Userland Approximate Slack Stealer with Low Time Complexity

Damien Masson, Serge Midonnet

Userland Approximate Slack Stealer with Low Time Complexity

Damien Masson and Serge Midonnet
Université Paris-Est
Laboratoire d'Informatique Gaspard-Monge
UMR 8049 IGM-LabInfo
77454 Marne-la-Vallée Cedex 2, France
{damien.masson, serge.midonnet}@univ-paris-est.fr

Abstract

The aim of our work is to provide a mechanism to deal with soft real-time aperiodic traffic on top of a fixed priority scheduler, without any kind of modification on it. We want to reduce as most as possible the time and implementation complexities of this mechanism. Moreover, the overhead of the framework has to be completely integrated in the feasibility analysis process. We propose a naive but low cost slack time estimation algorithm and discuss the issue of its utilization at the user level. We validate our work by extensive simulations and illustrate its utility by an implementation on top of the Real-Time Specification for Java (RTSJ).

1 Introduction

The need for more flexibility in real-time systems leads the community to address the problem of jointly scheduling hard periodic tasks and soft aperiodic events.

A solution is to set up a mechanism which allows non-periodic traffic to be served and analyzed without changing the feasibility conditions of a periodic task.

The easiest way to achieve this is to schedule all non-periodic tasks at a lower priority (assuming that the tasks are scheduled using a preemptive fixed priority policy). This policy is known as the *Background Servicing (BS)*. If it is very simple to implement, it does not offer satisfying response times for non-periodic tasks, especially if the periodic traffic is important.

Research turns on new scheduling approaches for minimizing the aperiodic tasks response times whilst guaranteeing the feasibility of the periodic tasks.

In this context, the periodic task servers were introduced by LEHOCZKY et al. in [6]. A periodic task server is a periodic task, for which classical response time determination and admission control methods are applicable (with or without modifications). This particular task is in charge of servicing the non-periodic traffic with a limited capacity.

Several types of task server can be found in the literature. They differ by the way the capacity is managed. We can cite the *Polling Server* policy (*PS*), the *Deferrable Server* policy (*DS*), the *Priority Exchange* policy (PE) first described by LEHOCZKY et al. in [6] and developed in [10, 8] and the *Sporadic Server* policy (SS) presented in [9].

In [5], LEHOCZKY and RAMOS-THUEL propose the *Static Slack Stealer*, an algorithm to compute the slack: the maximal amount of time available at instant t to execute aperiodic tasks at the highest priority without endangering the periodic tasks. They demonstrate the optimality of this approach in terms of maximizing the aperiodic tasks response times among non clairvoyant algorithms. Unfortunately, they also admit that its time and memory complexities are much too high for it to be usable.

In his PhD thesis work [4], DAVIS proposes a dynamic algorithm to compute the exact available slack time. Then he demonstrates that the complexity is too high for a real implementation and proposes two approximate slack time computation algorithms, one based on a static approximation, *SASS*, and the other which is dynamic, *DASS*. The dynamic approach presents the advantage to allow gain time[1] to be assimilated in the slack and then transparently reallocated for aperiodic traffic.

All these contributions make the assumption that the developer has the hand on the scheduler, and are in substance enhanced scheduling mechanisms. Unfortunately, the scheduler is most of the time a part of the hardware or of the operating system. That is why we propose a framework based upon these algorithms, but adapted to be implemented in userland, on top of the scheduler.

In this paper, we describe and evaluate the slack stealer part of this framework. Our goal is to provide a slack

[1] if a periodic task has a worst case execution time greater than its mean execution time, most of its executions generate reserved but unused time called *gain time*

stealer approach not only implementable on top of the system scheduler, but also with a very low overhead integrable in the feasibility analysis process.

To limit this overhead, we propose a very simple slack approximation algorithm of the slack which does not use scheduler or system specific data to perform. Moreover, we modify the way the slack is used in order to integrate the slack gesture cost in the worst case execution time of periodic tasks.

We formalize in Section 2 the task model and general notations. We describe in Section 3 the static and dynamic slack stealers, and the Dynamic Approximate Slack Stealer ($DASS$). We present our userland slack estimation in Section 4. Then its userland utilization in Section 5. Our simulations methodology is detailed in Section 6 and results are presented in Section 7. We propose to illustrate this work with an RTSJ implementation in Section 8.

2 Task Model and notations

We consider a process model of a mono processor system, Φ, made up of n periodic tasks, $\Pi = \{\tau_1, ..., \tau_n\}$ scheduled with fixed priorities. Each $\tau_i \in \Pi$ is a sequence of requests for execution characterized by the tuple $\tau_i = (C_i, T_i, D_i, Pi)$, where C_i is the worst case execution time of the request[2]; T_i is its period ; D_i its relative deadline with $D_i \leq T_i$ and P_i its priority, 3 being the highest. Tasks are ordered according to their priority, i.e. $P_1 < P_2 < ... < P_n$. We do not consider blocking factor nor release jitter.

The system also has to serve an unbounded number p of aperiodic requests, $\Gamma = \{\sigma_1, ..., \sigma_p\}$. A request $\sigma_i \in \Gamma$ is characterized by a worst case execution time C_i[2].

