Then, let t_{nsp}^{demand} denote the cumulative execution requirement of all jobs of non-split tasks which arrived at $t - L$ or later and whose deadlines lie no later than t, let t_{nsp} denote the time available to (and used up in its entirety by) non-split tasks for execution within the interval $[t - L,\ t)$ and let t_{sp} denote the cumulative execution time of split tasks on processor p over $[t - L,\ t)$. Obviously,

$$t_{nsp} = L - t_{sp} \tag{13}$$

Having a missed deadline by a non-split task is equivalent to

$$t_{nsp}^{demand} > t_{nsp} \tag{14}$$

Regarding t_{nsp}^{demand}, it follows from [6] that

$$t_{nsp}^{demand} \leq \sum_{j \in \mathrm{NS}[p]} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j \tag{15}$$

where $\mathrm{NS}[p]$ is the set of non-split tasks on processor p. Then, by combining Statements 13, 14 and 15 we obtain

$$t_{nsp}^{demand} > t_{nsp} \overset{(13),\ (15)}{\Longrightarrow} \sum_{\tau_j \in NS[p]} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j > L - t_{sp} \tag{16}$$

At this point, we note that

$$\begin{aligned} \sum_{\tau_j \in NS[p]} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j &\leq \sum_{\tau_j \in NS[p]} \frac{L}{T_j} \cdot C_j \\ &= L \cdot \sum_{\tau_j \in NS[p]} \frac{C_j}{T_j} = L \cdot U_{nsp} \end{aligned} \tag{17}$$

with U_{nsp} denoting the cumulative utilisation of non-split tasks on processor p. Then

$$(16) \overset{(17)}{\Rightarrow} L \cdot U_{nsp} > L - t_{sp} \Leftrightarrow U_{nsp} > \frac{L - t_{sp}}{L} \tag{18}$$

Let x, z denote the split task reserves (respectively, low and high) on processor p (which belong to different tasks). We then have

$$x = S \cdot (\alpha + lo\_split[p]) \tag{19}$$

$$z = S \cdot (\alpha + hi\_split[p]) \tag{20}$$

and clearly, since, the cumulative utilisation of split tasks on processor p (equal to $lo\_split[p] + hi\_split[p]$) represents a fraction of the total utilisation of processor p (which in turn does not exceed SEP),

$$U_{sp} \equiv lo\_split[p] + hi\_split[p] \leq \mathrm{SEP} \tag{21}$$

We draw attention to the fact that $L \geq T_i$, with T_i denoting the interarrival time of whichever task misses its deadline (for which in turn it holds that $T_i \geq \mathrm{TMIN} = \delta \cdot S$, from Equations 1 and 2), thus $L \geq \delta \cdot S$. It then holds that

$$\frac{L - t_{sp}}{L} \geq \frac{\delta \cdot (S - x - z)}{\delta \cdot S + (x + z)} \tag{22}$$

as derived by use of Lemma 1 in Appendix A for $\Lambda = L$. (A more intuitive explanation is that any two consecutive windows of execution, corresponding to adjacent reserves z and x merged, for the split task are separated by intervals, $S - x - z$ time units long, of split task idleness, and any time interval fully containing k of the latter there can fully contain up to $k + 1$ of the former.)

Then, by combining Inequalities 18 and 22 we get

$$\frac{\delta \cdot (S - x - z)}{\delta \cdot S + (x + z)} < U_{nsp} \tag{23}$$

Adding U_{sp} to both sides and rewriting, using Equations 19 and 20 yields:

$$U_{sp} + \frac{\delta \cdot S - \delta \cdot S \cdot (U_{sp} + 2 \cdot \alpha)}{S(\delta + U_{sp} + 2 \cdot \alpha)} < U_{sp} + U_{nsp} \tag{24}$$

Further rewriting yields:

$$U_{sp} + U_{nsp} > U_{sp} + \frac{\delta \cdot (1 - U_{sp} - 2 \cdot \alpha)}{\delta + U_{sp} + 2 \cdot \alpha} \tag{25}$$

Then, as long as the cumulative utilisation of split and non-split tasks on some processor (corresponding to $U_{sp} + U_{nsp}$) does not exceed the right-hand side of Inequality 25, it is not possible for any non-split task to miss its deadline. Note that the right-hand side of Inequality 25, which expresses the maximum cumulative utilisation so that no deadlines by non-split tasks are missed, is a function of U_{sp}. In other words, it depends on what fraction of processor utilisation corresponds to split and what to non-split tasks. Then, the utilisation bound (by definition, insensitive to what share of the utilisation is for split/non-split tasks) of our algorithm would correspond to the minimum for the above expression, over the entire range of variable U_{sp}.

