Resource Reservation for Real-Time Self-Suspending Tasks: Theory and Practice *

Alessandro Biondi
Scuola Superiore Sant'Anna
Pisa, Italy
alessandro.biondi@sssup.it

Alessio Balsini
Scuola Superiore Sant'Anna
Pisa, Italy
alessio.balsini@sssup.it

Mauro Marinoni
Scuola Superiore Sant'Anna
Pisa, Italy
mauro.marinoni@sssup.it

ABSTRACT

Nowadays, the real-time domain cannot neglect modern hardware architectures and the programming paradigms developed to fully exploit their capabilities. This has shown the limitations of classical task models, like the periodic one proposed by Liu&Layland, and it is pushing for the adoption of more realistic task models and the development of new schedulability analyses to guarantee their timing constraints. Self-suspending tasks are representative of enhanced task models considering explicit suspensions of the execution, happening when a task has to interact with an external device (e.g., through I/O operations) or to access shared resources. Real-time analysis of such a task model cannot neglect to take also into account temporal isolation techniques like bandwidth reservations and hypervisors, required to manage the complexity of actual software and the need of a modular development. In this paper we present a novel scheduling algorithm (H-CBS-SO) that provides temporal isolation for real-time self-suspending tasks. We also propose the implementation of this algorithm in the Linux kernel. Finally, experimental results are presented aiming at evaluating the performance of the implementation in terms of run-time overhead.

1. INTRODUCTION

Modern computation systems are characterized by a growing level of complexity due to an increasing number of cores and the availability of heterogeneous dedicated subsystems. To fully exploit this huge computation power, new programming models and paradigms that highly rely on parallel executions requiring synchronization among the different threads have been developed.

One of the most well known among these approaches is the fork-join one, that is used in a wide range of domains from library for multicore-enabled applications like OpenMP, to Map-Reduce applications that nowadays characterize cloud services. This and other synchronization issues need to be managed with proper protocols [9] that could introduce self-suspensions in the tasks execution. Another aspect is correlated with the presence of dedicate computation units (e.g., FPGA, DSP, GPU), where threads offload highly optimized elaborations to speedup the execution, like the use of a DSP to perform signal processing (e.g., filtering, FFT). Nowadays, applications strongly rely on the communication among nodes and with devices. These could range from sensors acquisitions in the embedded domain to highly interconnected systems in the automotive environment or disk-intensive tasks in the BigData application field. All these behaviors share the necessity for the task to *self-suspend waiting for some event.*

Also real-time applications cannot neglect any more the use of this features, but they need to be provided a way that does not jeopardize timing constraints. The interest regarding scheduling analysis for self-suspending tasks has grown in recent years. However, the currently available results are not comparable with those provided for more classical task model due to the complexity of the problem [22].

Approaches to deal with the schedulability analysis of self-suspending tasks can be dived in two main branches: *suspension-oblivious* and *suspension-aware*. In the *suspension-oblivious* approach [15, 16] the maximum suspension time that each job of a task could endure is included in the worst-case execution time (WCET) of the task. Even if this approach presents the same pessimism in the analysis as the use of busy execution (i.e., the task actively waits for the suspension to finish), it presents advantages at runtime because the task leaves the processor that could be used to reduce response time of other tasks, serve aperiodic request and best-effort activities or be reclaimed to reduce energy consumption. Instead, *suspension-aware* analysis explicitly considers suspensions in the task model and in the related schedulability analysis.

The complexity of modern software solutions has driven the adoption of a modular approach. This is now an established common practice in terms of code design and implementation. In recent years, this view has been introduced also for runtime execution to better exploit the computational power of modern platforms while reducing the complexity of the analysis. Most of the proposed approaches are based on the concept of resource reservation, that assigns a fraction of the computation time provided by the platform to each activity enforcing that no more than such an amount is effectively given. The mechanism could be applied to a huge range of platforms, from small embedded systems to server farms.

The Constant Bandwidth Server (CBS) has been origi-

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

RTNS 2015, November 04-06, 2015, Lille, France

DOI: http://dx.doi.org/10.1145/2834848.2834875

nally proposed by Abeni and Buttazzo [2] for multimedia applications. They proposed it as a scheduling methodology based on reserving a fraction of the processor bandwidth to each task, under the EDF scheduling algorithm. Marzario et al. identified that the CBS is not able to ensure hard reservation due to the deadline aging problem [18]. The Hard CBS [7] (H-CBS) has been proposed extending the original CBS to implement *hard reservation* [20] (i.e., guaranteeing a minimum budget in any time interval). Bertogna et. al. [6,8] presented the BROE algorithm to provide hard real-time guarantee to tasks with an approach light enough to be implemented even in small microcontrollers. BROE extends the H-CBS to handle resource sharing in hard real-time Hierarchical Scheduling Frameworks. Recently, an implementation of the Hard CBS algorithm called SCHED_DEADLINE [10] has been included in the mainline Linux.

Related work on self-suspending tasks. Richard identified that *suspension-aware* schedulability analysis of periodic self-suspending tasks is NP-hard in the strong sense [21]. Ridouard et al. studied the *suspension-aware* schedulability analysis of self-suspending tasks in uniprocessor systems presenting several negative, but very interesting, results [22], [23].

Abdeddaïm and Masson [1] presented a timed automata based model for self-suspending tasks and proposed a method to test the sustainability of a schedule with respect to the execution and self-suspension durations. Recently, Nelissen et al. [19] have invalidated existing results on suspension-aware analysis for uniprocessor systems. In this work they presented an exact analysis for self-suspending task with one self-suspension region and a sufficient test in the case of multiple self-suspension regions, both in case of fixed-priority scheduling.

Liu and Anderson proposed *suspension-aware* schedulability tests for self-suspending tasks in multiprocessor systems [12], [13], addressing both G-EDF and G-FP scheduling policies. Liu and Anderson have also derived a tardiness bound [14] for self-suspending tasks in the context of soft real-time multiprocessor systems under G-EDF and G-FIFO scheduling.

Contribution. This paper has three main contributions:

- we identify that the Hard CBS algorithm (and its current implementation in mainline Linux) is not able to provide resource reservation for self-suspending tasks under *suspension-oblivious* analysis;
- a novel reservation algorithm called H-CBS-SO is proposed, extending the H-CBS to support temporal isolation among self-suspending tasks;
- since the novel algorithm has been implemented in the Linux kernel, implementation details are presented and experimental results aiming at evaluating the performance of the implementation in terms of run-time overhead are reported.

Paper structure. The remainder of the paper is organized as follows. Section 2 presents the system model, a formal definition of the H-CBS scheduling algorithm and the *suspension-oblivious* approach for self-suspending tasks. In section 3 is described the proposed Constant Bandwidth Server (H-CBS-SO) which is able to guarantee temporal isolation for self-suspending tasks. Section 4 describes our implementation of the H-CBS-SO algorithm into the Linux kernel, presenting a set of experimental results. Finally, Section 5 states our conclusions.

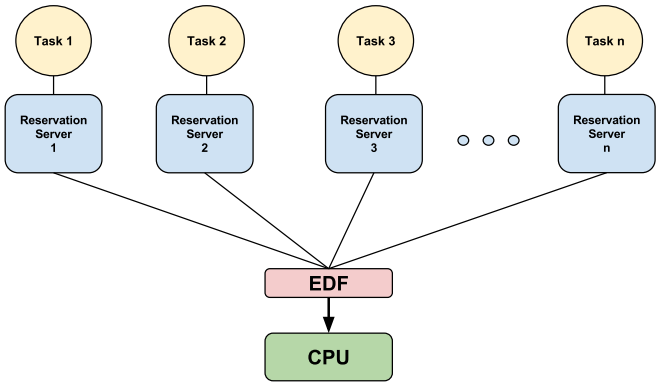

Figure 1: Example of a Task Isolation Framework.

