Sporadic Server Revisited [*]

Dario Faggioli, Marko Bertogna, Fabio Checconi
ReTiS Lab, Scuola Superiore Sant'Anna, CEIICP
via G. Moruzzi 1, 56124 Pisa (Italy)
{d.faggioli, m.bertogna, f.checconi}@sssup.it

ABSTRACT

The Sporadic Server (SS) overcomes the major limitations of other Resource Reservation Fixed Priority based techniques, but it also presents some drawbacks, mainly related to an increased scheduling overhead and a not so efficient behavior during overrun situations.

In this paper we introduce and prove the effectiveness of an improved SS with reduced overhead and fairer handling of server overrun situations. We also show how this can be efficiently exploited to provide temporal isolation in a multiprocessor platform, adapting already existing schedulability tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management

Keywords

Scheduling, Fixed Priority, Multiprocessor, Sporadic Server, Overruns

1. INTRODUCTION

Typical real-time applications range from safety critical controls, like flight and defense systems, to multimedia, networking and streaming applications, where the Quality of Service (QoS) perceived by the user is the most important aspect. Due to the rapid development of microprocessor technology, multiprocessor platforms are nowadays a viable solution even in this field of application, offering a significantly higher computing power at a limited cost.

In this work, we analyze the problem of scheduling a workload composed by hard, soft and non real-time tasks on a multiprocessor platform. In particular, we will improve and adapt to multiprocessor systems a previously proposed technique for the scheduling of aperiodic workloads on a uniprocessor, fixed priority, system: the Sporadic Server [24].

[*]This work has been partially supported by the IRMOS (FP7/2008/ICT/214777) and ACTORS (FP7/2008/ICT/216586) projects.

This is because most of the existing (real-time or not) Operating Systems are based on fixed – rather than dynamic – priority scheduling, since it is often easier to implement and it entails a smaller scheduling overhead. Moreover, application developers still feel more comfortable about using priorities, instead of relying on dynamic schedulers. Finally, coming to multiprocessors, the gap in the maximum achievable utilization between fixed and dynamic priority approaches is reduced.

The Sporadic Server, proposed by Sprunt *et al.* in [24], allows implementing both resource reservation and aperiodic request handling in a fixed priority real-time system. However, both theoretical and practical issues concerning resource reservation mechanisms for globally scheduled fixed priority systems received a significantly smaller attention.

Organization of the paper.

The remainder of the paper is organized as follows: in Sec. 2, we introduce background concepts. In Sec. 3 we compare against existing solutions. In Sec. 4 we deal with Sporadic Server on multiprocessor systems. In Sec. 5 and 6 we give all the details about our proposals. In Sec. 7, we show our simulation results before drawing our conclusions in Sec. 8.

2. SYSTEM MODEL

We consider a shared memory multiprocessor real-time system, with m identical processors $P_1, \dots, P_m$ with unit capacity. Each activity is referred to as a *task* τ. Each periodic or sporadic task τ_i consists of a stream of jobs, $J_{i,j}$, each one characterized by an arrival time $a_{i,j}$, a computation time $c_{i,j}$ and an absolute deadline $d_{i,j}$. Moreover, each task τ_i is characterized by a triplet (C_i, D_i, T_i), where $C_i = max_j\{c_{i,j}\}$ is the worst case execution time (WCET), $D_i = d_{i,j} - a_{i,j}$ is the relative deadline, and T_i is the period or minimum inter-arrival time ($a_{i,j+1} \geq a_{i,j} + T_i$). The processor utilization factor U_i of τ_i is defined as $U_i = \frac{C_i}{T_i}$. Hard real-time tasks must meet all deadlines, while soft real-time tasks may finish after their deadline, degrading the resulting QoS perceived by the user.

In this paper we focus on fixed priority (FP) preemptive scheduling and we are mainly interested in open real-time systems, where hard, soft and non real-time tasks may coexist and are dynamically activated and terminated at system runtime.

Resource Reservation.

The Resource Reservation framework (RR) has proven to be an effective technique to keep the QoS of soft activities under control, and to achieve bandwidth isolation among different hard, soft and non real-time tasks. When RR is used, one or more tasks can be assigned to a reservation (or server) S, with budget Q and period P, that will provide some scheduling guarantee. Typically, execution for at least Q every P time-units is enforced, with no room for task reciprocal interference. We say that a server is *backlogged* if it has pending jobs to execute.