The right-hand side of Inequality 25 is a continuous and differentiable function of U_{sp}. If we substitute the optimal α as a function of δ (derived earlier and given by the right-hand side Equation 9) then, using calculus, one may see that it is minimised for

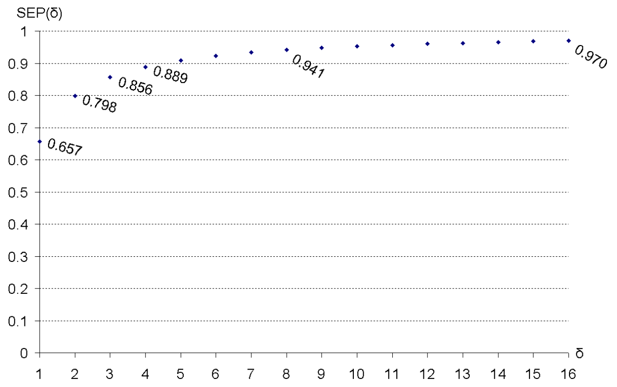


Figure 6. The utilisation bound (SEP) of our algorithm for various values of parameter δ

$$U_{sp} = 3 \cdot (\sqrt{\delta \cdot (\delta + 1)} - \delta) - 1 \tag{26}$$

and that the respective minimum is

$$\mathrm{SEP}(\delta) = 4 \cdot (\sqrt{\delta \cdot (\delta + 1)} - \delta) - 1 \tag{27}$$

Note also that (as can be verified via substitution of the value for α given by Equation 9), it holds that

$$\mathrm{SEP}(\delta) = 1 - 4 \cdot \alpha \tag{28}$$

Figure 6 shows the utilisation bound of our algorithm (SEP) for different values of the parameter δ.

Equation 27 (or, equivalently, Equation 28) thus suggests the highest value that we could use for the variable SEP (already restricted to at most $1 - 2 \cdot \alpha$ by our earlier assumptions, so as to avoid reserve overlap) so that no deadlines (whether by split or non-split taks) are ever missed on non-dedicated processors utilised up to SEP. This means that, if we use the above value for SEP, then the utilisation bound of our algorithm is SEP.

Our reasoning so far thus suffices as proof to the following theorem:

Theorem 1. *Assume that tasks are assigned to processors (and split, where necessary) according to the algorithm of Figure 2 and dispatched using the algorithm in Figure 3, using for α,* SEP *the values dictated by the right-hand side of Equations 9 and 27 respectively, given some value for parameter δ by the designer.*

Then, if the system utilisation does not exceed the utilisation bound given by Equation 27, all deadlines are met and no task executes on two or more processors simultaneously.

3.3 Reasoning about preemptions

An upper bound on the number of preemptions within a given interval as a function of its length is provided by Theorem 2.

Theorem 2. *Assume that tasks are assigned to processors (and split, where necessary) according to the algorithm of Figure 2 and dispatched using the algorithm in Figure 3, using for α, SEP the values dictated by Equations 9 and 27 respectively, given some value for parameter δ by the designer.*

Let $njobs_p(t)$ denote the maximum number of jobs by tasks (split or non-split) assigned to processor p that may arrive during a time interval of length t. Then, during a time interval of length t, the algorithm generates at most $3 \cdot \delta \cdot \lceil t/\mathrm{TMIN} \rceil$ + 2 + $njobs_p(t)$ preemptions on processor p.

Proof. Observe that preemptions may only occur on instants which coincide either with (i) task arrivals or (ii) with the boundaries of split task reserves. An upper bound on the number of such instants within a given time window is thus also an upper bound for the number of preemptions within that time window.