The highest priority, 1, is reserved for the task T which implements our mechanism. When an aperiodic request is raised, T registers it in a queue. When slack becomes available, enqueued aperiodic tasks can be released by T with the priority 2. They are then scheduled by the system according to this priority.

3 Static and Dynamic Slack Stealing

The slack stealing technique was first introduced by LEHOCZKY and RAMOS-THUEL in [5]. It consists in determining how much computation time is available at the highest priority without jeopardizing the execution of the periodic tasks. Then this time can be used for aperiodic servicing. Since our mechanism is inspired by this technique, we review it in this section.

[2]for brevity concern, we can denote it *cost* in this paper

3.1 Static Slack Stealing

This slack stealing algorithm is called static because the most significant computations are made off line. However, there is also a run-time phase.

3.1.1 Off line computations

The aim of this first step is to compute for each task a function which gives the task laxity at time t. The laxity of the task τ_i at time t is the sum of aperiodic processing which can be process immediately in preference of τ_i without compromising the respect of its deadline. These functions are noted $A_i(t)$. The function giving the aperiodic processing which can be processed at time t at the highest priority without compromise any deadline is noted $A(t)$. The authors proved Equation 1 and gave an algorithm to compute the $A_i(t)$.

$$A(t) = \min_{1 \leq k \leq n} A_k(t) \tag{1}$$

3.1.2 On line algorithm

$$A(t) = \min_{1 \leq i \leq n} (A_i(t) - \mathcal{I}_i) - \mathcal{A} \tag{2}$$

At time t, the available laxity is given by Equation 2 where $\mathcal{I}_i$ is the *i-level* inactivity and $\mathcal{A}$ the sum of the aperiodic processing time. These two values are maintained up to date each time the CPU begins a new activity.

The available laxity can only be increased when a hard periodic task ends. Hence, the laxity is only computed in two cases: if an aperiodic task is released when the aperiodic queue is empty and if a periodic task ends whilst the aperiodic queue is not empty.

Though this first algorithm is optimal (it provides the exact available time), it has time and memory complexities much too high and it is only applicable to simple systems, without synchronization constraints, resources sharing or sporadic hard real-time tasks. Moreover it can not be extended for gain time reclaiming. That is why DAVIS proposes a dynamic algorithm.

3.2 Dynamic Slack Stealing

This part of DAVIS thesis was first published with TINDELL and BURNS in [3]. The first step to determine the available slack time is to compute for each task the maximum guaranteed slack, $S_i^{max}(t)$, i.e. the maximum amount of slack time which may be stolen at priority level i, during the interval $[t, t + d_i(t))$, whilst guaranteeing that task τ_i meets its deadline. The value $d_i(t)$ denotes the remaining time before τ_i next deadline.

The interval $[t, t + d_i(t))$ is viewed as a succession of *i-level* busy periods[3] and *i-level* idle periods[4]. Then, $S_i^{max}(t)$ is the sum of the *i-level* idle period lengths. The author provides two equations, one to compute the end of a busy period starting at time t, one to compute the length of an idle period starting at time t. To determine $S_i^{max}(t)$, the two equations are recursively applied until the next deadline is reached.

Then, if an aperiodic task is released at time t, let k be the priority of the running hard periodic task, the soft aperiodic task processing can proceed immediately in preference to task τ_k for a duration of $\min_{\forall j \geq k} S_j^{max}(t)$.

This algorithm is not directly usable because of its time complexity. However, the slack computation can be replaced by a slack estimation. The algorithm is then no more optimal but can offer a significant improvement compared to other solutions. DAVIS proposes slack approximation mechanisms, but his algorithms are schedulers, with full knowledge of tasks and current execution parameters. Rather than adapting them, we prefer to set up a naive approximation based on data pieces easily available. Moreover, we want to keep a low computation complexity. Concerning the use of the computed slack, our algorithm is driven by our will to integrate the overhead in the feasibility analysis.

3.3 Dynamic Approximate Slack Stealer

Since $S_i(t)$ is the sum of the i-level idle period lengths in the interval $[t, t+d_i(t))$, DAVIS proposes to estimate this quantity by computing a bound on the maximal interference the task τ_i can suffer in this interval. A bound on this interference is given by the sum of the interferences from each task with a higher priority than τ_i. Then Equation 3 gives the interference suffer by a task τ_j from a task τ_i in an interval $[a, b]$.

$$I_i^j(a,b) = c_i(t) + f_i(a,b)C_i + \min(C_i, (b - x_i(a) - f_i(a,b)T_i)_0) \quad (3)$$

The function $f_i(a, b)$ returns the τ_i instance number which can begins and completes in $[a, b]$. It is given by Equation 4.

$$f_i(a,b) = \left\lfloor \frac{b - x_i(a)}{T_i} \right\rfloor_0 \quad (4)$$

The function $x_i(t)$ represents the first activation of τ_i which follows t. Then the interference is composed by the remaining computation time needed to complete the current pending request, by a number of entire invocations given by $f_i(a, b)$, and by a last partial request.