2. BACKGROUND AND NOTATION

This paper considers a taskset Γ composed of n real-time self-suspending tasks (SS-tasks) running upon a uniprocessor system. A SS-task alternates execution and self-suspending phases; no limitation is given to the number of interleaved phases. A SS-task τ_i is characterized by a worst-case execution time (WCET) C_i, a maximum self-suspension time S_i, a period (or minimum interarrival time) T_i and an implicit relative deadline $D_i = T_i$. Each SS-task must start and end with an execution phase (i.e., we made the realistic assumption to not have self-suspensions at the beginning or at the end of the "body" of the SS-task).

Each SS-task τ_i executes upon a dedicated reservation server $\mathcal{S}_i$ characterized by a budget Q_i and a period P_i. In this paper we will consider two different types of reservation server algorithms: the H-CBS algorithm, briefly recalled in the next section, and a novel algorithm, the H-CBS-SO, proposed in Section 3. We assume that the reservation servers are scheduled according to the EDF scheduling policy. An example of the system configuration considered in this paper is illustrated in Figure 1; this scheduling scheme is denoted as Task Isolation Framework (TIF).

2.1 Overview of the H-CBS Algorithm

In this section, the rules of the Hard Constant Bandwidth Server (H-CBS) [7], [18] are described in detail using its budget-based formulation and its fundamental properties are recalled. In the next section we will demonstrate that the H-CBS algorithm is not suitable to directly support a suspension-oblivious analysis of SS-tasks running upon H-CBS servers.

The rules of a H-CBS server with period P and maximum budget Q (bandwidth $\alpha = Q/P$) are summarized below. At any time t, the server is characterized by an absolute deadline $d(t)$ and a remaining budget $q(t)$. When a job executes, $q(t)$ is decreased accordingly. The H-CBS server can be characterized by three states: Idle, Ready and Suspended.

Below we report the algorithmic rules describing the H-CBS server:

Definition 1 (H-CBS Algorithm)**.**

- **Rule 1** Initially, each server is Idle with $q = 0$ and $d = 0$.

- **Rule 2** When the H-CBS server is Idle and a job arrives at time t, a replenishment time is computed as $t_r = d(t) - q(t)/\alpha$:
  1. if $t < t_r$, the server becomes Suspended and it remains suspended until time t_r. At time t_r, the server becomes Ready, the budget is replenished to Q and $d \leftarrow t_r + P$.
  2. otherwise, if the server becomes Ready, the budget is immediately replenished to Q and $d \leftarrow t+P$;

- **Rule 3** When $q = 0$, the server becomes Suspended and is suspended until time d. At time d, the server becomes Ready, the budget is replenished to Q and the deadline is postponed to $d \leftarrow d + P$.

- **Rule 4** When the server has no more pending workload it turns to the Idle state, holding the current values for both budget q and deadline d.

To address the schedulability analysis of the H-CBS server, we recall the following result:

Theorem 1 (Theorem 1 in [18])**.** *Given a set of n H-CBS servers (Q_i, P_i), $1 \leq i \leq n$, all the servers are schedulable under EDF if and only if:*

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1. \quad (1)$$

The H-CBS server can be used to achieve *temporal isolation* among a set of real-time tasks.

Theorem 2 (Theorem 2 in [18])**.** *Given a set of n classical sporadic tasks (i.e., without self-suspensions) having implicit deadline, we associate each task to a H-CBS server $\mathcal{S}_i$. For each H-CBS server $\mathcal{S}_i$, we define $Q_i = C_i$ and $P_i = T_i = D_i$. Then, this set of tasks, each one executing upon a dedicated H-CBS, is schedulable with EDF if and only if the test of Equation* (1) *holds.*

The advantage of associating each task to an H-CBS server is enabling the protection of the system from execution overruns of the tasks. In other words, scheduling the system without the reservation servers, an execution overrun of a task can impact on the schedulability of the whole system, potentially allowing deadline misses on other tasks that are not exceeding their WCET. On the contrary, when each task executes upon a reservation server, all the overruns are protected by the budget exhaustion mechanism that stops the execution of the task. In this way, only the task that is experiencing an overrun is affected in terms of schedulability, while it is possible to guarantee the deadlines of the other tasks.

Proposition 1 (Isolation property of the H-CBS)**.** *Given a set of n tasks, each one running upon a dedicated H-CBS server as in Theorem 2, such that condition (1) holds, if a task experiences an execution overrun (i.e., it exceeds its WCET) then the schedulability of the other tasks will not be affected.*

Concerning self-suspending tasks, to the best of our knowledge no *suspension-aware* schedulability tests have been proposed for SS-tasks executing upon the H-CBS.

2.2 Suspension-oblivious analysis

The exact schedulability analysis of real-time systems with self-suspending tasks has been shown to be a challenging problem, even considering simplified task models executing on a single processor [22], [21]. Of course, all the classical schedulability analyses for sporadic tasks do not work in the presence of self-suspending tasks.

In order to address the schedulability analysis of self-suspending tasks with a tractable complexity, the *suspension-oblivious* analysis approach [15,16] has been proposed. With such an approach, for each self-suspending task, the maximum self-suspension time is accounted as execution time inflating the task WCET. Then, all the typical existing analyses for sporadic tasks can be applied to the whole taskset. Clearly, this approach does not lead to an exact characterization of the schedulability region for SS-tasks, but has a strong effectiveness to derive safe bounds on it.

More formally, this method consists in applying a transformation on the taskset parameters. Let Γ be a taskset of self-suspending tasks as defined in our system model. We define a new taskset Γ^* of typical sporadic tasks as follows: for each task $\tau_i \in \Gamma$ we define a task $\tau_i^* \in \Gamma^*$ having $C_i^* = C_i + S_i$, $T_i^* = T_i$ and $D_i^* = D_i$. Being Γ^* a taskset of typical sporadic tasks, is possible to apply all existing schedulability analysis techniques for such a task model.

Remark 1 (Suspension-oblivious analysis)**.** If the taskset Γ^* is EDF schedulable then also Γ is EDF schedulable.

Overall, since we are accounting self-suspension times as execution times, this approach is pessimistic and allows to provide a sufficient only schedulability test. In fact, during the self-suspensions, we have idle times that could be exploited by additional workload. It is worth observing that, with *suspension-oblivious* analysis, we are modeling the task as it would be executing (i.e., wasting processor cycles) during each self-suspension.

3. H-CBS FOR SUSPENSION-OBLIVIOUS ANALYSIS

When considering SS-tasks executing upon H-CBS servers, one can try to extend the *suspension-oblivious* approach by using the schedulability results from Theorem 1 and 2. That is, given a set of n SS-tasks τ_i, each one executing upon a dedicated H-CBS server $\mathcal{S}_i$, we could configure the server parameters as $Q_i = C_i + S_i$ and $P_i = T_i = D_i$ by accounting self-suspensions as execution times. Then, we could apply Theorem 1 and 2 to verify whether the taskset is schedulable. Unfortunately, the H-CBS algorithm is not directly suitable to support such an approach to deal with the schedulability analysis of self-suspending tasks, as the following example illustrates.