Therefore, assuming (periodic) hard real-time workload to have been guaranteed off-line, we need a mechanism that provides this isolation to soft tasks, as well as fast response time to aperiodic ones, without jeopardizing the guarantees of hard tasks.

In the field of fixed priority preemptive scheduling, many server based approaches have been proposed.

The Polling Server and the Deferrable Server (PS, DS [22, 25]) are two such algorithm, based on periodic budget refilling. Rather, the Sporadic Server (SS [24]) is slightly different, and it outperforms PS and DS from many points of view [24, 12, 11]. The server budget is replenished one period after the server activation, and only by the amount of capacity that has been consumed in that time interval. In more detail, a Sporadic Server S, with budget Q and period P, works as follows:

1. The server is in *Active* state when it is backlogged and it has a positive remaining budget;

2. The server is in *Idle* state when it is not backlogged or its budget is exhausted;

3. Initially, the server is *Idle* and its budget is Q. When the server becomes *Active* at time t_1, the recharging time is set to $(t_1 + P)$;

4. when the server becomes *Idle* at a time t_2, the recharge amount corresponding to the last recharging time is set to the amount of capacity consumed by S since the last transition, i.e., in $[t_1, t_2)$.

Other fixed priority servers, (e.g., Priority Exchange [25]), are not described here due to space reason. Let us only say that they have either more complex implementations or larger schedulability penalties.

3. RELATED WORK

To the best of our knowledge, there are only few works dealing with reservation mechanisms for multiprocessor environments scheduled with Fixed Priority. One of them is the multiprocessor TBS implementation presented by Baruah and Lipari in [6], which is applicable to every work-conserving algorithm. With this approach, each aperiodic job is scheduled either in background, or with a very low priority, so that is does not interfere with hard real-time tasks. However, this significantly increases the response time of aperiodic activities. Furthermore, it is necessary to know in advance the execution requirements of each aperiodic request.

In [23], Sha *et al.* thoroughly studied the applicability of RM scheduling on multiprocessor distributed systems, using SS for aperiodic activities. However, the main focus was on hardware and network-level real-time support for distributed, or loosely coupled multiprocessor systems, quite different from our perspective.

In [17], Davis and Burns studied the problem of server overruns in resource sharing fixed priority hierarchical systems. A "payback" mechanism is presented that decreases by the amount of the overrun the capacity allocated to the overrunning server in the subsequent period. This is one of the main inspiration for this work, where we conceived a similar mechanism for the Sporadic Server. In [15], Caccamo *et al.* confine an overrun by means of a dedicated server. Buttazzo *et al.* introduced in [14] the elastic task model to describe tasks that may dynamically change their characteristics. A criticality-based EDF scheduling with admission and rejection control is proposed in [13]. However, these works tackle the problem of transient or permanent overload conditions for the task set as a whole. In this paper we will instead adopt a different perspective, dealing with specific server overruns.

Finally, actual implementations of the Sporadic Server can be found in [11, 21, 4, 10]. Among commercial operating systems, support for POSIX `SCHED_SPORADIC` is claimed by QNX [1], RTEMS [2] and, recently, by Xenomai [3].

4. SPORADIC SERVER AND MULTIPROCESSORS

Multiprocessor Scheduling.

For general task system scheduled with FP on a multiprocessor platform, an upper bound on the schedulable utilization is $\left(\frac{m}{2} + \frac{1}{3}\right)$, as shown in [3]. However, no known priority assignment allows a schedulable utilization equal to the above upper bound and no exact schedulability test is known for such systems. Sufficient tests can be found in [2, 1, 8, 5, 7, 9].

Indeed, no particular mechanism is needed to make a Sporadic Server capable of working well in global FP scheduled multiprocessors. However, it needs to be proven that the existing schedulability tests are still valid for systems that include sporadic servers. This is not as immediate as on uniprocessors, since no concept of critical instant exists in multiprocessor schedulability theory. Nevertheless, we will show in this section that at least two previously proposed tests for sporadic task systems are suitable solutions.

Andersson *et al.* provided RM and RM-US schedulability bounds in [2, 1]. These bounds have been later improved by Bertogna *et al.* in [8], proving a schedulable utilization bound of $(m + 1)/3$ for RM-US[1/3], a priority assignment that gives highest priority to tasks with utilization larger than 1/3, scheduling the remaining ones with RM.

