Constant Bandwidth Server Revisited

Luca Abeni
University of Trento
Via Sommarive 9, Povo,
38123 Trento (Italy)
luca.abeni@unitn.it

Giuseppe Lipari
Scuola Superiore Sant'Anna
Piazza Martiri della Libertà 33,
56127 Pisa (Italy)
g.lipari@sssup.it

Juri Lelli
Scuola Superiore Sant'Anna
Piazza Martiri della Libertà 33,
56127 Pisa (Italy)
j.lelli@sssup.it

ABSTRACT

The Constant Bandwidth Server (CBS) is an algorithm for providing temporal protection and real-time guarantees to real-time sporadic tasks. Recently, an implementation of this algorithm called `SCHED_DEADLINE` has been included in the Linux kernel. Therefore, the CBS algorithm is now used to serve more generic tasks than do not obey to the classical sporadic task model. One important type of tasks which was not considered by the original CBS algorithm is the so called "self-suspending task model", where a task instance can suspend itself waiting for an external event. Even if the original algorithm is adapted so that the temporal protection property continues to hold, it is difficult for developers to provide guarantees and to select the most appropriate server parameters for such tasks. This paper investigates the problem of using the CBS algorithm for serving self-suspending tasks, by analysing it from a theoretical point of view and showing how to select the server parameters (budget and periods) for self-suspending tasks. Finally, the effectiveness of these proposals is shown through both simulations and real experiments on Linux / `SCHED_DEADLINE`.

1. INTRODUCTION

The Constant Bandwidth Server (CBS) [1] is a reservation-based scheduling algorithm originally introduced in 1998 to handle tasks characterised by variable execution times and soft real-time requirements.

This goal was achieved by efficiently providing temporal protection between tasks: each task is *reserved* a specified amount of execution time every period, and is guaranteed to have the possibility to execute for the reserved time regardless of the behaviour of the other tasks. This property (which characterises all the reservation-based scheduling algorithms) is respected even if the tasks do not have a regular/periodic activation/execution pattern. Moreover, the CBS allows tasks to execute earlier (consuming some of their *future* reserved time) if there is enough idle time in the system.

The CBS algorithm also provides real-time guarantees to sporadic real-time tasks that respect the so-called Liu and Layland task model [14]: if every task is assigned a budget larger than its worst-case execution time and a server period smaller than the minimum inter-arrival time between two instances, then every instance of the task is guaranteed to complete before its deadline. This property is called *Hard Schedulability*.

For real-time applications, the temporal protection property is as important as the *traditional* memory protection (or address space protection) mechanism provided by general-purpose Operating Systems (OSs). As a matter of fact, some general-purpose OS kernels started to use some temporal protection mechanisms to bound the amount of execution time used by fixed priority (real-time) processes or threads. For example, the Linux kernel provides a mechanism called *RT throttling* to avoid the risk that a high-priority real-time process starves all the other applications in the system. It can be considered as a first necessary step to allow non privileged users to use real-time priorities. Although RT throttling is useful in many situations and can achieve its goals in general, it lacks a strong theoretical foundation; hence, it is not possible to provide a complete schedulability analysis. Moreover, processes or threads characterised by aperiodic / irregular arrival / execution patterns can end up consuming more CPU time than expected, compromising the real-time performance of the other ones (because, for example, of the infamous *deferrable server problem* [20]: the utilisation of a deferrable server is larger than the ratio between runtime and period).

To address this issue, and to allow the predictable execution of real-time application without starving the non real-time ones, an implementation of the CBS algorithm, named `SCHED_DEADLINE` [10], has been recently included in the mainline Linux kernel (and is available without having to patch the kernel since version `3.14`).

Thanks to `SCHED_DEADLINE`, it is possible to use the CBS to schedule every application that can run on a Linux-based OS, including applications that were not originally developed with the Liu and Layland task model in mind. Of

This work has been partially supported by the 7th Framework Programme JUNIPER (FP7-ICT-2011.4.4) project, founded by the European Community under grant agreement n. 318763.

EWiLi'14, November 2014, Lisbon, Portugal.

course, this fact provides new interesting possibilities, but also raises some issues: how to provide real-time guarantees to such tasks (which do not respect the original Liu and Layland model)? How to assign the CBS parameters to them?