On processor p there can be at most $njobs_p(t)$ instants of the first type $\big($as per the definition of $njobs_p(t)\big)$ within any interval of length t. Regarding the number of timeslot boundaries which are contained within the same interval, we reason as follows:

Every timeslot $[k \cdot S,\ (k+1) \cdot S)\ \forall k \in \mathbb{N}$ contains three instants corresponding to reserve boundaries. Moreover, a time window of length t can overlap (fully or partially) with at most $\lceil \frac{t \cdot \delta}{\mathrm{TMIN}} \rceil \leq \delta \cdot \lceil \frac{t}{\mathrm{TMIN}} \rceil$ timeslots. There can thus be at most $3 \cdot \delta \cdot \lceil \frac{t}{\mathrm{TMIN}} \rceil + 2$ preemptions due to the implementation of the reserves. The additional two preemptions are due to unfavorable alignment of the time window relative to reserve boundaries.

Adding these preemption counts gives us the figure suggested by the theorem.

□

3.4 Discussion of tradeoffs associated with the selection of parameter δ

We showed previously how the utilisation bound of our algorithm increases with δ (see Equation 27). Unfortunately, so does the number of preemptions (see Theorem 2). Let us reason about this tradeoff:

According to Theorem 2, the number of preemptions increases linearly with δ. For practical purposes, this means that in cases where having fewer preemptions matters more than increased utilisation, the designer should opt for a value for δ no more than that past which diminishing returns (in terms of utilisation) occur. Fortunately, as seen in Figure 6, our algorithm attains high utilisations even for very low values of δ.

For example, selecting $\delta = 3$ over $\delta = 4$ results in a 25% reduction of the linear term of the expression for the number of preemptions (see Theorem 2) with a sacrifice of less than 4% in terms utilisation, which remains high, at 85% (see Figure 6).

4. On attaining a utilisation bound of exactly 100%

The right-hand side of Equation 27 can be written as

$$4 \cdot \frac{(\sqrt{\delta \cdot (\delta+1)} - \delta)(\sqrt{\delta \cdot (\delta+1)} + \delta)}{\sqrt{\delta \cdot (\delta+1)} + \delta} - 1$$

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

Then, $\lim\limits_{\delta \to \infty} \mathrm{SEP}(\delta) = \lim\limits_{\delta \to \infty} 4 \cdot \dfrac{1}{\sqrt{1 + \frac{1}{\delta}} + 1} - 1 = 1$

In other words, by increasing δ, one may reach utilisation bounds arbitrarily close to unity (100%, in percentage terms). Yet the question arises: Is it possible (assuming total freedom in the selection of integer parameter δ) for a utilisation bound of *exactly* 100% to be attained, using our algorithm? We will proceed to prove that this is the case if minimum interarrival times of all tasks are a rational multiples of TMIN.

Theorem 3. *If $\forall i \in \{1,\ 2,\ \ldots,\ n\}$ it holds that T_i is a rational multiple of* TMIN, *then, using for integer parameter δ the value $\frac{\mathrm{TMIN}}{s}$, where s denotes the greatest number such that $\forall i$: $\frac{T_i}{s} \in \mathbb{N}^+$, our algorithm, with $\alpha = 0$, has a utilisation bound of 100%.*

Proof. Consider the greatest non-negative number s such that, for every task τ_i in the system, $\frac{T_i}{s}$ is an integer. Such a number always exists if all task interarrival times are rational multiples of TMIN, as we assumed. Let us then choose a timeslot length of $S = s$ or equivalently $\delta = \frac{\mathrm{TMIN}}{s}$ and explore whether a utilisation bound of 100% is attainable.

Previously, we derived an expression for the optimal reserve inflation factor α, as a function of δ (see Equation 9). Yet, that was for the general case, whereas now we additionally assume that all task interarrival times are rational multiples of TMIN and also assume a specific criterion for selecting δ. We first proceed to show that, in this case, the optimal inflation factor is $\alpha = 0$, in other words that (i) any split task will always meet its deadline without any inflation of its reserves and subsequently that (ii) non-split tasks may fully utilise the remaining (i.e. not claimed by split tasks) processor utilisation.