A different result for globally scheduled fixed priority systems has been derived in [9], where the following upper bound on the workload of a task τ_i, with slack $\geq s_i$, in a window L is proved:

$$\begin{aligned} W_i(L, s_i) = \min\left(C_i, L + D_i - C_i - s_i - N_i(L, s_i)T_i\right) \\ + N_i(L, s_i)C_i, \end{aligned} \tag{1}$$

with

$$N_i(L, s_i) = \left\lfloor \frac{L + D_i - C_i - s_i}{T_i} \right\rfloor \tag{2}$$

[1]http://www.qnx.com/
[2]http://www.rtems.com/
[3]http://www.xenomai.org/

and

$$s_i = D_i - C_i - \left\lfloor \frac{\sum_{j<i} \min(W_j(D_i, s_j), D_i - C_i + 1)}{m} \right\rfloor. \quad (3)$$

A simple schedulability test can be derived (see [9] for details):

- initially, all slack lower bounds are set to $s_i = 0$;
- the slack lower bound s_k of each task $\tau_k \in \Gamma$ (the task set), is computed using Equation (3); slacks are updated in decreasing priority order;
- if, for a task τ_k, Equation (3) returns a negative value, the test fails;
- otherwise, all tasks have a non negative slack, and the task set is schedulable with Fixed Priority.

Both results in [9] and [8] apply as well to systems in which periodic and sporadic tasks are scheduled together with Sporadic Servers. The proofs in the original papers are derived using upper bounds on the workload produced by each task in an interval of length D_k. We will show that the workload produced by a SS cannot be larger than the workload produced by a sporadic task having WCET and period equal to, respectively, the server budget and period.

THEOREM 1. *The workload of a Sporadic Server S_i in a window L cannot be larger than when S_i is continuously backlogged throughout L.*

PROOF. Suppose, by contradiction, that the largest workload of a SS S_i in a window L is found when the server is not continuously backlogged. Let $[t_1, t_2)$ be the first time interval $\in L$ during which the server is not backlogged. Examining the SS rules, S_i must be *Idle* throughout $[t_1, t_2)$. Therefore, when the server will become *Active*, the next recharging time is set to at least time $t_2 + P_i$. If instead the server is backlogged throughout $[t_1, t_2)$, the server is either *Idle* or *Active* if, respectively, its budget is exhausted or not. In the first case, the budget has been exhausted before t_1, and a recharge of Q_i time-units will be set before time $t_1 + P_i < t_2 + P_i$; therefore, the server will be able to contend for execution earlier than when it is not backlogged, potentially producing a larger workload. In the second case, the server is immediately allowed to contend for execution for its remaining budget. Moreover, since the server is continuously backlogged in $[t_1, t_2)$, the last time it became *Active* was earlier than t_1, and all recharging times are again set to a time $< t_1 + P_i$. Therefore, also in this case, the server will be able to contend for the execution of Q_i time units within time $t_1 + P_i < t_2 + P_i$, potentially producing a larger workload than in the non-backlogged case, and leading to a contradiction. Repeating the same argument for any other interval in which the server is not backlogged, the theorem follows. □

Thus, an SS produces its largest possible workload in L when it executes for Q time-units before being fully recharged, and then it executes for Q time-units at the beginning of each period — i.e., it is continuously backlogged in L. Hence, since such situation is identical to the worst-case scenario considered above for a sporadic task having period P and WCET Q, the schedulability tests in [9] and [8] are applicable as well to sporadic servers, treating each SS S_k as a sporadic task τ_k having $D_k = T_k = P_k$ and $C_k = Q_k$.

5. SPORADIC SERVER AND BUDGET OVERRUNS

Enforcing the execution of serviced tasks for at most the server budget Q is a key aspect in Resource Reservation. There are, however, situations where a *budget overrun* is either impossible to avoid, or even desirable.

Limited Timer Resolution.

In a modern OS, the account of the execution time of a task is either event-based — at each scheduling event like task activation, deactivation or termination — or it is time-based — periodically at each *system tick*.