A lower bound on the $S_i(t)$ value is given by the length of the interval minus the sum of the interferences from each task with a higher or equal priority than τ_i. It is recapitulated by Equation 5.

$$S_i(t) = \left(d_i(t) - t - \sum_{\forall j \leq i} I_j^i(t, d_i(t)) \right)_0 \quad (5)$$

4 Userland algorithm for slack time estimation

We decompose the maximum available slack per task ($S_i^{max}(t)$) into two different pieces of data: first $W_i^{max}(t)$, the maximum possible work at priority i regardless of lower priority processes ; second $c_i(t)$, the effective hard real-time work we have to process at the instant t. The approximation is performed on the first term. To compute a bound on the available slack at time t at the highest priority, $S(t)$, we then have to compute $S_i(t)$ for all priorities and take the minimum. This operation has an $\mathcal{O}(n)$ time complexity.

$$\begin{aligned} S_i^{max}(t) &= W_i^{max}(t) - c_i(t) \\ S_t^{max} \geq S_t &= \min_{1 \leq i \leq n} W_i(t) - c_i(t) \end{aligned} \quad (6)$$

So we have to keep up to date for each periodic task the two values $W_i(t)$ and $c_i(t)$.

4.1 Data initialization

Under hypothesis of a synchronous activation of all periodic hard real-time tasks at t_0, we have:

$$W_1^{max}(t_0) = D_1 \quad (7)$$

$$\forall i, W_i^{max}(t_0) = D_i - \sum_{\forall k < i} \left\lceil \frac{D_i}{T_k} \right\rceil C_k \quad (8)$$

$$\forall i, c_i(t_0) = C_i \quad (9)$$

If we relax the synchronous activation hypothesis, the exact number of τ_k activation between the first release of τ_i and its first deadline, Nb_a, should be computed and equation 8 becomes:

$$\forall i, W_i^{max}(t_0) = D_i - \sum_{\forall k < i} Nb_a C_k \quad (10)$$

The data initialization has a time complexity in $\mathcal{O}(n^2)$, but can be completed before starting the system.

[3] periods where the processor is servicing priorities higher or equal to i

[4] processor idle periods or periods where processor serves priorities lower than i

4.2 Dynamic operations

The values we want to keep up to date are function of time, so they potentially evolve at each clock tick. Between two dates t_1 and t_2, if k is the priority level of the executing task, W_i^{max} is reduced by $dt = (t_2 - t_1)$ for any task with a higher priority than k, and $c_i(t)$ is reduced for τ_k. We approximate W_i^{max} by considering that it is reduced by dt for all tasks. We correct this pessimistic evaluation when a task τ_k ends by adding its cost to W_i for all task with a lesser priority than k. We will see in Section 5 that we only need to compute the slack when a task ends. So we just have to update the W_i when a task completes and the c_i at each context switches. If we do not consider systems with resource sharing, a context switch occurs only when a task begins and when a task ends.

1. **End of periodic task** τ_k. Let dt be the elapsed time since t_{le} the last periodic task ending, i.e. the last update of a W_i. Then, W_i is reduced by dt for all tasks and W_k is increased by T_k and reduced by the interference of higher priority tasks during the next period of τ_k. We consider an upper bound on the maximal interference the task can suffer during one period. This bound, I_k^*, given by equation 11, can be computed during the initialization phase. A closer approximation technique is discussed in Section 4.3. Interference suffers by τ_k before its next activation is already included in the approximation of W_k.

$$I_k \leq I_k^* = \sum_{\forall j<k} \left\lceil \frac{T_k}{T_j} \right\rceil C_j \tag{11}$$

Moreover, for all tasks with a smaller priority than τ_k, W_i is increased by C_k. Finally, $c_k(t)$ is reset to C_k.

$$\begin{array}{ll} \forall i < k, & W_i(t) = W_i(t_{le}) - dt \\ \forall i > k, & W_i(t) = W_i(t_{le}) - dt + C_k \\ i = k, & \left\{ \begin{array}{rcl} W_k(t) & = & W_k(t_{le}) - dt + T_k - I_k^* \\ c_k(t) & = & C_k \end{array} \right. \end{array} \tag{12}$$

These operations are summarized in Equation 12. Their time complexity is $\mathcal{O}(n)$. We will see in Section 4.3 that the time complexity to obtain the interference is also $\mathcal{O}(n)$. This overhead can be a priori added to each periodic task cost since the operations are performed at the end of each instance.

2. **Beginning of periodic task** τ_k.

As in the previous case, the maximum available process time is reduced by dt at each priority levels. However, since we do not need to accurate the W_i values before the next task ending, we do not update these values in this case. Since the last update of data (end or begin of periodic request servicing), there is two possibilities. First, the scheduler can have served periodic traffic, say a τ_j request. Then $c_j(t)$ is reduced by dt. The other possibility is that the system was idle or servicing a soft request. Then there is no need for data update.