One important task model which was not considered by the original CBS is the *self-suspending task model*, in which tasks can self-suspend waiting for external events. For example, a task may suspend waiting for the response of a hardware device, the response of a server or from a co-processor. Tasks that make use of the GPU can be typically modelled as self-suspending tasks. The CBS algorithm and the `SCHED_DEADLINE` implementation, interpret every suspension as an *end-of-instance*; therefore, it is difficult to provide guarantees to self-suspending tasks scheduled by the CBS algorithm, and it is not easy to select the most appropriate budget and period for such tasks.

While the problem of analysing self-suspending tasks has already been addressed in the real-time systems literature [13], we are not aware of other papers that address the same problem in the context of resource reservation algorithms like the CBS.

In this paper, we address this problem first from the theoretical point of view: after recalling the algorithm in Section 2, we propose one alternative rule for the algorithm in Section 2.2. Moreover, in Section 3 we analyse the problem of setting the CBS parameters for a self-suspending task. In Section 4 we show a set of simulation experiments on synthetically generated tasks set to evaluate the proposed methodology. Finally in Section 5 we discuss related work, and in Section 6 we present our conclusions.

2. REVISITING THE CBS

The original CBS algorithm has been designed to schedule real-time tasks which can be described by the so-called "Liu and Layland real-time task model" (each real-time task can be modelled as sequences of jobs that never block). In this section, after quickly recalling the original model and algorithm, we show how the original CBS can be easily modified to serve real-time tasks that do not respect such a strict model.

2.1 Original Task Model and Algorithm

Traditionally, a real-time task τ_i is modelled as a stream of jobs $J_{i,j}$, with the j^{th} job of the task (named $J_{i,j}$) arriving (becoming ready for execution) at time $r_{i,j}$, executing for a time $c_{i,j}$ and finishing at time $f_{i,j}$. Each job is also characterised by an absolute deadline $d_{i,j}$ which is respected if $f_{i,j} \leq d_{i,j}$. Notice that job $J_{i,j}$ never blocks, so it is ready for execution from time $r_{i,j}$ to time $f_{i,j}$.

Based on these definitions (which characterise the previously mentioned "Liu and Layland real-time task model"), the CBS algorithm was defined as follows:

1. A real-time task τ_i can be associated to a CBS (representing a CPU reservation) in order to schedule it
2. The CBS associated to task τ_i is characterised by a budget q_i^s and by an ordered pair (Q_i^s, T_i^s), where Q_i^s is the *maximum budget* and T_i^s is the so called *server period* (or *reservation period*). A *scheduling deadline* d_i^s is also associated to the CBS
3. The CBS associated to task τ_i is said to be *active* at time t if there are pending jobs for task τ_i; that is, if there exist a job $J_{i,j}$ such that $r_{i,j} \leq t < f_{i,j}$. A CBS is said to be idle if it is not active
4. When a job $J_{i,j}$ of a task τ_i served by an active CBS arrives, a request is enqueued
5. When a job $J_{i,j}$ of a task τ_i served by an idle CBS arrives, if $q_i^s \geq (d_i^s - r_{i,j})\frac{Q_i^s}{T_i^s}$ then the CBS generates a new scheduling deadline $d_i^s = r_{i,j} + T_i^s$ and the budget is recharged to the maximum value Q_i^s: $q_i^s = Q_i^s$. Otherwise, the job is served with the current scheduling deadline and budget
6. Whenever a task τ_i served by a CBS executes for a time δ, the budget q_i^s is decreased accordingly: $q_i^s = q_i^s - \delta$
7. When $q_i^s = 0$, it is recharged to the maximum value Q_i^s, and the scheduling deadline is postponed by one server period T_i^s: $q_i^s = Q_i^s, d_i^s = d_i^s + T_i^s$
8. When a job finishes, the next pending job for the task, if any, is served using the current budget and scheduling deadline. If there are no pending jobs, the CBS becomes idle.

According to these rules, every active real-time task is assigned a scheduling deadline d_i^s, which is used by an EDF scheduler to decide which task to execute at any time instant.

If $U = \sum_i \frac{Q_i^s}{T_i^s} \leq 1$, it can be proved that the finishing time of every job will be smaller than the scheduling deadline of the task at the instant of the job completion (see Theorem 1 in [1]). Since the values of the scheduling deadline d_i^s only depends on the parameters of the served task τ_i (that is, on the execution and arrival times of τ_i's jobs) and on the assigned CBS parameters (budget Q_i^s and server period T_i^s), it is possible to prove that *the worst-case behaviour of task τ_i does not depend on the behaviour of the other tasks* (see Lemma 1 in [1]). This is known as *temporal protection* (or *temporal isolation*) property.