Therefore, it could happen that the OS is not able to stop a server execution at the exact instant of budget exhaustion. It is easy to see that the worst possible error in budget accounting is equal to the tick period P_{tick}. In fact, suppose the budget of an executing server S_i is found barely positive at one particular tick. Since the budget will be checked again only after P_{tick} time units, the server will experience an overrun of exactly that amount of time. Furthermore, since the tick delivery may be subject to a jitter Δ, the bound may increase to $P_{tick} + \Delta$. Using a timer for budget enforcing may alleviate but not solve the issue, while introducing more overhead.

Widened OS Latency.

Wide kernel latencies can be modeled as temporary reductions of the OS tick (and also timers) resolution. A relevant example is given by virtualized systems, where a (guest) OS is executed as a common process of another (host) OS. In fact, a tick of the guest fired at time t_G, could be not serviced before $(t_G + t_H)$, if at t_G the guest VM is not a running process on the host, with t_H dynamically varying and possibly being significantly larger than t_G itself.

Exhaustion During Critical Sections.

The plain Sporadic Server policy does not deal with the problem of budget exhaustion while holding one or more locks on some shared data [16, 20, 18]. Two viable solutions for this are:

- preventing a task to enter a critical section if there is not enough budget to complete it;
- allowing a server to overrun its budget while inside a critical section.

The first solution requires to know in advance the computational length of each Critical Section, and it is therefore more suited for hard real-time environments[4]. Thus, the second solution is more interesting to us. However, allowing a plain Sporadic Server to overrun, or trying to apply a simple payback mechanism, will produce some scheduling oddities, as shown in the next section.

5.1 Sporadic Server with Payback

If avoiding overrun is impossible or unwanted, we can at least try to "restore some fairness" in the schedule, e.g., by making the overrunning server payback in its subsequent instance(s), as in [18]. When using a PS or a DS (or even a CBS, in dynamic priority systems), it is sufficient to decrease

[4]We are preliminary studying how to apply this solution to dynamic soft real-time systems in which critical section lengths are not known a priori [19]

the budget of the next server instance by the amount of the overrun.

Unfortunately, applying this technique to a Sporadic Server is not equally simple, as shown in Fig. 1. In fact, subtracting the overrun amount, O_1, from the budget of the server at its next replenishment will produce the following:

- if we replenished the budget by $(Q + O_1)$, as it should be, then it would be recharged to Q, without any payback;
- if we replenished the budget only by at most Q, then a lower server budget will propagate toward next executions, producing a *permanent* budget leakage, as depicted in Fig. 1(b).

This happens because, in the SS, replenishment times are *not* periodic and especially because replenishment amounts are *not* equal to the initial server budget, as it is in almost all other algorithms.

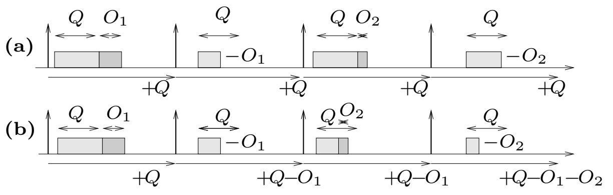

Figure 1: "Naive" payback for DS (a) and SS (b). Up arrows in the timeline are task activations, i.e., instants when the server become backlogged. Overruns are in dark grey; replenishment event, with corresponding replenishment amount, are the arrows below the timelines.

Therefore, what we propose is to enhance a Sporadic Server by the following:

- the accumulated overrun is saved in an *overrun pool* O_{pool}, i.e., when an overrun O occurs, $O_{pool} += O$;
- each recharge amount is limited by the initial server budget Q, i.e., if the server executed for $(Q+O)$ units, recharge amount is set to Q;
- while replenishing, at time t:
  1. the replenishment amount is, as usual, added to the current server budget q;
  2. if $O_{pool} > 0$, both the O_{pool} and q are decreased by $\min\{q, O_{pool}\}$ [5];
  3. a replenishment of $\min\{q, O_{pool}\}$ units is planned to happen at $t + P$.

In other words, *in case of an overrun O, the proposed mechanism works as if a budget amount of O* (or as close as possible to O) *is instantly consumed by the server at the very next replenishment time.* This way, the overrunning server pays his debt as soon as possible, no new recharge overhead is added, and the budget leakage is no longer in place. In Fig. 2 the same situation depicted in Fig. 1(b) is shown, this time with our payback mechanism.

Let us now, consider a SS with budget Q and period P, in a system where the largest possible overrun is to Δ. A classic

[5] this means, in case $O \geq Q$, one or more server instances may be skipped.