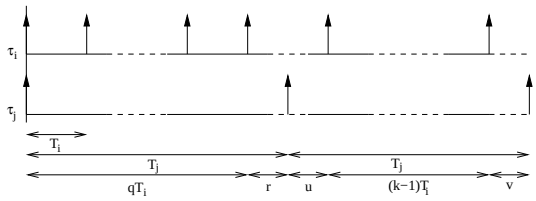

Figure 1. Number of T_i in T_j

$$If \quad j \neq 0, \quad c_j(t) = c_j(t) - dt \tag{13}$$

This operation is summarized in Equation 13. This time, the time complexity is $\mathcal{O}(1)$ and again, the induced overhead can be added to each periodic task cost.

3. **Soft real-time aperiodic task** τ_α **arrival**.

When a soft real-time aperiodic task is released, it is enqueued. Utilization of the slack to serve enqueued aperiodic requests is the topic of Section 5.

4.3 Interference upper bound

Let τ_i and τ_j be two periodic hard real-time tasks with $T_i < T_j$. The priorities are assigned with a Rate Monotonic policy. We discuss in this section of $I_i^j(t)$, the interference task τ_j suffers from task τ_i. More precisely, we give here a way to determine $Na_i^j(t)$, the number of activations of task τ_i during the current period of task τ_j. We have the following properties:

$$I_i^j(t) \leq Na_i^j(t).c_i \tag{14}$$

$$I_j(t) \leq \sum_{h \in hp(j)} I_h^j(t) \tag{15}$$

If we can find the $Na_i^j(t)$ value in constant time, we can determine a bound for the total interference in $\mathcal{O}(n)$ time complexity.

4.3.1 Possible values for $Na_i^j(t)$

Let q and r be the quotient and the remainder in the euclidean division of T_j by T_i. We have $T_j = qT_i + r$. Then,

for any activation of τ_j, let u denotes the time before the next τ_i activation, k the number of τ_i activations before the next τ_j activation and v be equal to $T_j - (k-1)T_i - u$. We have $T_j = u + (k-1)T_i + v$, $u < T_i$ and $v < T_i$.

Figure 1 resumes these notations.

Theorem 1. *There is only two possible values for k, which are $q+1$ and q.*

Proof of $k > q-1$.
Suppose $k \leq q-1$,
$k \leq q-1 \Rightarrow k-1 \leq q-2 \Rightarrow$
$u + (k-1)T_i + v < T_i + (q-2)T_i + T_i$
because $u < T_i$ and $v < T_i$.
Then, since we have $u + (k-1)T_i + v = T_j$, we get $T_j < qT_i$.
This is in contradiction with $T_j = qT_i + r$, $r \geq 0$.
□

Proof of $k < q+2$.
Suppose $k \geq q+2$,
$k \geq q+2 \Rightarrow k-1 \geq q+1 \Rightarrow$
$u + (k-1)T_i + v \geq u + (q+1)T_i + v \Rightarrow$
$T_j \geq (q+1)T_i$, since $u \geq 0$ and $v \geq 0$. This is in contradiction with $T_j = qT_i + r$, $q \geq 1$ and $r < T_i$. □

In conclusion, determining the two possible values is a constant time complexity operation, and can be done offline before starting the system.

4.3.2 Determining the next interference

When a task τ_j ends at time t, we want to find an upper bound on I_i the interference it will suffer from tasks with higher priorities during its next activation. This interference is bounded by the sum of the interferences caused by each task with higher priority. Then the interference caused by task τ_i is bounded by kC_i, where k is the number of τ_i activations during the τ_j next period. We demonstrate that k can be equal to q or $q+1$, where q is the quotient in the euclidean division of T_j by T_i.

In order to determine the correct value of k in this particular case, we first determine the value of u. At time t, let u be the difference between b and a, where a is the next T_j activation date and b the first T_i activation date that follow a. We have:

$$\begin{aligned} a &= \left\lceil \frac{t}{T_j} \right\rceil T_j \\ b &= \left\lceil \frac{a}{T_i} \right\rceil T_i \\ u &= b - a \end{aligned}$$

Now, we just have to compare u and r :

Theorem 2.

$$u \leq r \quad \Rightarrow \quad k = q+1 \tag{16}$$
$$u > r \quad \Rightarrow \quad k = q \tag{17}$$

Proof of Equation 16.
$k = q+1 \Rightarrow k-1 = q \Rightarrow$
$u + (k-1)T_i + v = qT_i + u + v \Rightarrow$
$u + v = r \Rightarrow$
$u \leq r$ since $u \geq 0$ and $v \geq 0$. □

Proof of Equation 17.
$k = q \Rightarrow$
$u + (k-1)T_i + v = (q-1)T_i + u + v \Rightarrow$
$u + v - T_i = r \Rightarrow$
$u > r$ since $v < T_i$. □

If we relax the assumption of the rate monotonic priorities assignment, we can have $T_j < T_i$. Than we have $q = 0$ and $r = T_j$. That gives us:

$$u \leq T_j \quad \Rightarrow \quad k = 1$$
$$u > T_j \quad \Rightarrow \quad k = 0$$

which is correct.

Choosing the correct value is then a constant time complexity operation. We are able to determine the $Na_i^j(t)$ value with a $\mathcal{O}(1)$ time complexity and then the interference with a $\mathcal{O}(n)$ time complexity. Note that the complexity is the same for the exact computation of the interference used by $DASS$ and given by Equation 3. However, the constant number of computations needed is smaller and we do not need any execution related data such as the remaining execution time needed to complete τ_i.