Thanks to this temporal protection property provided by the CBS algorithm, it is possible to ensure that all the deadlines of all the jobs of task τ_i will be respected by setting $Q_i^s \geq \max_j\{c_{i,j}\}$ and $T_i^s \leq \min_j\{r_{i,j+1} - r_{i,j}\}$. This is known as *hard schedulability* property. It is also possible to perform a stochastic/probabilistic analysis to provide an upper bound for the probability of missed deadlines for each single task τ_i [2].

2.2 Self-Suspending Tasks

As already mentioned, the algorithm description presented above (which is basically the original CBS) assumes that each real-time task τ_i served by a CBS blocks only when a job finishes, and then wakes up when the next job arrives. Examples can be a periodic task that only blocks waiting for a periodic timer (which is used to activate the various jobs)

or a sporadic task that only blocks waiting for the event that activates the next job. For these tasks, a job $J_{i,j}$ can be described by its arrival time $r_{i,j}$, its deadline $d_{i,j}$, and its execution time $c_{i,j}$.

There are two reasons for this assumption. The first one is the simplicity of implementation in the kernel: every time the task blocks the kernel interprets this event as *end-of-job*; and every time the task is unblocked, the kernel interprets is as a new *job activation*. The second reason concerns the behaviour of the algorithm: if a job blocks on some other event different from a job completion (for example, on a file descriptor waiting for data coming from an external device), the scheduler has to be modified to properly handle this situation so that the CBS properties are not broken. In particular, if the remaining budget and scheduling deadline are not correctly updated the temporal protection property risks to be broken.

However, although the original assumption about non self-suspending jobs can help in simplifying the scheduler, it is not realistic in practice: a job $J_{i,j}$ of a real task τ_i might block (even multiple times) before finishing, hence describing its execution through a single execution time $c_{i,j}$ might be too simplistic. A more realistic task model considers self-suspending tasks, where job $J_{i,j}$ executes for a time $c^0_{i,j}$, then blocks for a time $s^0_{i,j}$, executes for a time $c^1_{i,j}$, blocks for a time $s^1_{i,j}$, etc... In other words, $J_{i,j}$ is composed by $k_{i,j}$ different segments, having execution times $c^0_{i,j} \ldots c^{k_{i,j}}_{i,j}$, and between segment h and segment $h+1$ the job sleeps for a time $s^h_{i,j}$.

In order to be really usable in practice, a modern CBS implementation (such as `SCHED_DEADLINE`) has to correctly support self-suspending tasks, hence both the two problems mentioned above have to be addressed. As a result, it is important to understand how to adjust the budget and the scheduling deadline when a job of the task blocks and unblocks (between two consecutive segments). A first idea could be to use the original CBS "wake up rule" (rule 5) for all the wake-ups (even if a new job is not arrived). Such a rule, which was used by the original CBS algorithm only when a new job $J_{i,j}$ arrives (at time $r_{i,j}$) performs the following check:

$$q^s_i \geq (d^s_i - r_{i,j}) \frac{Q^s_i}{T^s_i}. \quad (1)$$

If the condition holds, it is necessary to compute a new budget and a new scheduling deadline, otherwise the old ones can be re-used without compromising the CBS properties [1, 3].

The rule can be informally explained as follows: the scheduler checks if the fraction of CPU time that the job will use (given by the current budget q^s_i divided by the time $d^s_i - r_{i,j}$ from now to the scheduling deadline) is larger than $\frac{Q^s_i}{T^s_i}$ or not:

$$q^s_i \geq (d^s_i - r_{i,j}) \frac{Q^s_i}{T^s_i} \Rightarrow \frac{q^s_i}{d^s_i - r_{i,j}} \geq \frac{Q^s_i}{T^s_i} \quad (2)$$

If this condition is true, then the task cannot be scheduled using the current scheduling deadline and the current budget, because it would consume a fraction of CPU time larger than Q^s_i/T^s_i, breaking the temporal protection property. The solution used in the original paper was to generate a new scheduling deadline $d^s_i = r_{i,j} + T^s_i$ and a new budget $q^s_i = Q^s_i$ so that the condition is verified.