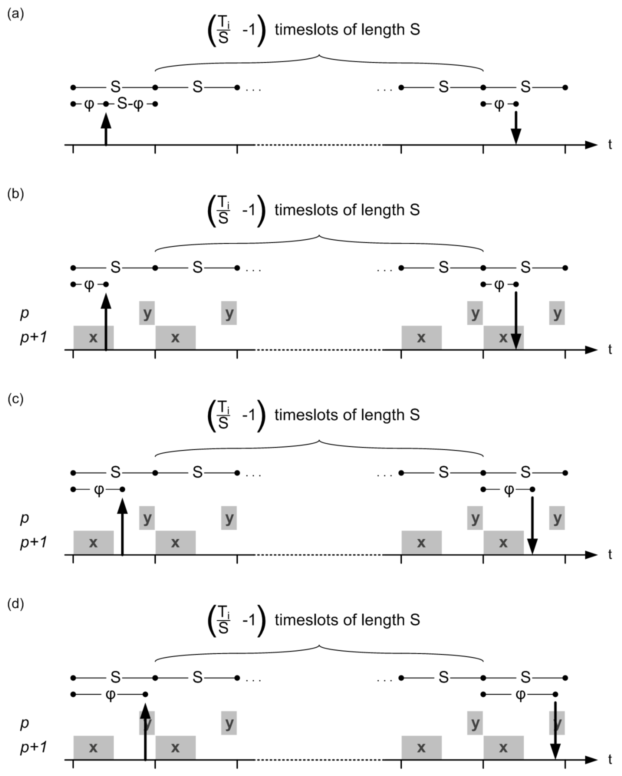


Figure 7. In the case that all task minimum interarrival times divided by the timeslot length yield integers, there is no need for the inflation of the reserves of split tasks, as their schedulability does not depend on their offset relative to timeslot boundaries.

Assume $\alpha = 0$ and consider a split task τ_i arriving φ time units past a timeslot boundary (see Figure 7(a)). It follows that its deadline will also lie φ time units past some other a timeslot boundary. Then the interval between its arrival and deadline spans the last $S - \varphi$ time units of the timeslot in which it arrives, $\frac{T_i}{S} - 1$ subsequent timeslots in their entirety and the first φ time units of the timeslot after those.

Within each of the subsequent $\frac{T_i}{S} - 1$ timeslots to the one in which it arrived, the time reserves for τ_i on the two processors which it utilises are cumulatively $x + y$ time units.

By substituting $\alpha = 0$ into Equations 3 and 4 we obtain

$$y = S \cdot hi\_split(i) \qquad (29)$$

$$x = S \cdot lo\_split(i) \qquad (30)$$

thus, each of the subsequent $\frac{T_i}{S} - 1$ timeslots to the one in which τ_i arrived, provides for the execution of τ_i exactly

$$S \cdot (hi\_split(i) + lo\_split(i)) = S \cdot \frac{C_i}{T_i}$$

time units. It then follows that upon reaching its deadline τ_i will have executed for

$$C_{total} = C_{head} + \left(\frac{T_i}{S} - 1\right) \cdot S \cdot \frac{C_i}{T_i} + C_{tail}$$

time units, where C_{head} is the time available for the execution of τ_i within the interval from its release to the end of the timeslot in which it was released and C_{tail} is the time available for the execution of τ_i within the interval from the start of the timeslot where its deadline lies until its deadline. Note that the above intervals are of the form $[start,\ end)$.

We note three cases:

• Case 1: $0 \leq \varphi < x$
Then (see Figure 7(b)) the last $x - \varphi$ time units of the low reserve for τ_i within the timeslot in which it is released are available to it and also the entire y time units of the respective high reserve, thus $C_{head} = x - \varphi + y$. Additionally, only the first φ time units of the low reserve within the timeslot where the deadline of τ_i lies are available to it and the respective high reserve is entirely unavailable to (as it lies past the deadline), thus $C_{tail} = \varphi$.

• Case 2: $x \leq \varphi < S - y$
Then (see Figure 7(c)) the low reserve for τ_i within the timeslot in which it is released is entirely unavailable to it but the entire y time units of the respective high reserve are available to it, thus $C_{head} = y$. Additionally the entire x time units of the low reserve τ_i within the timeslot where its deadline lies are available to it and the respective high reserve is entirely unavailable to it (as it lies past the deadline), thus $C_{tail} = x$.

• Case 3: $S - y \leq \varphi < S$
Then (see Figure 7(d)) the low reserve for τ_i within the timeslot in which it is released is entirely unavailable to it and only the last $S - \varphi$ time units of the respective high reserve are available to it, thus $C_{head} = S - \varphi$. Additionally the entire x time units of the low reserve τ_i within the timeslot where its deadline lies are available to it and the first $\varphi - (S - y)$ time units of the respective low reserve are also available to it thus $C_{tail} = \varphi - S + x + y$.