5 Slack utilization in userland

In the slack stealer algorithms proposed by LEHOCZKY and RAMOS-THUEL and then by DAVIS, the slack at the highest priority is computed at the arrival of an aperiodic request, and the request is served in the limit of the slack. In *DASS*, $S_i(t)$ is computed each time a task with a higher or equal priority than τ_i ends an execution, and assumed to have decreased by the elapsed time otherwise. Then the slack is computed in $\mathcal{O}(n)$ time complexity each time an aperiodic arrives. In our userland context, we oppose two objections to these models.

First, if we consider a request with a cost greater than the available slack, we have to start the service of the request, stop it when the slack is exhausted and then resume it later when more slack is available. This is not a problem if we are writing a scheduler, but becomes quite complex in userland. We can off course play with the priority of the task, but

it supposes that the system allows dynamic changes of the priorities. So the first conclusion is that we must wait to have enough slack to schedule the request in one shot.

From that limitation cames the question of the queue policy. Not having enough slack to serve the first arrived aperiodic request does not mean that we not have enough to serve the second one. In our simulations, we test the following policies: *first in first out*, *last in first out*, *lowest cost first* and *highest cost first*.

The second objection is that we cannot compute the slack each time an aperiodic request arrives. Since the number and the arrival model of the aperiodic requests are unknown and the computation of the slack has an $\mathcal{O}(n)$ time complexity, the overhead of this approach cannot be bounded nor incorporated in the feasibility analysis process.

From this second objection cames the question of the best instant to begin a pending request. Several approaches are possible. Two remarks are that the slack can only grows up when a periodic task ends its execution and that in the worst case, after x time unit since the last growth, the slack has decreased by x.

The general conclusion is that we can compute in $\mathcal{O}(n)$ time complexity a close approximation of the slack at each periodic task ending, and assume at any other instant that the slack has just decreased by the amount of elapsed time unit. In this way, the slack estimation computation in $\mathcal{O}(n)$ can be included in the task costs, and the rough estimation perform each time an aperiodic request arrives is just limited to a constant time complexity, inducing a very low overhead. This overhead is not greater than the unavoidable cost of enqueuing (or rejecting) the aperiodic request.

The use of a slack estimation instead of an exact computation and the fact that we have to wait for having enough slack to complete a request before starting it raises a new issue. Some times, the algorithm can return a slack value of 0 time unit although the system is idle. Alternatively, there can exist some situations where the slack has often a non null but low value. In that sort of situations, our algorithm cannot begins the request while a simple BS could have been able to complete it. To limit the side effect of these situations on the average response time of the aperiodic requests, we test a policy we called "duplicate BS". It consists to duplicate all aperiodic requests. One copy is immediately scheduled with BS and the other copy is enqueued in our mechanism. The first of the two to end interrupts the other one.

We call this userland utilization of the slack associated with our minimal approximation of the available slack the Minimal Approximate Slack Stealer ($MASS$) algorithm.

6 Simulations

The validation of the effectiveness of our slack estimation is the first purpose of our simulations. Despite the simplicity of our algorithm induced by the wanted low overhead, we expect results comparable to an exact slack computation based algorithm. The targeted performance metric is the mean response time of the aperiodic requests. To achieve this goal, we simulate the same systems with aperiodic tasks served according to our slack estimation and with the same aperiodic tasks (same release times and same costs) served according to the approximation of *DASS* and with an exact computation of the available slack.

The second expecting result is the validation of the userland exploitation of slack time. The only other available algorithm that does not need to customize the scheduler is the Background Scheduling (BS). So we compare our results with the BS. Moreover, in [7] we proposed a framework for implementing task servers with RTSJ on top of the default scheduler. This framework permitted us to write a Polling server and a Deferrable Server, but with modifications on this two algorithms. We denote the modified algorithms *MPS* and *MDS*. We also compare our results with this ones to know how the slack stealing based approach performs comparing to server based ones.

Finally, we also want to determine the best queue policy through our simulations.

6.1 Methodology

We measure the mean response time of soft tasks with different aperiodic and periodic loads.

First, we generate groups of periodic task sets with utilization levels of 30, 50, 70 and 90%. The results presented in this section are averages over a group of ten task sets. For the same periodic utilization, we repeat generations over a wide range of periodic task set composition, from systems composed by 2 periodic tasks up to systems composed by 100 periodic tasks.

The periods are randomly generated with an exponential distribution in the range [40-2560] time units. Then the costs are randomly generated with an uniform distribution in the range [1-period] In order to test systems with deadlines less than periods, we randomly generate deadlines with an exponential distribution in the range [cost-period]. Priorities are assigned assuming a deadline monotonic policy.

Non feasible systems are rejected, the utilization is computed and systems with an utilization level differing by less than 1% from that required are kept.