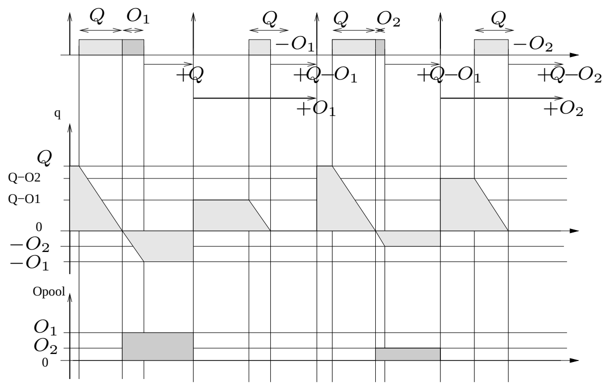

Figure 2: The same schedule of Fig. 1(b), this time with our payback in place; additional replenishments introduced by us are thicker arrows. The current budget (q) and the overrun pool (O_{pool}) are shown as well for relevant time instants.

SS without any payback mechanism may experience subsequent overruns, executing for $(Q + \Delta)$ in each period, i.e., for instance, after n periods the cumulative overrun may be $n \cdot \Delta$. On the other hand, thanks to our payback mechanism the following theorem holds.

THEOREM 2. *In any interval L, the cumulative execution of SS cannot exceed by more than Δ time-units the one of a non-overrunning server with identical budget and period.*

PROOF. For Theorem 1, the largest workload in L can be found when the SS is continuously backlogged throughout L. If an overrun of Δ occurs, the subsequent instance of SS is subject to a budget reduction of Δ. Therefore, even if another overrun ($\leq \Delta$) occurs, the second instance cannot possibly consume more than $(Q - \Delta + \Delta) = Q$ time units. The same is true for each overrunning instance, and the overall execution of a SS with payback mechanism cannot, therefore, exceed the ideal behavior by more than Δ time-units. □

Thus, in case we want to consider the possibility of overruns in the schedulability test, we can significantly reduce the scheduling penalty associated to them. In fact, if no payback mechanism is present, it can be shown that the largest workload of an overrunning server S_i is produced in the situation of Figure 3(a). Thus, we can, again, use the test from [9], inflating the execution time of each and every instance of the tasks by Δ; this means always replacing C_i with $(Q_i + \Delta)$, and we will denote such test as SS_Original. When instead the payback mechanism is introduced, a continuously backlogged SS may execute for up to $(Q_i + \Delta_i)$ only for one instance. So, an upper bound $W'_i(L)$ on the execution allowed in L can again be found with usual techniques. It is possible to show that a worst-case condition is

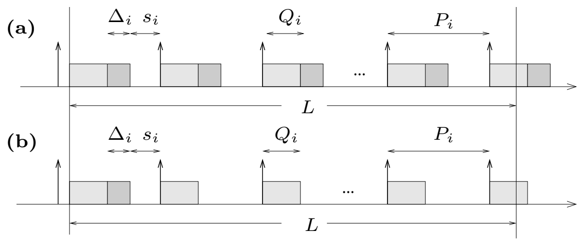

Figure 3: Densest possible execution of a SS (a) without and (b) with payback mechanism.

given by the situation of Fig. 3(b) and [6].

$$W'_i(L,s_i) = \min\left(Q_i+\Delta_i, L-Q_i-s_i-\left\lfloor\frac{L-Q_i-s_i}{P_i}\right\rfloor P_i\right) + \left(\left\lfloor\frac{L-Q_i-s_i}{P_i}\right\rfloor+1\right)Q_i. \quad (4)$$

A lower bound on the slack of a server S_k with maximum overrun Δ_k is then given by Equation (3) replacing C_k with $(Q_k + \Delta_k)$, and using $W'_i(L, s_i)$ instead of $W_i(L, s_i)$ — and $W'_i(L, s_i) \le W_i(L, s_i)$ — for each server with payback mechanism. This test will be denoted as SS_Payback.