In any of the three cases, $C_{head} + C_{tail} = x + y = S \cdot \frac{C_i}{T_i}$, thus

$$C_{total} = \left(\frac{T_i}{S} - 1\right) \cdot S \cdot \frac{C_i}{T_i} + S \cdot \frac{C_i}{T_i} = C_i$$

which proves that the deadline of the split task is met despite zero reserve inflation.

We are now going to prove that no non-split tasks ever miss deadlines in a 100% utilised processor, under the above assumptions.

In uniprocessor EDF with arbitrary deadlines, if a deadline is missed, a necessary and sufficient condition (as per [6] where the arbitrary deadline model was used and D_i denoted the relative deadline) that there exists some

$$L \in \cup_{i=1}^{n} \cup_{\beta \in \mathbb{N}} \{T_i \cdot \beta + D_i\}$$

such that

$$\sum_{j=1}^{n} \max\left(\left\lfloor \frac{L - D_j}{T_j} \right\rfloor + 1, 0\right) \cdot C_j > L$$

where $\mathbb{N}$ denotes the set of natural numbers, i.e. $\{0,\ 1,\ 2, ...\}$ and $\mathbb{N}^{+} = \mathbb{N} \backslash \{0\}$.

In the case that $D_j = T_j, \forall j$, the above is simplified to

$$\exists L \in \cup_{i=1}^{n} \cup_{\beta \in \mathbb{N}^{+}} \{T_i \cdot \beta\} : \sum_{j=1}^{n} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j > L$$

Since however, on any processor p, the non-split tasks, which are scheduled under EDF, also receive interference from the split tasks during time intervals which correspond to the reserves of the latter, in our case, for a deadline to be missed on some processor p, the respective necessary condition is:

$$\exists L \in \cup_{i=1}^{n} \cup_{\beta \in \mathbb{N}^{+}} \{T_i \cdot \beta\} : \\ \sum_{\tau_j \in NS[p]} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j + \left\lceil \frac{L}{S} \right\rceil \cdot S \cdot U_{sp} > L \qquad (31)$$

where $\mathrm{NS}[p]$ is the set of non-split tasks on processor p. Recall that we have selected S such that

$$\frac{T_i}{S} \in \mathbb{N}^{+},\ \forall i \in \{1, 2, .., n\} \qquad (32)$$

Let us consider the value of L in the inequality which forms part of Statement 31. For this value of L, it holds that there exists a task with index k such that

$$\frac{L}{T_k} \in \mathbb{N}^{+} \qquad (33)$$

Applying the value of k on Statement 32 yields:

$$\frac{T_k}{S} \in \mathbb{N}^{+} \qquad (34)$$

Combining Statements 33 and 34 yields:

$$\frac{L}{S} \in \mathbb{N}^{+} \qquad (35)$$

Applying Statement 35 to Statement 31 yields:

$$\exists L \in \cup_{i=1}^{n} \cup_{\beta \in \mathbb{N}^{+}} \{T_i \cdot \beta\} : \\ \sum_{\tau_j \in NS[p]} \left\lfloor \frac{L}{T_j} \right\rfloor \cdot C_j + L \cdot U_{sp} > L \qquad (36)$$

Relaxing and simplifying Statement 36 in turn yields (using the notation that we have used elsewhere):

$$\sum_{\tau_j \in NS[p]} \frac{C_j}{T_j} + U_{sp} > 1 \Leftrightarrow U_{nsp} + U_{sp} > 1 \qquad (37)$$

Taking this all together, we have that if $U_{sp} + U_{nsp} \leq 1$ then all deadlines are met (which proves the theorem). □

Note that Theorem 2 regarding worst-case preemption counts still holds.

An interesting corollary of Theorem 3 is that if all interarrival times are integer multiples of TMIN, our algorithm has a utilisation bound of 100% even with $\delta = 1$ (thus with very few preemptions). On the other hand, for systems where $s \ll \mathrm{TMIN}$, the above approach towards achieving 100% utilisation would be impractical (as $\delta \gg 1$ would be needed, resulting in numerous preemptions).

5. Conclusions

We have proposed an algorithm for scheduling sporadic tasks. It is configurable for a utilisation bound as high as 88% with few generated preemptions. Alternatively, it may be configured for a utilisation bound arbitrarily close to 100% with increased preemptions. In the case that the interrarival times of all tasks are rational multiples of each other, we showed that the utilisation bound can reach exactly 100%. We left open the important question on how to extend this algorithm for sporadic tasks where the deadline of a task is not equal to its minimum interarrival time.