For the polling server, we have to find the best period T_s and capacity C_s couple. We try to maximize the system

load U composed by the periodic load U_T and the server load U_S.

$$\begin{aligned} U &= U_T + U_S \\ U &= \sum \frac{C_i}{T_i} + \frac{C_s}{T_s} \end{aligned}$$

A feasible system load being bounded by 1, we have Equation 18.

$$\frac{C_s}{T_s} \leq 1 - \sum \frac{C_i}{T_i} \tag{18}$$

To find a lower bound C_s^{min} of C_s, we first set the period to 2560 (the maximal period). We then search the maximal value for C_s in $[1, 16]$ in order to keep a feasible system. This maximal value is our lower bound on C_s.

Then we seek the lower possible T_s value in $[C_s^{min}/(1 - \sum C_i/T_i), 2560]$. For each T_s value tested, we try decreasing values of C_s in $[C_s^{min}, T_s(1 - \sum C_i/T_i)]$.

Note that it is possible to find a capacity lower than the maximal aperiodic tasks cost. In such cases, since we have to schedule the aperiodic tasks in one shot, we have no solution but to background scheduling the tasks with a cost greater than the server capacity.

For the deferrable server, the methodology is similar, except that since the server has a bandwidth preservation behavior, we do not try to minimize the period and we can search the maximal C_s value in $[1, 2560]$.

Finally, we generate groups of ten aperiodic task sets with a range of utilization levels (plotted on the x-axis in the following graphs). Costs are randomly generated with an exponential distribution in the range [1-16] and arrival times are generated with an uniform distribution in the range [1-100000]. Our simulations end when all soft tasks have been served.

6.2 Real-time Simulator

We develop a real-time event-based system simulator. This is a Java program which can simulate the execution of a real-time system and display a temporal diagram of the simulated execution.

This tool is distributed under the General Public License GNU (GPL), and can be found on the following web page: `http://igm.univ-mlv.fr/~masson/RTSS`

In order to evaluate our slack time evaluation, we add a PFP scheduler which schedules aperiodic tasks according to an exact slack computation and another according to our slack evaluation. The policies simulated take into account the restrictions due to the targeted userland implementation.

7 Results

Figures 2 to 9 present our simulations results. On these figures, ESS, $DASS$ and $MASS$ refer to our slack stealer modified algorithm associated respectively with an exact computation of the available slack time, the slack time approximation given by $DASS$ and our approximation of the available slack time. MPS and MDS designate the modified polling server and deferrable server we developed in a previous work. Finally BS designates the background scheduling associated with a $FIFO$ queue policy and MBS a modified background server which cannot begins a task if a previously started task has not completed. The notation $X\&BS$ refers to the policy X with a BS duplication.

Figures 2 to 5 show the $MASS$ results for all periodic composition systems. We notice that our algorithm performs much better than BS for all policies if the periodic load is low (see Figure 2 and 3). However, when the periodic load increases, the BS duplication became unavoidable. With the extreme 90% load (Figure 5), the non BS-duplicated curves are not viewable on the same graph than the other and are completely out performed by the BS. The explications of this phenomena can be found in Section 5.

The second thing to note is that the queue policy which offers the best results is the LCF one. Due to space limitations and clarity purpose, we cannot put all our results in this paper, but this trend is confirmed by simulations on MPS, MDS, $DASS$ and ESS. This is for sure amplified by the one shot execution limitation. The shorter a task is, the greater is the probability to have quickly enough slack to schedule it completely.

Figures 6 to 9 present the results of the best queue policy for each algorithm (MPS, MDS, $MASS$, $DASS$ and ESS). For ESS, since the time complexity is dependant on the number of task, the results do not include systems composed of more than 40 periodic tasks. For all load conditions, servers bring real improvement comparing to BS. The MDS offers better performances than the MPS. Then, $MASS$ performs better than the servers, $DASS$ better than $MASS$ and ESS better than $DASS$. For systems with periodic loads of 30% and 50%, results obtained with $MASS$, $DASS$ and ESS are quite similar. Considering the differences between the time complexities of these algorithms (constant, linear and pseudo polynomial), this is a very satisfying result. However $MASS$ performances degrade when periodic load increases. Nevertheless $MASS$ remains the best userland-implementable algorithm even for systems with a periodic load of 90%.