When considering self-suspending tasks, the same deadline assignment rule can be re-used to check if the current scheduling deadline d^s_i and budget q^s_i can be used when a job wakes up at time t (at the beginning of a new segment). However, other approaches could also be used. For example, it is possible to set

$$q^s_i = (d^s_i - t) \frac{Q^s_i}{T^s_i} \quad (3)$$

and leave d^s_i unchanged thus preserving the bandwidth limit. This will be referred as *revised wake-up rule* in the rest of the paper.

The solution adopted in the original paper makes sense for new job arrivals (because it tries to associate a new deadline equal to $r_{i,j} + T^s_i$ to each job), but can have bad effects on the task's response times when other kinds wake-ups happen. The revised wake-up rule discussed above (decrease the current budget leaving the scheduling deadline unchanged), instead, could be more useful when the wake-up does not correspond to a new job arrival, because it allows to continue serving the current job with the current scheduling deadline.

3. ANALYSIS

When considering only non self-suspending tasks, configuring the CBS parameters Q^s_i and T^s_i is pretty simple: for example (as already mentioned) if $Q^s_i \geq \max_j\{c_{i,j}\}$ and $T^s_i \leq \min_j\{r_{i,j+1} - r_{i,j}\}$ then the deadlines of all the jobs $J_{i,j}$ of task τ_i will be respected (regardless of the behaviours of all the other tasks running in the system). In particular, for periodic tasks τ_i with $r_{i,j+1} - r_{i,j} = P_i$ it is possible to set $T^s_i = P_i$. Although smaller values of T^s_i are sometimes used when performing stochastic analysis [2] or adaptive scheduling [4], $T^s_i = P_i$ is generally preferred in order to reduce the number of context switches and the resulting overhead.

However, when considering self-suspending tasks assigning the CBS parameters might require more care. In order to better understand how to properly schedule a self-suspending task using the CBS algorithm, consider the simplest example of periodic self-suspending task, in which every job has only 2 segments: each job $J_{i,j}$ of the task executes for a time $c^0_{i,j}$, then sleeps for a time $s_{i,j}$, executes for a time $c^1_{i,j}$ and finally finishes. If the server period T^s_i is set equal to P_i, the only thing that can be guaranteed about the execution of the first segment of $J_{i,j}$ is that it will finish before $t' = r_{i,j} + \lceil c^0_{i,j}/Q^s_i \rceil T^s_i - Q^s_i + (c^0_{i,j} \% Q^s_i)$. Hence, the job will sleep from t' to $t' + s_{i,j}$ and the second segment of the job will start at $t'' = t' + s_{i,j}$. Even when considering $Q^s_i >> c^0_{i,j}$, in the worst case the second segment of the job will start at time

$$t'' = r_{i,j} + T^s_i - Q^s_i + c^0_{i,j} + s_{i,j} \quad (4)$$

which if $T^s_i = P_i$ and $c^0_{i,j} + s_{i,j} > Q^s_i$ is larger than $d_{i,j} = r_{i,j} + P_i = r_{i,j} + T^s_i$. Since the actual start time of the second segment of $J_{i,j}$ (as opposed to the worst-case start time) will

depend on the execution of the other tasks in the system, in this case the temporal protection property is not useful to provide real-time performance guarantees to the task. In this case, the difference between the two different "wake-up rules" presented in the previous section does not affect the worst-case performance of the tasks, but only the average performance.

Hence, in order to have a better control on the worst-case real-time performance of self-suspending real-time tasks scheduled by a CBS it could be useful to set $T_i^s < P_i$; in particular, $T_i^s = P_i/R$, with $R \in \mathcal{N}$.

Reducing the value of T_i^s (while keeping the ratio Q_i^s/T_i^s constant), it is possible to have a better control on the jobs' response times (at the cost of a larger number of preemptions). In particular, when $T_i^s \to 0$ (and consequently $Q_i^s \to 0$) the finishing time for the job becomes:

$$\lim_{T_i^s, Q_i^s \to 0} r_{i,j} + \left\lceil \frac{c_{i,j}^0}{Q_i^s} \right\rceil T_i^s - Q_i^s + (c_{i,j}^0 \% Q_i^s) + s_{i,j} + \left\lceil \frac{c_{i,j}^1}{Q_i^s} \right\rceil T_i^s - Q_i^s + (c_{i,j}^1 \% Q_i^s) = = r_{i,j} + \frac{c_{i,j}^0}{Q_i^s} T_i^s + s_{i,j} + \frac{c_{i,j}^1}{Q_i^s} T_i^s$$