6. SPORADIC SERVER AND USELESS REPLENISHMENTS

Each time a SS sets a replenishment event, an OS timer is involved, and this entails some overhead. In fact, if the system had some activity ongoing, it must be interrupted to run the timer handling routine; if was idle, and thus likely in a power-saving state, this has to be revoked for a power consuming one. However, if the server is not backlogged, no task is able to exploit the replenishment that is being performed, and such event is thus only system and power overhead, as shown in Fig. 4

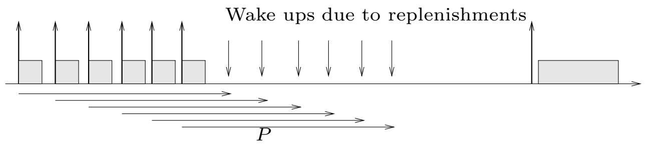

Figure 4: Spurious wake-ups due to "suspending after a burst" task behavior.

Thus, what we propose is to perform only one single, cumulative, budget replenishment when the server becomes backlogged again. Formally speaking:

- when the server *stops* being backlogged, each timer associated to a replenishment event is stopped;
- when the server become *backlogged* again, (i) each pending recharge associated to a past replenishment event is immediately issued, and (ii) each timer associated to a future replenishment event is rearmed.

[6]a tighter bound of the workload in L can be obtained adding a limitation to L in the minimum of (4). However, this limitation is not needed, since the considered workload in $L = D_k$ is already limited by a lower value $(D_k - C_k + 1)$ in the theorem.

	$P_{tick} = 1000$	$P_{tick} = 4000$	$P_{tick} = 10000$
$m = 2$	44%	45%	47%
$m = 4$	48%	49%	51%
$m = 8$	53%	54%	55%

Figure 5: Schedulability loss of SS_Original with respect to SS_Payback.

7. SIMULATIONS

Simulation based study has been conducted to demonstrate the effectiveness of the proposed approach. We considered a platform composed of $m = 2, 4, 8$ processors upon which a set of sporadic servers are scheduled using RM. A randomized distribution of sporadic servers SS_i with budget Q_i and period P_i has been generated in the following way. Initially, a set of $m + 1$ servers has been created, with

- periods P_i uniformly distributed in $[10000, 1000000]$;
- utilization U_i from an exponential distribution with mean $\lambda = 0.25$, and budgets accordingly computed as $Q_i = U_i P_i$.

If test SS_Payback cannot prove the schedulability of the derived set, then the set is discarded, and a new set of $m + 1$ servers is created. Otherwise, the set is considered for evaluation. Then, another set (with $m + 2$ elements) is created, generating a new server, and adding it to the previous set. This until 10^6 sets have been generated.

We considered two different scenarios. In the first one, we compared the number of schedulable sets with and without payback mechanism, using, respectively, the tests SS_Payback and SS_Original. The maximum overrun has been set equal to the OS tick, which can be 10000, 4000 or 1000 time units. What we measured are significant losses without the payback mechanism. With $m = 2$ processors and $P_{tick} = 1000$, 44% of the generated task sets are schedulable only with SS_Payback and not with SS_Original. Increasing P_{tick} to 4000 and 10000, the schedulability loss of SS_Original increases to, respectively, 45% and 47%, since the overruns are higher. Increasing the number of processors, the situation is even worse. Fixing the tick period to 1000, the schedulability loss of SS_Original with 4 and 8 processors increases to, respectively, 48% and 53%. All results are summarized in Figure 5.

In the second scenario, we wanted to find out how many potentially useless replenishing events the system may be subject to. To do that, we considered each server SS_i to be activated every period P_i with a certain probability. The "skip probability" is the probability the server has to skip the next activation. Each time the server is activated, it executes K consecutive sub-instances, each one for Q_i/K time-units, where K is chosen in $\{1, 4, 8, 16\}$. We measured the total number of useless wake-ups the system experiences for a total simulation of 1000 time-units.

Figure 6 shows the number of such spurious wakeup the original SS algorithm causes to the system. If our improved algorithm is used none of these events happen. A marked bursty behavior and a skip probability up to 50%, may cause a large number of spurious wake-ups. It is easy to understand why the number of useless wake-ups decreases for skip probabilities larger than 50%. In fact, since fewer jobs are executed, less replenishing events are posted as well.