Example 1 (H-CBS and SS-task without busy execution)**.** Consider a task τ with $C = 1$, $S = 3$ and $D = T = 8$. Suppose the task τ be executed upon a reservation server $\mathcal{S}$ having $Q = C + S = 4$ and $P = T = 8$. As Figure 2 shows, the task τ starts executing at $t = 4$, that is the latest time at which τ can start executing still guaranteeing $Q = 4$ budget units until the deadline $P = T = 8$. Suppose now that τ immediately self-suspends its execution: according to the classical H-CBS formulation, since $\mathcal{S}$ has no more workload to be served, the server $\mathcal{S}$ becomes Idle. At time $t = 7$ the task can resume its execution. Hence, Rule 2 of the

H-CBS is applied. In this case we have $t_r = 8 - q(7)/\alpha = 6$: then, having $t > t_r$ (*bandwidth check*), the budget is immediately replenished to Q and the deadline shifted at $d = 15$. In this way, since the server (in the worst-case) cannot start executing before $d - Q = 11$, the task τ will miss its deadline. In other words, the reservation server was not able to guarantee C time units of computation in a period P, notwithstanding that the server budget was set at $Q = C + S$. □

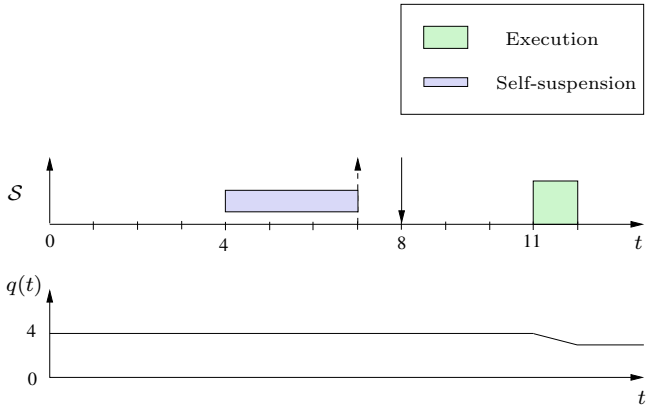

Figure 2: H-CBS serving a SS-task: a counterexample using *suspension-oblivious* analysis.

Since under suspension-oblivious analysis suspensions are treated as execution time, a second approach could be to replace the task self-suspensions with *busy executions*, modifying the actual task implementation. With this approach, when a task has to suspend (e.g., due to an I/O operation), it starts a busy execution wasting processor cycles. The busy execution ends when the task can be resumed from the self-suspension. This solution, illustrated by the following example, is clearly simple and has a strong practical effectiveness.

Example 2 (H-CBS and SS-task with busy execution)**.** Consider a SS-task τ and a H-CBS server as in Example 1. We replace the self-suspension of τ with a busy execution. As shown in Figure 3, when τ self-suspends at time $t = 4$, it continues executing wasting processor cycles until $t = 7$. Thanks to busy execution, the server is now able to guarantee $C = 1$ execution units over a period $P = 8$; this is because the server remains active without performing any *bandwidth check*. □

As illustrated in Example 2, after replacing self-suspension with busy execution Theorem 1 and 2 allow to check for the system schedulability of a SS-task running upon a H-CBS server with $Q = C + S$. However, while replacing self-suspensions with busy executions does not provide any benefits when a *suspension-oblivious* analysis is used, it is clearly worse when other performance metrics are considered. For example, since the busy execution consists in wasting processor cycles, it is not possible to reclaim the idle-time generated from the self-suspensions, which could be used to serve non real-time workload or improve the average response-times of the tasks. In the same way, also when energy constrained systems are addressed, it is preferable to avoid busy executions still guaranteeing the schedulability of the system.

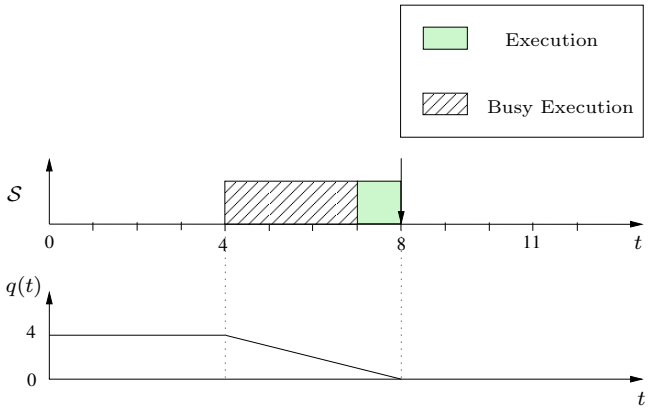

Figure 3: H-CBS serving a SS-task where self-suspensions have been replaced with busy executions.

The key observation is that using busy execution, if a server with pending workload is the highest priority one (i.e., earliest absolute deadline) then it is executing; whilst with self-suspension of servers this is not still true. The highest priority server can be in idle state due to a self-suspension, while in practice this server should not be considered as an idling server since it has pending workload temporarily self-suspended.

Looking at Example 1, we note that deadline miss is caused by the *bandwidth check* ($t > t_r$) that disallows the server to execute in the time interval [7,8]. Unfortunately, as the following example shows, this problem cannot be solved by simply removing the *bandwidth check* when a server is resumed from a self-suspension.

Example 3 (H-CBS without *bandwidth check* and SS-tasks)**.** Consider a taskset composed by two self-suspending tasks, τ_1 having $C_1 = 2, S_1 = 0, T_1 = D_1 = 4$, and τ_2 having $C_2 = 2, S_2 = 1, T_2 = D_2 = 7$. Using *suspension-oblivious* analysis this taskset results schedulable under EDF, since

$$\sum_{i=1}^{2} \frac{C_i + S_i}{T_i} = \frac{2}{4} + \frac{2+1}{7} \leq 1. \tag{2}$$

Suppose that both τ_1 and τ_2 are associated to two H-CBS servers $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with $Q_1 = C_1 + S_1 = 2$ and $Q_2 = C_2 + S_2 = 3$. The period of the servers is the same of the served task. As Figure 4 shows, both the servers are released at the same time; then, according to the EDF policy, $\mathcal{S}_1$ starts executing. At time $t = 2$ the server $\mathcal{S}_2$ can start to execute and immediately self-suspends its execution. Suppose now that τ_2 violates its maximum self-suspension time, self-suspending its execution for 2 time units. Then, at time $t = 4$, τ_2 resumes its execution without performing the *bandwidth check* and executes for 3 time units, making 1 time unit of overrun. Since the budget of the server $\mathcal{S}_2$ was set to $Q_2 = 3$, the task τ_2 is able to overrun without any budget exhaustion mechanism is triggered. Finally, at time $t = 7$, τ_1 can start executing missing its deadline at time $t = 8$. Unfortunately, since a greater self-suspension time and an overrun of τ_2 compromise the schedulability of τ_1, this example shows that the task isolation property (Proposition 1) of the H-CBS could be broken. □

To address the problems discussed in this section, we propose an extension of the H-CBS algorithm which "takes into account" task self-suspensions exactly as the task was busy

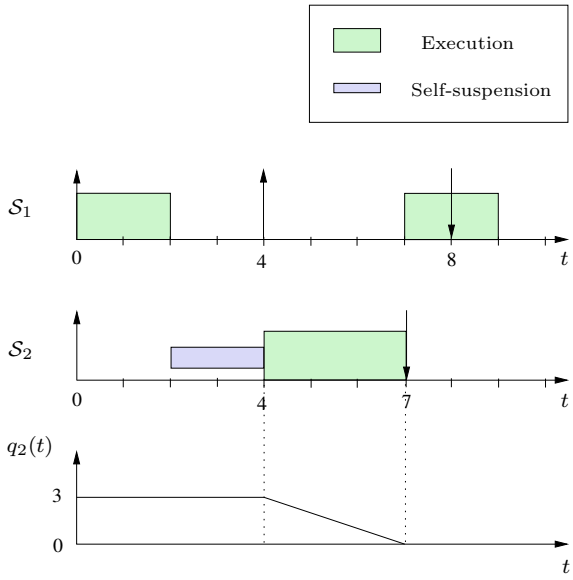

Figure 4: H-CBS with SS-tasks: the *bandwidth check* of the H-CBS is disabled.