because $T_i^s \to 0 \Rightarrow Q_i^s \to 0$, $T_i^s \to 0 \Rightarrow \lceil c_{i,j}^0/Q_i^s \rceil \to c_{i,j}^0/Q_i^s$, and $T_i^s \to 0 \Rightarrow c_{i,j}^0 \% Q^s \to 0$. This is the so called "fluid flow execution model", according to which every task executes in parallel with the other tasks at a speed Q_i^s/T_i^s. Notice that the fluid flow execution model is a mathematical abstraction which is not reasonable in practice ($T_i^s \to 0$ makes no sense in a real system), but provides interesting results for comparison.

According to this model, the first segment of the job executes in $c_{i,j}^0(T_i^s/Q_i^s)$ time units, so the second segment starts a time $t'' = r_{i,j} + c_{i,j}^0(T_i^s/Q_i^s) + s_{i,j}$ and the job finishes at time

$$r_{i,j} + c_{i,j}^0 \frac{T_i^s}{Q_i^s} + s_{i,j} + c_{i,j}^1 \frac{T_i^s}{Q_i^s} = (c_{i,j}^0 + c_{i,j}^1) \frac{T_i^s}{Q_i^s} + s_{i,j} \quad (5)$$

If $C_i^h = \max_j\{c_{i,j}^h\}$ and $S_i = \max_j\{s_{i,j}\}$, then it is possible to say that in the (ideal) fluid flow execution model each job would respect its deadline if

$$\frac{Q_i^s}{T_i^s} = \frac{C_i^0 + C_i^1}{P_i - S_i} \quad (6)$$

hence in a real system (with $T_i^s = P_i/R >> 0$) the ratio between the CBS maximum budget Q_i^s and the server period T_i^s should be larger than $(C_i^0 + C_i^1)/(P_i - S_i)$.

Notice that the discussion above has been performed considering only two segments, but can be generalised: in case of tasks with segments $0...k$,

$$\frac{Q_i^s}{T_i^s} \geq \frac{\sum_{h=0}^{k} C_i^h}{P_i - \sum_{h=0}^{k-1} S_i^h} \quad (7)$$

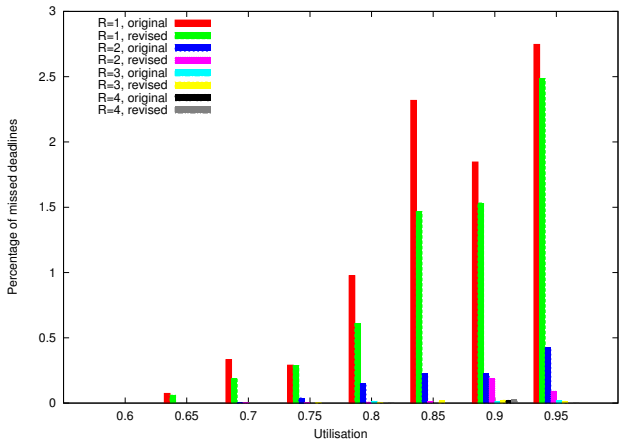

Figure 1: Percentage of missed deadlines for self-suspending tasks served by a CBS, for various configurations of the CBS parameters and wake-up rule.

4. EXPERIMENTAL RESULTS

The suitability of the CBS algorithm (with the two different "wake-up rules" described in this paper and with different P_i/T_i^s values) has been tested through a large set of simulations and real experiments (based on `SCHED_DEADLINE`).

Both simulations and real experiments have been performed by using some randomly generated sets of traditional and self-suspending real-time tasks. In order to generate such task sets, `taskgen` [9] has been used to generate a set of (C_i, P_i) pairs with a given utilisation $U = \sum_i C_i/P_i$. The algorithm generates N randomly distributed numbers $U_i \in (0, 1)$, whose sum is equal to the desired system utilisation: $\sum_i U_i = U$. Then, the periods P_i are randomly generated according to a uniform distribution in $[10ms, 100ms]$ and C_i are set equal to the task utilisation multiplied by the task period: $C_i = U_i P_i$. Some of the pairs are interpreted as "traditional" (non self-suspending) periodic tasks $\tau_i = (C_i, P_i)$, served by CBSs ($Q_i^s = C_i, T_i^s = P_i$), while the remaining pairs represent self-suspending tasks with jobs composed by 2 segments. For each one of these pairs, a sleeping time S_i is randomly generated with a uniform distribution between 0 and $2/3P_i$; then the execution time C_i^0 of the first segment is randomly generated with a uniform distribution between 0 and $(P_i - S_i)C_i/P_i$. Finally, the execution time C_i^1 of the second segment is computed as $C_i^1 = (P_i - S_i)C_i/P_i - C_i^0$. Each self-suspending task is served with a CBS ($Q_i^s = C_i/R, T_i^s = P_i/R$), with $R = 1, 2, 3, 4$. Since these assignments respect Equation 7, a fluid flow schedule would not cause any missed deadline. However, since $T_i^s > 0$ some missed deadlines can be expected, and the next experiments evaluate the impact of R and of the wake-up rule on such missed deadlines.