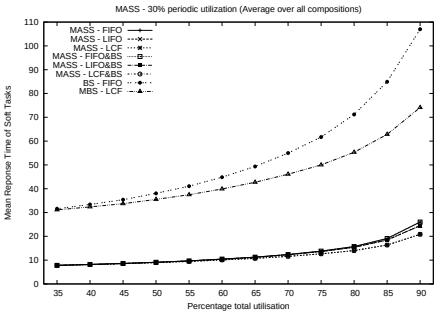

Figure 2. MASS, 30% load

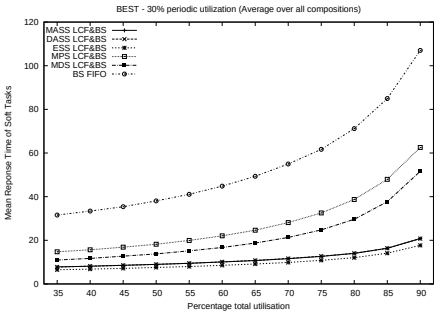

Figure 6. Best policies, 30% load

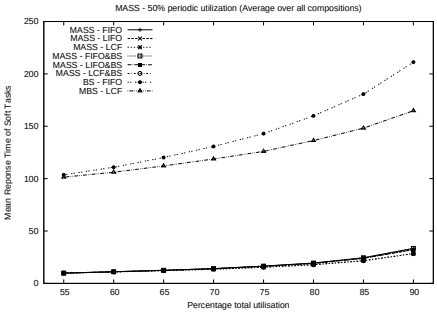

Figure 3. MASS, 50% load

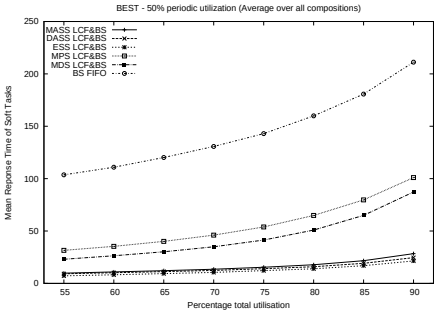

Figure 7. Best policies, 50% load

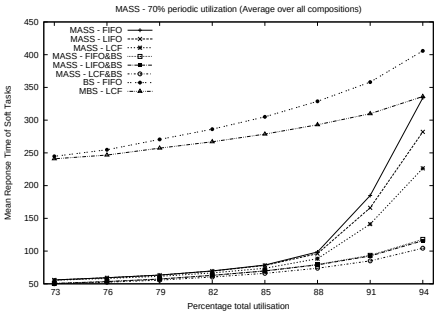

Figure 4. MASS, 70% load

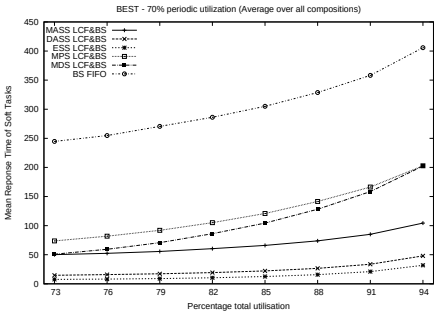

Figure 8. Best policies, 70% load

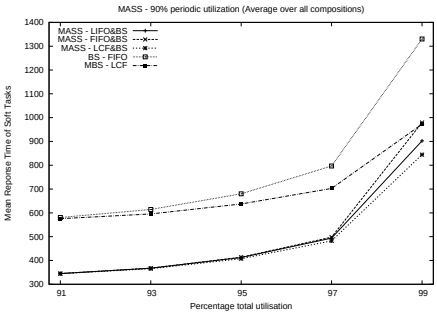

Figure 5. MASS, 90% load

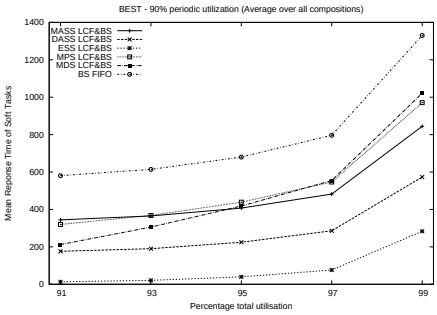

Figure 9. Best policies, 90% load

8 Example: RTSJ implementation

To illustrate the relevance of this work, we propose here an RTSJ [1] implementation of this slack stealer mechanism. Under RTSJ, it is possible theoretically to change the scheduler, but all the implementation of the specification rely on system scheduler. So if you want to use advance algorithms for jointly scheduling soft and hard real-time tasks in a portable application, you have to do it on top of the basic priority scheduler available by default.

8.1 Aperiodic Tasks and RTSJ

The RTSJ proposes two classes `AsyncEvent` and `AsyncEventHandler` to model respectively an asynchronous event and its handler(s). The only way to include an handler in the feasibility process is to treat it as an independent task, and that implies to know at least its worst-case occurring frequency.

The RTSJ does not support any particular task server policy. It also provides the so called "Processing Group Parameters" (PGP), which allows programmer to assign resources to a group of tasks. A PGP object is like a `ReleaseParameters` which is shared between several tasks. More specifically, PGP has a `cost` field which defines a time budget for its associated task set. This budget is replenished periodically, since PGP has also a field `period`. This mechanism provides a way to set up a task server at a logical level. Unfortunately it does not take into account any server policy. Moreover, as pointed in [2], it is far too much permissive and it does not provide appropriate schedulability analysis techniques. Finally, since cost enforcement is an optional feature for an RTSJ-compliant virtual Java machine, PGP can have no effect at all. This is the case with the Timesys Reference Implementation of the specification (RI).

In next Section, we describe our previously published task server framework for RTSJ and we show how we can use it to write a slack stealer.

8.2 Slack Stealer Implementation with RTSJ

In [7], we proposed an RTSJ extension to use task servers. Since we extend this framework to propose a slack stealer, we first describe it in this section.

8.2.1 Task Servers implementation

Our framework (Task Server Framework) is composed of six new classes: `ServableAsyncEvent` (SAE), `ServableAsyncEventHandler` (SAEH), `TaskServer`, `PollingTaskServer`, `DeferrableTaskServer` and `TaskServerParameters`.