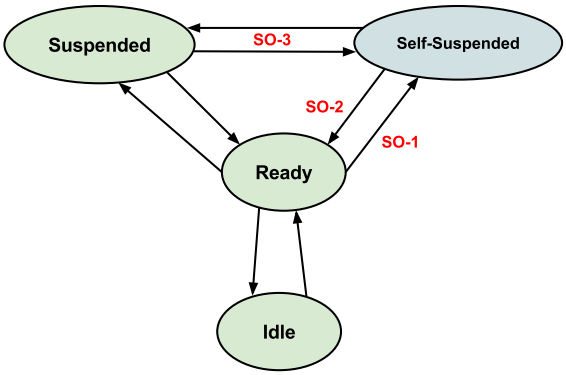

Figure 5: State transition diagram for the H-CBS-SO algorithm.

executing. In this way we can obtain a reservation server algorithm that can be analyzable with a *suspension-oblivious* analysis.

3.1 The H-CBS-SO algorithm

In this section we define the H-CBS-SO algorithm, an extension of the H-CBS algorithm to support self-suspending tasks analyzable with a *suspension-oblivious* analysis, and we derive its basic properties.

The H-CBS-SO server introduces a new state (with respect to H-CBS) denoted Self-Suspended. As Figure 5 illustrates, this new state can be reached by transitions from the Ready and the Suspended states; the Self-Suspended state allows transitions to the Ready and the Suspended states.

The H-CBS-SO extends the H-CBS algorithm reported in Definition 1 by adding the following rules and the SS-QUEUE data structure:

Definition 2 (H-CBS-SO Algorithm)**.**

SS-QUEUE. A queue containing all the server in the Self-Suspended state is introduced; the server in that queue are ordered by increasing absolute deadline. We denote with S_{SS} the server on top of SS-QUEUE, d_{SS} and q_{SS} its absolute deadline and budget respectively. If SS-QUEUE is empty, we set $d_{SS} = q_{SS} = \infty$.

- **Rule SO-1**. When a task τ self-suspends, the server associated to τ becomes Self-Suspended and is inserted in the SS-QUEUE.
- **Rule SO-2**. When a task τ resumes form self-suspension, the server associated to τ becomes Ready and is removed from the SS-QUEUE.
- **Rule SO-3**. When $q_{SS} = 0$, the server $\mathcal{S}_{SS}$ is removed from the SS-QUEUE, becomes Suspended and is suspended until time d_{SS}. We rename $\mathcal{S}_{SS}$ by $\mathcal{S}_j$. At time d_j, the budget q_j is replenished to Q_j and the deadline d_j is postponed to $d_j + P_j$; finally, $\mathcal{S}_j$ becomes Self-Suspended and is inserted in the SS-QUEUE.
- **Rule SO-4**. (Budget accounting rule) - If a server S_i is executing and $d_i \geq d_{SS}$ holds, then *both* q_{SS} and q_i are decreased. On the other hand, if $d_i < d_{SS}$ only q_i is decreased. If there is no server in the Ready state (idle-time), then q_{SS} is decremented.

Notice that Rules SO-1 and SO-2 do not modify the deadline of the server. Also, notice that, when the server is moved from the Self-Suspended state to the Ready state, Rule SO-2 does not perform any *bandwidth check* (Rule 2 of H-CBS).

Example 4 (H-CBS-SO and SS-tasks, following Example 1)**.** Consider the same SS-task τ of Example 1 executing upon a server $\mathcal{S}$ implemented with the H-CBS-SO algorithm. Suppose also that τ starts executing at $t = 4$ as in the previous example and immediately self-suspends its execution. According to the H-CBS-SO algorithm, the server $\mathcal{S}$ becomes self-suspended and is inserted in SS-QUEUE. Then, Rule SO-3 is applied and the budget $q(t)$ is decremented until τ resumes its execution (time $t = 7$). At this time, Rule SO-2 is applied and $\mathcal{S}$ becomes Ready with budget $q = 1$. Since Rule SO-2 does not provide any bandwidth check, the server $\mathcal{S}$ can execute until time $t = 8$ avoiding τ to miss its deadline. □

Example 5 (H-CBS-SO and SS-tasks, following Example 3)**.** Consider the same taskset of Example 3 where the servers $\mathcal{S}_1$ and $\mathcal{S}_2$ are implemented with the H-CBS-SO algorithm. Both the servers are configured with the same parameters of the previous example. At time $t = 2$ the server $\mathcal{S}_2$ can start to execute and immediately self-suspends its execution; then, according to Rule SO-1, $\mathcal{S}_2$ becomes Self-Suspended and is inserted in SS-QUEUE. Suppose now that τ_2 violates its maximum self-suspension time, self-suspending its execution for 2 time units. During the self-suspension, Rule SO-3 is applied and the budget q_2 is decremented by 2 time units. Then, at time $t = 4$, τ_2 resumes its execution with budget of $\mathcal{S}_2$ equal to $q_2 = 3 - 2 = 1$.

Likewise Example 3, we suppose that τ wants to execute for 3 time units, making 1 time unit of overrun. In this case, having used the H-CBS-SO algorithm, we can note that τ is able to execute only in the time interval $[4, 5]$ due to a budget exhaustion at time $t = 5$.

Finally, at time $t = 5$, τ_1 can start executing without missing its deadline at time $t = 8$. Unlike Example 3, in this case a greater self-suspension time and an overrun of

τ_2 do not compromise the schedulability of τ_1, guaranteeing the task isolation property. □

The rules of the H-CBS-SO have been designed to imitate the parameters updating of a server in the H-CBS with busy execution algorithm, but removing the wasting of processor cycles typical of the busy execution. This property is related to the H-CBS-SO server having earlier deadline, since it is the one that would have executed when self-suspensions are replaced with busy executions. This is expressed in the following proposition, which is the main property of the H-CBS-SO algorithm.