8. CONCLUSIONS AND FUTURE WORKS

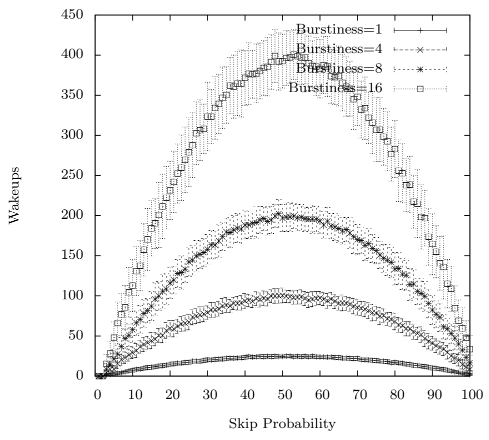

Figure 6: Number of spurious wake-ups over job skip probability, at each different maximum sub-instance number.

This paper considered an improved implementation of the Sporadic Server for the resource reservation of fixed priority scheduled real-time systems. In particular, it improved over the state of the art by

- providing schedulability tests that can be used when scheduling a SS in a multiprocessor system using a global fixed priority scheduler; and
- proposing simple modifications to the classic SS, for a more robust and efficient implementation on a real Operating Systems.

The combination of the proposed mechanisms has multiple positive outcomes: from a fairer behavior in presence of overruns, to an improved schedulability and predictability of the system, to a reduced replenishment overhead. The effectiveness of the proposed solutions has been showed with simple examples and simulations.

As a future work, we are planning to implement the presented SS mechanisms on Linux, in order to run an experimental evaluation on a real physical system. Moreover, we are trying to integrate our server implementation with reclaiming mechanism for the management of the unused bandwidth, as well as for the access of shared resources.

9. REFERENCES

[1] B. Andersson. *Static-Priority Scheduling on Multiprocessors*. PhD thesis, Department of Computer Engineering, Chalmers University, 2003.

[2] B. Andersson, S. Baruah, and J. Jansson. Static-priority scheduling on multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 193–202. IEEE Computer Society Press, December 2001.

[3] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. Technical Report 01-2, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2001.

[4] T. Baker, A. Wang, and M. Stanovich. Fitting linux device drivers into an analyzable scheduling framework. In *Proceedings of the* 4^{th} *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2007.

[5] T. P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems: The International Journal of Time-Critical Computing*, 32(1–2):49–71, 2006.

[6] S. Baruah and G. Lipari. A multiprocessor implementation of the Total Bandwidth Server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.

[7] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona (USA), 2007.

[8] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, Pisa, Italy, December 2005. IEEE Computer Society Press.

[9] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[10] R. J. Bril and P. J. L. Cuijpers. Towards exploiting the preservation strategy of sporadic servers. In *Work in Progress (WiP) Session of the 20th Euromicro Conference on Real-Time Sustems (ECRTS'08)*, 2008.

[11] L. Burdalo, A. Espinosa, A. Terrasa, and A. Garcia-Fornes. Experimental results of aperiodic fixed-priority preemptive policies in RTLinux. In *Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2007.

[12] G. Buttazzo. Rate-monotonic vs. EDF: Judgement day. *Real-Time Systems: The International Journal of Time-Critical Computing*, 29(1):5–26, 2005.

[13] G. Buttazzo and J. Stankovic. Red: Robust earliest deadline scheduling. In *Third International Workshop on Responsive Computing Systems*, New Hampshire, USA, September 1993.

[14] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[15] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE RTSS*, pages 295–304, Orlando, Florida, 2000.

[16] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, UK, December 2001. IEEE Computer Society Press.

[17] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 389–398, Miami, Florida, 2005. IEEE Computer Society.

[18] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-time Systems Symposium*, pages 257–267, Rio de Janeiro, December 2006. IEEE Computer Society Press.

[19] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing*, Honolulu, Hawaii, USA, March 2009.

[20] N. Fisher, M. Bertogna, and S. Baruah. The design of an EDF-scheduled resource-sharing open environment. In *28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona (USA), 2007.

[21] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modelling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2007.

[22] L. Sha, J.P.Lehoczky, and R. Rajkumar. Solutions for some parctical problems in prioritised preemptive scheduling. In

Proceedings IEEE Real-Time Systems Symposium, pages 181–191, 1986.

[23] L. Sha and S. S. Sathaye. Distributed system design using generalized rate monotonic theory. Technical Report CMU/SEI-95-TR-011, Carnegie Mellon University, 1995.

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

[25] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-teime environments. *IEEE Transactions on Computers*, 44(1), January 1995.