Acknowledgements

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the ARTIST2 Network of Excellence on Embedded Systems Design.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[2] B. Andersson. Sporadic multiprocessor scheduling with few preemptions. Technical report, IPP-HURRAY Research Group. Institute Polytechnic Porto, HURRAY-TR-070501, available online at http://www.hurray.isep.ipp.pt/privfiles/tr-hurray-070501.pdf, May 2007.

[3] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[4] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority EDF scheduling for hard real time. Technical Report TR-050601, Department of Computer Science, Florida State University, Tallahassee, available at http://www.cs.fsu.edu/research/reports/tr-050601.pdf, July 2005.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[6] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[7] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUSRT: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

[8] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 101–110, 2006.

[9] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, 2005.

[10] A. Khemka and R. K. Shyamasundar. Multiprocessor scheduling of periodic tasks in a hard real-time environment. In *Proc. of the 6th International Parallel Processing Symposium*, pages 76–81, 1992.

[11] D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Proc. of the IEEE Real-Time Systems Symposium*, pages 142–151, 2003.

Appendix A

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

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

Proof. Irrespective of the actual value of Λ, t_{sp} (the cumulative time belonging to reserves for split tasks within the interval in consideration) cannot be more than what it would have been within an interval of the same length Λ whose start is offset by $S - z$ time units relative to timeslot boundaries. Then, it holds for t_{sp} that

$$t_{sp} \leq \left\lfloor \frac{\Lambda}{S} \right\rfloor \cdot (x + z) + \min(\Lambda - \left\lfloor \frac{\Lambda}{S} \right\rfloor \cdot S, x + z) \quad (38)$$

The right-hand side of Inequality 38 $\big($which we will denote as $h(\Lambda)\big)$ is a continuous function of $\Lambda \in [\delta \cdot S..\infty)$. It is piecewise differentiable, non-decreasing in intervals $(k \cdot S,\ k \cdot S + (x + z))$ and constant in intervals $(k \cdot S + (x + z),\ (k + 1) \cdot S)$, $\forall k \in \mathbb{N}$.

$$\frac{dh(\Lambda)}{d\Lambda} = \begin{cases} 1, & k \cdot S < \Lambda < k \cdot S + x + z \\ 0, & k \cdot S + x + z < \Lambda < (k + 1) \cdot S \end{cases} \quad (39)$$

Then we have

$$\frac{d}{d\Lambda}\left(\frac{h(\Lambda)}{\Lambda}\right) = \begin{cases} \frac{\Lambda - h(\Lambda)}{\Lambda^2} \geq 0, & k{\cdot}S{<}\Lambda{<}k{\cdot}S{+}x{+}z \\ \frac{-h(\Lambda)}{\Lambda^2} < 0, & k{\cdot}S{+}x{+}z{<}\Lambda{<}(k{+}1){\cdot}S \end{cases} \quad (40)$$

The global maximum for $\frac{h(\Lambda)}{\Lambda}$ over $[\delta{\cdot}S,\ \infty)$ then occurs for some value of Λ belonging to the set

$$\{k \cdot S + (x + z)\} \cap [\delta \cdot S,\ \infty),\ \forall k \in \mathbb{N}$$

which is the same as the set

$$\{k \cdot S + (x + z)\},\ \forall k \in \{\delta,\ \delta + 1,\ \delta + 2,\ ...\}$$

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

$$\left.\frac{h(\Lambda)}{\Lambda}\right|_{\Lambda = k \cdot S + (x+z)} - \left.\frac{h(\Lambda)}{\Lambda}\right|_{\Lambda = (k+1) \cdot S + (x+z)} =$$
$$\frac{(k + 1)(x + z)}{k \cdot S + (x + z)} - \frac{(k + 2)(x + z)}{(k + 1) \cdot S + (x + z)} =$$
$$\frac{(x + z) \cdot S - (x + z)^2}{(k \cdot S + (x + z)) \cdot ((k + 1) \cdot S + (x + z))} =$$
$$\frac{(x + z) \cdot (S - x - z)}{(k \cdot S + (x + z)) \cdot ((k + 1) \cdot S + (x + z))} \geq 0$$

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