Proposition 2. *With H-CBS-SO, self-suspensions of the server having the earlier deadline are accounted as with H-CBS with busy executions.*

Proof. We first consider the state-transitions from Ready to Self-Suspended and from Self-Suspended to Ready (Rules SO-1 and SO-2, respectively). Let us consider a server $\mathcal{S}_{SS}$ in the Ready state and executing (d_{SS} is the earliest deadline) serving an active task τ_{SS} that self-suspends. Then Rule SO-1 is applied and the server becomes Self-Suspended. We distinguish two cases: (i) a server $\mathcal{S}_i$ executes while $\mathcal{S}_{SS}$ is Self-Suspended (ii) there is no such server $\mathcal{S}_i$ while $\mathcal{S}_{SS}$ is Self-Suspended (idle-time). In case (i), let us first suppose $d_i < d_{SS}$. Then with H-CBS with busy execution $\mathcal{S}_{SS}$ is preempted by $\mathcal{S}_i$ and q_{SS} is not decremented. Similarly, Rule SO-4 implies that q_{SS} is not decremented while $\mathcal{S}_{SS}$ is in the Self-Suspended state. Let us instead suppose $d_i \geq d_{SS}$. Then with H-CBS with busy exection $\mathcal{S}_{SS}$ continues executing (i.e., $\mathcal{S}_i$ can not preempt $\mathcal{S}_{SS}$) and the budget q_{SS} is decremented accordingly. Similarly, Rule SO-4 implies that q_{SS} is decremented. In case (ii), with H-CBS with busy execution $\mathcal{S}_{SS}$ continues (busy) executing (there is no idle-time) and the budget q_{SS} is decremented accordingly. Similarly, Rule SO-4 implies that q_{SS} is decremented. Finally, let us consider a server $\mathcal{S}_{SS}$ in the Self-Suspended state and resuming its execution. Then Rule SO-2 is applied and the server becomes Ready without performing any budget check. As with H-CBS with busy execution, $\mathcal{S}_{SS}$ continues executing with (unchanged) deadline d_{SS} and budget q_{SS}.

We now consider the remaining state-transitions: from Self-Suspended to Suspended and from Suspended to Self-Suspended (Rule SO-3). Let us consider a server $\mathcal{S}_{SS}$ in the Self-Suspended state and exhausting its budget q_{SS} ($q_{SS} = 0$, after applying Rule SO-4). By Rule SO-3, $\mathcal{S}_{SS}$ is removed from the SS-QUEUE, becomes Suspended and is suspended until d_{SS}; the budget q_{SS} is then no more decremented by applying Rule SO-4. A similar result holds with H-CBS with busy execution by applying Rule 3. Let us consider a server $\mathcal{S}_j$ in the Suspended state serving a self-suspended SS-task τ_j at time d_j. By Rule SO-3, the budget q_j is replenished to Q_j and the deadline is postponed to $d_j + P_j$. A similar result holds with H-CBS with busy execution by applying Rule 3. Finally, Rule SO-3 inserts $\mathcal{S}_j$ into the SS-QUEUE, thus leading back to the cases described in the first part of the proof. □

When self-suspensions are replaced by busy executions, it is clearly not possible to have nested self-suspensions, because the processor would be occupied by the busy execution in place of the self-suspension of the server having the earliest deadline. When nested self-suspensions occur, the H-CBS-SO server does not imitate the parameters updating of the H-CBS with busy executions. However, the following proposition addresses this point showing that nested self-suspensions do not affect the system schedulability.

Proposition 3. *If a deadline is missed by a H-CBS-SO server, then it would be missed also with H-CBS with busy executions.*

Proof. Since H-CBS-SO extends H-CBS by adding the Self-Suspended state and rules describing transition to and from this state, we have to consider only the behavior of the H-CBS-SO server triggered by self-suspensions. We distinguish two cases: non-nested self-suspensions and nested self-suspensions. In the first case, since we have only a single self-suspension, Proposition 2 guarantees that the H-CBS-SO has the same behavior of H-CBS with busy executions. Consider now the second case: suppose $\mathcal{S}_j$ be a H-CBS-SO server that is self-suspended when at least another server is self-suspended (*i.e.,* nested self-suspension). Let SS be the set of self-suspended servers having deadlines $d_i \leq d_j,\ \forall \mathcal{S}_i \in SS$. If the set SS is empty, then $\mathcal{S}_j$ is the self-suspended server having earlier deadline, and the considerations of Proposition 2 can be applied to replicate the behavior of the H-CBS with busy execution. From now on suppose that the set SS is not empty, hence considering the case in which $\mathcal{S}_j$ is in the SS-QUEUE and it is not the first server in SS-QUEUE. Let t be the earlier time instant at which $\mathcal{S}_j$ is resumed from its self-suspension or the set SS becomes empty, which is the time instant at which the server $\mathcal{S}_j$ stops to have a budget update different from the case of H-CBS with busy execution (i.e., also the time at which $\mathcal{S}_j$ is removed from the SS-QUEUE or it becomes the first server in SS-QUEUE).

If the deadline d_j is missed, then $\mathcal{S}_j$ was not able to execute $q_j(t)$ before time d_j, i.e., $q_j(t) + I > (d_j - t)$, where I is the interference caused by servers having earlier deadline than $\mathcal{S}_j$ from time t on.

In case of using the H-CBS with busy execution, due to the busy execution of self-suspended servers in the set SS, we note that the budget $q_j^{'}(t)$ of $\mathcal{S}_j$ at time t is greater or equal than the one with H-CBS-SO, i.e., $q_j^{'}(t) \geq q_j(t)$. This is because server $\mathcal{S}_j$ could have executed during self-suspensions of the servers in the set SS, while it would be prevented from executing in case of self-suspensions replaced by busy executions.

Hence, if $q_j(t) + I > (d_j - t)$ then also $q_j^{'}(t) + I > (d_j - t)$ holds, thus concluding the proof. □

Proposition 3 allows to use the schedulability test from Theorem 2 for SS-tasks after inflating their WCETs with the worst-case self-suspension times.

Theorem 3 (Suspension-Oblivious Analysis)**.** *Given a set of n SS-tasks τ_i with implicit deadline, we associate each SS-task to a H-CBS-SO server $\mathcal{S}_i$. For each H-CBS-SO server $\mathcal{S}_i$, we define $Q_i = C_i + S_i$ and $P_i = T_i = D_i$. Then, this set of tasks, each one executing upon a dedicated H-CBS-SO, is schedulable with EDF if Equation* (1) *holds.*

Notice that the schedulability condition from Theorem 3 is sufficient but, in general, not necessary for the schedulability of the taskset. This is illustrated in the following example.

Example 6. Consider a taskset composed by two self-suspending tasks, τ_1 having $C_1 = 1, S_1 = 2, T_1 = D_1 = 4$, and τ_2 having the same parameters. We assume that both τ_1 and τ_2 are associated to two H-CBS-SO servers $\mathcal{S}_1$ and $\mathcal{S}_2$

respectively, with $Q_i = C_i + S_i = 3$ for $i = 1, 2$. The period of the servers is the same of the served task. Using *suspension-oblivious* analysis this taskset results non-schedulable under EDF, since

$$\sum_{i=1}^{2} \frac{C_i + S_i}{T_i} = \frac{3}{4} + \frac{3}{4} > 1. \quad (3)$$

However it is easy to see, by checking all possible schedules over the hyper-period of 4, that this taskset is EDF schedulable. This is because the self-suspension times of a given job can be used to execute other pending workload. □

Notice that multiple servers can be in the Self-Suspended state (i.e., enqueued in the SS-QUEUE) at the same time-instant, as illustrated in the following example. This justifies the presence of the queue.