4.1 Simulations

The simulations have been performed by using RTSim [18], generating 50 sets of tasks for each N, U configuration (where N is the number of tasks and U is the system utilisation).

As an example, Figure 1 shows the probability to miss a deadline (expressed as a percentage) when considering 1 CPU, $N = 6$ tasks, and different values of the utilisation U

Wake-Up Rule	R	Simulation	Experiment
Original	1	0.0097604371	0.0100367111
Original	2	0.0014936333	0.0062934666
Original	3	0.0001274509	0.0025347555
Original	4	0.0000000000	0.0001720666
Revised	1	0.0060903104	0.0073498888
Revised	2	0.0000825723	0.0010361999
Revised	3	0.0000543774	0.0000895111
Revised	4	0.0000075500	0.0001192444

Table 1: Probability to miss a deadline obtained via simulation or through some experiments on a real system.

Wake-Up Rule	R	Deadline Miss Probability
Original	1	0.0005650666
Original	2	0.0000368583
Original	3	0.0000416083
Original	4	0.0000275916
Revised	1	0.0004005916
Revised	2	0.0000000000
Revised	3	0.0000120999
Revised	4	0.0000000000

Table 2: Probability for a self-supending task to miss a deadline, measured in some experiments on a real system using multiple CPU cores.

going from 0.6 to 0.95 (the results obtained on the 50 runs with the 50 different tasksets have been averaged). The figure compares the original CBS and the revised wake-up rule presented in Section 2.2, with $T_i^s = P_i/R$ and $R = 1..4$. From this first experiment, it is possible to notice that:

1. As expected, the deadline miss probability generally increases when increasing the system utilisation U
2. The revised rule generally works better (causes less missed deadlines) than the original CBS
3. Decreasing T_i^s (increasing R) helps to greatly reduce the probability of missing deadlines (arriving to nearly 0 even for high system utilisations)

Also notice that setting $T_i^s = P_i/2$ already allows to reduce the probability to miss a deadline to less than 0.5%, for almost all the values of the utilisation (and using the revised wake-up rule further improves the real-time performance without introducing the additional overhead of higher values of R). Finally, the real-time performance of the "pure periodic" (non self-suspending) tasks have been checked, verifying that such tasks do not miss any deadline; this proves that the temporal protection property is respected.

4.2 Experiments on a Real System

After evaluating the proposed approach through simulations, some experiments have been performed by running real periodic tasks on Linux 3.15.6, with the `SCHED_DEADLINE` scheduling policy. The experiments have been performed on an machine based on an Intel Xeon W3690 CPU (having 6 cores) running at 3.47GHz. The Linux kernel used for the experiments (3.15.6) already includes also the `SCHED_DEADLINE` policy which implements the original CBS algorithm, and has been modified to optionally provide the revised wake-up rule presented in Section 2.2.

The real-time tasks used for the experiments are implemented by an application called `rt-app`. Using this application it is possible to run on a real system the same tasksets used for the previous simulations (of course, in the real system effects like the kernel latency or some other kind of overhead can affect the results). Since each run of each experiment is $60s$ long, the real experiments require much more time than the simulation; hence, only some interesting number of tasks / utilisation configurations have been used.

First of all, `rt-app` has been used to execute on a single core the tasksets simulated for $N = 6$, $U = 0.8$. The results are shown in Table 1. From the table, it is possible to notice that the results obtained in real experiments with small values of R are comparable with the simulation results, proving the accuracy of the simulation model and the correctness of the `SCHED_DEADLINE` implementation. However, notice that increasing R (decreasing the server period T_i^s) the difference between the results of the real experiments and the results of the simulations increases (in particular, the number of missed deadlines in real experiments becomes considerably larger than in simulations), because of the overhead implied by small server periods (which is not modelled in the simulations). In any case, decreasing the server period decreases the probability of missing a deadline, even in real experiments (although the performance improvement is smaller than for simulations).