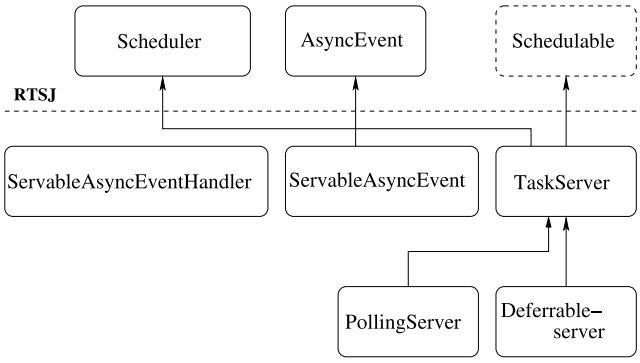

Figure 10. Classes to implement the server policies

Figure 10 shows dependencies between classes in the Task Server Framework and standard RTSJ classes.

To summarize the mechanism, when an SAE is fired, the `servableEventReleased()` methods of the bounded servers are called for each of its SAEHs. This method add the SAEH in the associated task server job queue. Depending of the subclass used, the server can be scheduled periodically or aperiodically to serve the aperiodic jobs in his queue. Several queue policies are possible. This allows developers to write different behaviors for different task server policies.

For example, we proposed a `PollingTaskServer` class which is implemented with a periodic real-time thread and a `DeferrableTaskServer` class with an asynchronous event handler bounded to a special event.

8.2.2 Slack Stealer is a Task Server

We can add a new class `SlackStealerTaskServer` to Figure 10. This class extends `TaskServer`. The slack stealer has to be wake up aperiodically: when an aperiodic task is released while the queue is empty, or when the slack time is increased while the queue is not empty.

The logic of the `SlackStealerTaskServer` class can be delegated to an `AsyncEventHandler` bounded to a special `AsyncEvent`. The method `fire()` of this event is called in the two situations described above.

This design was used for our deferrable server implementation and validated by execution experiments.

9 Conclusions

In this paper, we address the problem of jointly scheduling hard periodic tasks and soft aperiodic events when we cannot change the scheduler and when data relative to the current execution state are not accessible.

We briefly review aperiodic server algorithms. We describe the static and the dynamic exact slack computation algorithms, both theoretically optimal, and the Dynamic Approximate Slack Stealer. We present these algorithms limits in our userland context.

We then propose a simple algorithm to compute in linear time a bound on the available slack time. Since this algorithm operates only on periodic tasks beginning and ending, we show that the induced overhead can be included in these tasks worst case execution times.

The remaining side effect of our mechanism is a constant time operation to perform each time an aperiodic request occurs. This is unavoidable, even if the request is scheduled in background.

We simulate our slack stealer associated to an exact computation of the slack time and associated to our slack approximation. In order to limit the side effect of the approximation, we propose a duplication policy that simulations tend to validate. We compare our simulations with simulations on modified Polling and Deferrable servers we proposed in another publication[7]. These modified server algorithms are also targeted for userland implementation. Our slack stealer is always the most efficient algorithm.

In our simulations, we test four aperiodic tasks queue policies and the most efficient one both for task servers and for slack stealers is the one which schedules in priority the aperiodic task with the lowest cost. This is a behavior amplified by the userland restriction: we have to schedule aperiodic tasks in one shot. Even with very high periodic loads (90%), our algorithm still brings improvements comparing with a BS.

To illustrate the aim of this work, we present an example of application: the implementation of portable task servers and slack stealers with the Real-Time Specification for Java.

We do not consider in this work systems where periodic task costs fluctuate. In such systems, a resource reservation has to be made for the worst case scenario. Each time a task completes earlier than in the worst case, gain time is generated. Even if we did not explain it in this paper, gain time can be integrated in the definition of the slack. Actually, this do not suppose any modification on our slack estimation, since it use dynamic pieces of data.

To continue this work, we will have to consider more complex systems, with resource sharing and precedence constraints. We also have to conduct experiments with real life executions in order to evaluate the exact cost of the BS duplication policy.

References

[1] G. Bollella and J. Gosling. *The Real-Time Specification for Java*, volume 33. Addison-Wesley Publishing, 2000.

[2] A. Burns and A. J. Wellings. Processing group parameters in the real-time specification for java. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 360–370. Springer, 2003.

[3] R. Davis, K. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 222–231, 1993.

[4] R. I. Davis. *On Exploiting Spare Capacity in Hard Real-Time Systems*. PhD thesis, University of York, 1995.

[5] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems. In *proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 110–123, Phoenix, Arizona, December 1992.

[6] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 110–123, San jose, California, December 1987. IEEE Computer Society.

[7] D. Masson and S. Midonnet. The design and implementation of real-time event-based applications with rtsj. In *WPDRTS'07 (in proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium)*, page 148, Long Beach, CA USA, March 2007.

[8] B. Sprunt, J. P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Real-Time Systems Symposium, 1988., Proceedings.*, number 0-8186-4894-5, pages 251–258, Huntsville, AL, USA, December 1988.

[9] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 1:27–60, 1989.

[10] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.