Example 7. Consider a taskset composed by three self-suspending tasks, τ_1 having $C_1 = S_1 = 1$, $T_1 = D_1 = 4$, τ_2 having $C_2 = S_2 = 1$, $T_2 = D_2 = 8$ and τ_3 having $C_3 = S_3 = 1$, $T_3 = D_3 = 10$ executing within H-CBS-SO servers $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, respectively, with $Q_1 = 2$, $P_1 = 4$, $Q_2 = 2$, $P_2 = 8$ and $Q_3 = 2$, $P_3 = 10$. Suppose that at time $t = 0$ server $\mathcal{S}_1$ starts and immediately self-suspends (i.e., it is inserted in the SS-QUEUE) and that, subsequently, server $\mathcal{S}_2$ starts and self-suspends. In this situation $\mathcal{S}_1$ is the head of the queue (i.e., $\mathcal{S}_{SS} = \mathcal{S}_1$), having earlier deadline than $\mathcal{S}_2$. Suppose then that server $\mathcal{S}_3$ starts executing for C_3 time-units; according to Rule SO-4, $q_{SS} = q_1$ is decremented until its exhaustion. This behavior reflects Proposition 2: with H-CBS with busy execution, server $\mathcal{S}_1$ would have continued to (busy) execute disallowing $\mathcal{S}_2$ and $\mathcal{S}_3$ to execute; this also could explain the ordering by increasing deadlines of the SS-QUEUE. □

The H-CBS-SO is able to guarantee temporal isolation for SS-tasks, similarly to the isolation property of the H-CBS reported in Proposition 1. This is expressed by the following proposition:

Proposition 4 (Isolation property of the H-CBS-SO)**.** *Given a set of n SS-tasks, each one running upon a dedicated H-CBS-SO server as in Theorem 3, such that condition (1) holds, if a task exceeds its WCET C or/and its maximum self-suspension time S then the schedulability of the other SS-tasks will not be affected.*

Proof. Since H-CBS-SO extends H-CBS by adding the Self-Suspended state and rules describing transition to and from this state, this similarly follows from Proposition 2 and Proposition 3, after recalling Proposition 1. □

Idle-time reclaiming. The H-CBS-SO algorithm, contrary to the H-CBS algorithm with busy execution, is able to guarantee, together with the schedulability of the taskset (Theorem 3), a higher lower-bound on the idle-time (i.e., time-instant where no server has pending workload).

Consider a taskset composed by n SS-tasks each one executing over a dedicated H-CBS-SO server with $Q_i = C_i + S_i$ and $P_i = T_i$ such that condition (1) holds. We denote by $H := \text{lcm}\{P_1, \ldots, P_n\}$ the hyper-period of the servers.

Suppose that all the servers are released simultaneously at time $t = 0$. At time $t = H$ the length of the idle-time intervals in $[0, H)$ is lower-bounded by the following quantity:

$$\mathcal{I}_{SS} \geq H \left(1 - \sum_{i=1}^{n} \frac{C_i}{P_i}\right) \quad (4)$$

Since during self-suspension times the SS-task does not require to execute, at time $t = H$ the workload executed by the processor is at most $\mathcal{C} = \sum_{i=1}^{n} C_i \frac{H}{T_i}$. Then the idle-time on the interval $[0, H)$ is at least the difference between the elapsed time H and the time $\mathcal{C}$ in which the processor has executed.

Notice that, with H-CBS with busy execution, since self-suspensions are replaced by busy executions, then the idle-time in $[0, H]$ is lower-bounded by

$$\mathcal{I}_{BE} \geq H \left(1 - \sum_{i=1}^{n} \frac{C_i + S_i}{P_i}\right) \leq \mathcal{I}_{SS}. \quad (5)$$

We remark that, with H-CBS without busy execution, even if we can guarantee the same lower-bound $\mathcal{I}_{SS}$ for idle-time, it is not possible to guarantee the schedulability of SS-tasks as shown in Example 1.

The idle-time guaranteed by the H-CBS-SO can be exploited by integrating such a reservation algorithm with reclaiming mechanisms like IRIS [18] or H-GRUB [3]. Both these algorithms have been designed to work with H-CBS and are able to reclaim the idle-time guaranteeing that the spare bandwidth is fairly distributed among the needing servers. IRIS is based on an update of the server parameters when an idle-time occurs while H-GRUB provides a budget accounting rule that takes into account the spare bandwidth. The idle-time could be also exploited for energy-management purposes through the utilization of power-aware scheduling algorithms able to use such a wasted intervals of time to reduce the energy consumption applying techniques of Dynamic Frequency and Voltage Scaling (DVFS) and Device Power Management (DPM) like those proposed by Aydin et. al. [4] and Marinoni et. al. [17].

3.2 Generalization for hierarchical scheduling

The *hierarchical* or *component-based* design has been widely accepted as a methodology to enable modularity and simplify the analysis of large and complex systems (e.g., [24,25]). In this subsection, we extend the H-CBS-SO algorithm to the case where multiple tasks can be run upon the same server, after extending the system model from Section 2.

We consider a two-level hierarchical system. The system is composed by n H-CBS-SO servers $\{(Q_i, P_i)\}$, with server $\mathcal{S}_i := (Q_i, P_i)$ serving a set Γ_i of implicit-deadline SS-tasks. The task model is as in Section 2; we adopt the notation $\tau := (C_\tau, S_\tau, T_\tau)$ to denote a SS-task. The *global scheduler* is based on the H-CBS-SO algorithm. Each subsystem uses a *local scheduler* to select the running task; we consider EDF as local scheduling policy.

We define the H-CBS-SO rules as in Definition 2 but we replace Rules SO-1 and SO-2 with, respectively:

- **Rule SO-1-hier**. If there is no workload to execute on the server $\mathcal{S}_i$ *and* at least one task in Γ_i is self-suspended, then $\mathcal{S}_i$ becomes Self-Suspended and is inserted in the SS-QUEUE (if not already in the SS-QUEUE).

- **Rule SO-2-hier**. If there is workload to execute on the server $\mathcal{S}_i$ *or* there is no task in Γ_i which is self-suspended, then $\mathcal{S}_i$ becomes Ready and is removed from the SS-QUEUE (if in the SS-QUEUE).

With this modifications, it is possible to follow the proof of Proposition 2 thus concluding that self-suspensions *of the*

server are accounted with H-CBS-SO as busy executions are accounted with H-CBS with busy executions. As a direct consequence of this observation, the Isolation Property (or *global schedulability analysis*) of Proposition 4 extends to the hierarchical context.

The local (suspension-oblivious) schedulability analysis of a H-CBS-SO server can be performed using the test proposed in [25]. According to this test, the taskset Γ_i is schedulable by EDF on the server $\mathcal{S}_i$ if

$$\forall t > 0 \qquad \mathrm{dbf}(\Gamma_i, t) \leq \mathrm{sbf}(\mathcal{S}_i, t), \tag{6}$$

where

$$\mathrm{dbf}(\Gamma_i, t) := \sum_{\tau \in \Gamma_i} \left\lfloor \frac{t}{T_\tau} \right\rfloor \cdot (C_\tau + S_\tau) \tag{7}$$

is the *demand bound function* of the taskset Γ_i (i.e., the maximum computational demand of Γ_i in any interval of length $t > 0$) and $\mathrm{sbf}(\mathcal{S}_i, t)$ is the *supply bound function* of the server $\mathcal{S}_i$ (i.e., the minimum amount of service time provided by the server in any interval of length $t > 0$). An expression for $\mathrm{sbf}(\mathcal{S}_i, t)$ (the same of the H-CBS) can be found in [11]. Techniques to solve (6) can be found in [5].

4. LINUX IMPLEMENTATION

This section describes how the H-CBS-SO has been implemented in the Linux kernel and shows the overhead introduced by this extension.

The Linux kernel 3.14 introduces a new scheduling class called SCHED_DEADLINE that implements the H-CBS algorithm. This infrastructure can be extended to implement the novel H-CBS-SO algorithm and verify its performance. In particular, the implementation has been made by modifying version 3.19 of the vanilla Linux kernel.

SCHED_DEADLINE is characterized by a strong relationship between tasks and servers: each server must have one and only one associated task. This leads to the lack of a clear distinction in the data structures. For this reason, in SCHED_DEADLINE, the terms "task" and "server" represent the same entity and can be used be used both interchangeably.