Finally, notice that even in real experiments the revised CBS is able to improve the real-time performance of self-suspending tasks respect to the original CBS. A more detailed analysis showed that the revised CBS "suffers" in real experiments more than in simulations because in practice the revised wake up rule tends to reduce the current runtime to values which are too small, and the kernel immediately sets it to 0.

Some additional experiments have been performed on multicore configurations (using 4 of the 6 cores available on the Xeon CPU) with `SCHED_DEADLINE` using *global EDF* to schedule the tasks on multiple cores based on their scheduling deadlines. In general, these experiments confirmed the initial results for example, Table 2 reports the deadline miss probabilities for self-suspending tasks measured with 16 tasks and utilisation $U = 3.8$ (remember that with 4 cores the utilisation should be ≤ 4). Again, both reducing the server period and using the revised wake-up rule allow to reduce the number of deadlines missed by self suspending tasks; notice that with $R = 2$ or $R = 4$ the revised rule allows to avoid any missed deadline.

5. RELATED WORK

The reservation approach is not new [15, 19], and has been previously implemented in various Operating System kernels, including Linux [17]. `SCHED_DEADLINE` [10] implements a reservation-based CPU scheduler and is the first example of this kinds of schedulers that has been accepted in the mainstream version of a popular and commonly used OS

kernel. Various different modified version of Linux implementing advanced real-time scheduling algorithms also exist [7, 16, 8], but none of them has been integrated in the mainline version of the kernel. Some other works aim at introducing real-time scheduling algorithms in an existing OS kernel such as Linux without modifying the kernel source [5].

While the original CBS algorithm [1] has been extended in various ways [11] and many more advanced reservation-based algorithms have been proposed in literature [6, 12], `SCHED_DEADLINE` implemented the original algorithm because of its simplicity. The improvements proposed in this paper keep the original algorithm simplicity (less than 30 lines of code have been modified, including comments) while improving the performance as shown in Section 4, and have been inspired by practical usage of the CBS implementation provided by `SCHED_DEADLINE`. Some work to formally analyse the schedulability of self-suspending tasks has been done [13], but it does not consider reservation-based algorithms.

6. CONCLUSIONS

This paper described some issues experienced when using the new `SCHED_DEADLINE` scheduling policy to schedule real tasks. In particular, when considering self-suspending task the strategy used to assign scheduling parameters to tasks should be changed. It can also be argued that the rule used by the original CBS algorithm to handle tasks wake-ups should be revised, in order to properly handle the wake-ups that do not correspond to new job arrivals. Simulations show that both decreasing the server period and using the revised wake-up rule improve the real-time performance of self-suspending tasks, and these results are confirmed by some experiments with real applications running on a Linux-based machine, performed on `SCHED_DEADLINE`.

7. REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni and G. Buttazzo. Stochastic analysis of a reservation-based system. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium.*, San Francisco, California, April 2001.

[3] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, 2004.

[4] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.

[5] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar. Exsched: An external cpu scheduler framework for real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249, Aug 2012.

[6] S. A. Banachowski, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *RTSS*, pages 139–150, 2004.

[7] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. $LITMUS^{RT}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 111–126, Dec 2006.

[8] M. Dellinger, P. Garyali, and B. Ravindran. Chronos linux: a best-effort real-time multiprocessor linux kernel. In *Proceedings of the 48th Design Automation Conference*, pages 474–479. ACM, 2011.

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

[10] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[11] G.Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[12] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421, Washington, DC, USA, 2005. IEEE Computer Society.

[13] C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 425–436, Dec 2009.

[14] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[15] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating systems support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[16] D. Niehaus and N. Watkins. A flexible scheduling framework supporting multiple programming models with arbitrary semantics in linux. In *Proceedings of the Eleventh Real-Time Linux Workshop (RTLWS)*, Dresden, September 2009.

[17] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

[18] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti. An object-oriented tool for simulating distributed real-time control systems. *Software: Practice and Experience*, 32(9):907–932, 2002.

[19] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[20] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *Computers, IEEE Transactions on*, 44(1):73–91, Jan 1995.