4.1 Modifications

This section presents the data structures and the functions that have been added or modified to implement the H-CBS-SO algorithm.

4.1.1 Data Structures

The SS-QUEUE (see Definition 2) is implemented through a red-black tree, which has a complexity of $O(\log(n))$ for insertion, removal and search operations, where n represents the number of elements in the tree. The SS-QUEUE is arranged by absolute deadlines, thus the server S_{SS} (*i.e.,* the one having earlier deadline) is represented by the leftmost element of the tree. Because the most frequent operations (e.g., removal and budget update) are performed on S_{SS}, a pointer to the leftmost element is added to speed up these common operations, reducing the complexity to $O(1)$.

An additional flag must be added for each task in order to keep trace of its self-suspension status, because the system must determine if that task was self-suspended when a budget replenishment is performed on it. This is due to the fact that it needs to be decided if the task must be reinserted in the SS-QUEUE following the SO-3 rule or can be set as ready.

4.1.2 Functions

A performing solution to catch the transitions to and from the self-suspension state can be obtained intercepting the insertion or removal of the tasks in the SCHED_DEADLINE runqueue. The reasons leading to the removal of a task from the runqueue are (i) server budget exhaustion; (ii) task termination; (iii) scheduling class modification and (iv) self-suspension, hence, the self-suspension is detected when a task is removed from the runqueue and this removal is not caused by one of the first three cases.

Below is presented how the H-CBS-SO rules are implemented in SCHED_DEADLINE.

SO-1. When a SCHED_DEADLINE task leaves the runqueue, it is checked if the cause of this transition is a self-suspension, and if this is the case, it is inserted in the SS-QUEUE.

SO-2. If a self-suspended SCHED_DEADLINE task enters the runqueue, then it can switch to the ready state, thus be removed from the SS-QUEUE.

SO-3. When the S_{SS} exhausts its budget, then its self-suspension flag is set, it becomes suspended and a timer is activated for its replenishment. If the self-suspension flag is active when the budget is replenished, then the task is inserted in the SS-QUEUE, otherwise it is inserted in the runqueue.

SO-4. SCHED_DEADLINE tasks budgets are periodically updated. In the new implementation, the budget of the S_{SS} is also updated when needed by the server rule. To minimize the overhead, no budget updates are performed on the SS-QUEUE when there are no ready SCHED_DEADLINE tasks. This idle time is measured with a timer and is used to bring back the system to a consistent state by scaling it from the budgets of the self-suspended tasks.

4.2 Performance Evaluation

Some tests are performed to evaluate the overhead of the newly introduced SCHED_DEADLINE functions by the H-CBS-SO implementation. The tests are performed running several periodic tasks and measuring the execution times of the two implementations.

4.2.1 Setup

The tests are performed on a machine equipped with an Intel Core 2 Duo running at 3 GHz. Measurements are obtained through the Ftrace tool, that is the internal Linux kernel tracer and is used as function profiler in this experiments.

4.2.2 Tasksets

The tasksets used for the performance evaluation have a number of tasks equal to 2^k, where $k = \{1, \ldots, 10\}$. The total utilization factor of each taskset is chosen equal to $U = 0.8$. The utilization factor U_i of each task is randomly generated such that the minimum value is $U_{lb} = \frac{U}{n+1}$ and $\sum_i U_i = U$. The utilization factor is defined according the *suspension-oblivious* analysis, *i.e.,* it accounts for self-suspensions times as execution times. The period of each job is chosen as a random value between 0.1 ms and 10 ms. Each task releases 10000 jobs that (i) busy waits to simulate execution; (ii) self-suspends; (iii) busy waits again and (iv) waits for next activation.

The last step is implemented with the use of the *sched_yield()* system call, which in SCHED_DEADLINE zeros the budget and triggers the suspension until the next activation.

For each job, the busy wait to simulate the task execution is equal to $\frac{1}{6}$ of the total execution time, while the remaining $\frac{2}{3}$ is related to the self-suspension.

4.2.3 Execution Times

The following figures compare the performance of the SCHED_DEADLINE scheduling class before and after the H-CBS-SO extension, showing the execution times of the modified kernel functions. The overhead before the extension is indicated with the label H-CBS, since it is the name of the native server algorithm in SCHED_DEADLINE.

Each plot displays the execution times values as a function of the number of tasks in the task set of each modified function. The results are expressed in microseconds in a logarithmic scale. The circles and the squares represent the mean values of the two implementation, while the vertical bars delimit the minimum and the maximum values.

The main observation is that the additional overhead required for implementing the H-CBS-SO has been observed to be considerably small in all the tested scenarios.

Figure 6 shows the overhead introduced in the *update_curr_dl()* function, which updates the budgets of the running task and the S_{SS} (see rule SO-4). In addition to the budget update, if S_{SS} exhausts its budget, it is removed from the SS-QUEUE (see rule SO-3). This removal operation is the reason of the execution time growth of this function when the number of tasks increases.

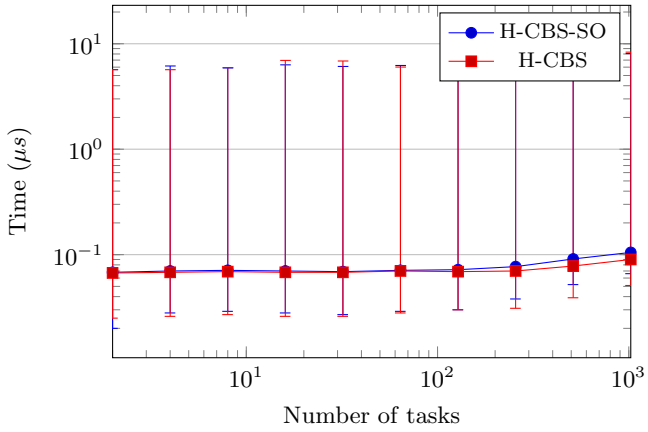

Figure 6: *update_curr_dl()* execution times.

Figure 7 shows the overhead introduced in the *enqueue_task_dl()* function, which puts a task into the SCHED_DEADLINE runqueue, and so, may resume a self-suspended task from the SS-QUEUE. Also in this case, the SS-QUEUE removal operation is the cause of the general larger execution time of this function. If the function is executed for a task which was in Suspended state due to the SO-3 rule, then it simply updates a flag and returns. This flag is required in order to decide if, after the budget replenishment, the task must turn back to the Ready state or follow the SO-3 rule, and so, be pushed in the SS-QUEUE. The flag update is a simple and quick operation and the probability of performing this operation increases with the number of self-suspending tasks. This is the reason of the convergence of the two execution time of the function before and after the H-CBS-SO extension.

The Figure 8 shows the overhead introduced in the *dequeue_task_dl()* function, which removes a task from the SCHED_DEADLINE runqueue, and so, may be caused by a

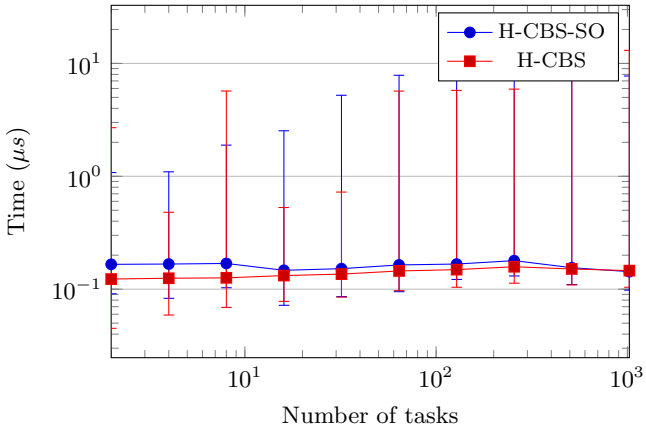

Figure 7: *enqueue_task_dl()* execution times.

self-suspension. This operation justifies the increased overhead shown in the picture, but the execution time grows logarithmically with the number of tasks, because of the rb-tree insert operation. The number of tasks used for this experiment is too small to show this behavior.

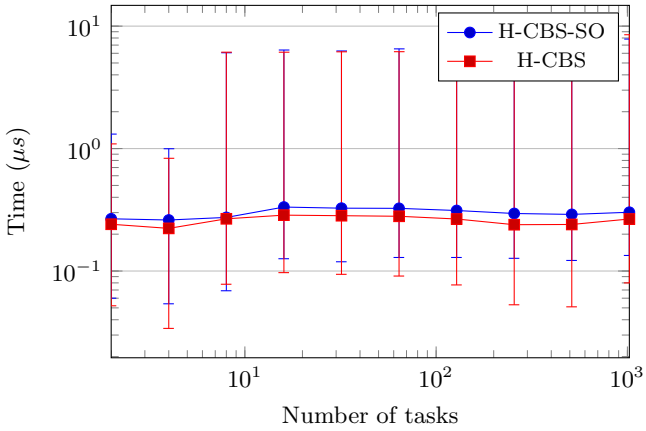

Figure 8: *dequeue_task_dl()* execution times.

The Figure 9 shows the overhead introduced in the *dl_task_timer()* function, which replenishes the budget of a suspended task. If the suspended task was self-suspended, then it must be pushed back in the SS-QUEUE, otherwise, it is pushed in the runqueue. Performance are apparently improved because, by splitting SS-QUEUE and runqueue, the nodes of the two trees are less, resulting in a lower access time to their elements.

Overall, the figures show that the computational cost of the new functions is comparable to the original SCHED_DEADLINE implementation. As a result, the modifications introduced to implement the H-CBS-SO algorithm present a low impact on the system load.

5. CONCLUSIONS

Modern platforms are composed by multiple heterogeneous computation units and are managed with software infrastructures providing temporal isolation among concurrent applications. To guarantee timing constraints of real-time applications executing in this kind of environments, new task models and scheduling algorithms must be introduced.

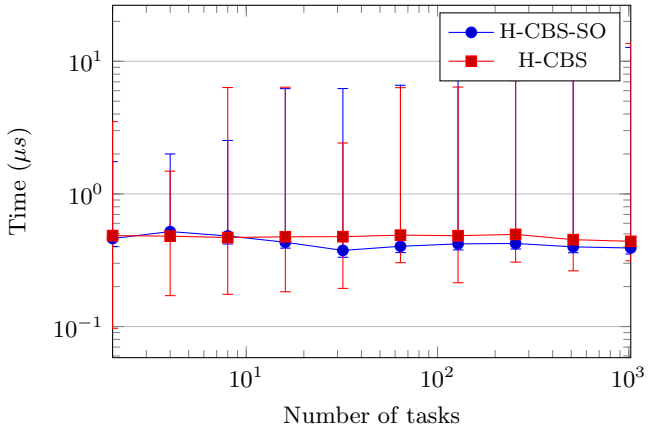

Figure 9: *dl_task_timer()* execution times.

In this paper is presented the novel H-CBS-SO scheduling algorithm that provides resource reservation for real-time self-suspending tasks. It has been shown that it is able to guarantee a *suspension-oblivious* schedulability analysis for self-suspending tasks running upon H-CBS-SO servers, while avoiding to waste processor cycles, as happens when using busy executions in place of self-suspensions.

The proposed algorithm has been implemented in the Linux kernel, as an extension of the SCHED_DEADLINE scheduling class, today part of the mainline of Linux. The implementation has been described and evaluated in terms of run-time overhead.

As future work, we want to address the issue of suspension-aware analysis of self-suspending tasks scheduled using H-CBS based reservation servers. We also intend to integrate existing idle-time reclaiming mechanisms with the H-CBS-SO algorithm. In addition, synchronization protocols for resource reservation scheduling will be addressed to cope with self-suspensions related with such protocols.

6. REFERENCES

[1] Y. Abdeddaïm and D. Masson. The scheduling problem of self-suspending periodic real-time tasks. In *Proceedings of 20th International Conference on Real-Time and Network Systems*, Pont-à-Mousson, France, Nov 2012.

[2] 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 2-4 1998.

[3] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari. Resource reservations for general purpose applications. *IEEE Transactions on Industrial Informatics*, 5(1):12–21, February 2009.

[4] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(4):584–600, May 2004.

[5] S. Baruah. Resource sharing in EDF-scheduled systems: a closer look. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, December 5-8, 2006.

[6] M. Bertogna, N. Fisher, and S. Baruah. Resource-sharing servers for open environments. *IEEE Transactions on Industrial Informatics*, 5(3):202–219, August 2009.

[7] A. Biondi, A. Melani, and M. Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proc. of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, Pisa, Italy, 18-20 June, 2014.

[8] A. Biondi, A. Melani, M. Bertogna, and G. Buttazzo. Optimal design for reservation servers under shared resources. In *Proc. of the 26th Euromicro Conference on Real-Time Systems (ECRTS 14)*, Madrid, Spain, 9-11 July, 2014.

[9] B. Brandenburg and J. Anderson. The omlp family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, pages 1–66, 2012.

[10] J. Lelli, D. Faggioli, T. Cucinotta, and S. Superiore. An efficient and scalable implementation of global edf in linux. In *Proc. of the 7th Workshop on Operating Systems Platforms for Embedded Real-Time applications*, Porto, Portugal, 2011.

[11] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, April 2005.

[12] C. Liu and J. H. Anderson. An O(m) analysis technique for supporting real-time self-suspending task systems. In *In Proc. of the 33rd Real-Time Systems Symposium 2012*.

[13] C. Liu and J. H. Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *In Proc. of the 25th EuroMicro Conference on Real-Time Systems (ECRTS 2013)*.

[14] C. Liu and J. H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *In Proc. of the 30th Real-Time Systems Symposium (RTSS 2009)*.

[15] J. Liu, editor. *Real-time systems*. Prentice Hall, 2000.

[16] R. Mall, editor. *Real-time systems: theory and practice*. Pearson Education, 2008.

[17] M. Marinoni and G. Buttazzo. Elastic dvs management in processors with discrete voltage/frequency modes. *IEEE Transactions on Industrial Informatics*, 3(1):51–62, Feb 2007.

[18] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 25-28 2004.

[19] G. Nelissen, J. Fonseca, G. Raravi, and V. Nélis. Timing analysis of fixed priority self-suspending sporadic tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, Lund, Sweden, July 2015.

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

[21] P. Richard. On the complexity of scheduling real-time tasks with self-suspension on one processor. In *Proc. of the 15th IEEE Int. Euromicro Conferecnce on Real-Time Systems (ECRTS 2003)*.

[22] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proc. of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*.

[23] F. Ridouard, P. Richard, and F. Cottet. Some results on scheduling tasks with self-suspensions. In *Journal of Embedded Computing, 2006*.

[24] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, December 3-5, 2003.

[25] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 57–67, Lisbon, Portugal, December 5-8, 2004.