Sporadic Server Revisited *

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

Abstract

The Sporadic Server (SS) has been introduced to overcome the limitations of other Resource Reservation techniques in Fixed Priority scheduled systems. However, even if SS provides good performances with relation to both system responsiveness and schedulability, it presents some drawback mainly related to an increased scheduling overhead. Moreover, to our knowledge, it has not yet been adapted to deal efficiently with overrun situations, nor to work on multiprocessor platforms.

In this paper we introduce and prove the effectiveness of an improved SS implementation that allows a reduced overhead, as well as a fairer behavior during server overruns. Moreover, we show how this modified Sporadic Server can be efficiently exploited to provide temporal isolation in a multiprocessor platform, adapting already existing schedulability tests to the case under consideration. The simulations presented prove the effectiveness of our approach, paving the way for an efficient implementation in the Linux kernel.

1 Introduction

Typical embedded systems applications range from safety critical controls, like flight and defence 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 for the embedded system market. The “multicore revolution” is offering a significantly higher computing power at a limited cost. However, current applications are only partially able to exploit the possibilities offered by such powerful platforms, due to the lack of efficient scheduling techniques suited for multiprocessor architectures.

In this work, we will analyze the problem of scheduling a workload composed by hard, soft and non real-time tasks on a multiprocessor platform, in a fixed priority environment. In particular, we will improve and adapt to multiprocessor systems a previously proposed technique for the scheduling of aperiodic workloads on a uniprocessor systems: the Sporadic Server [29]. Our final goal is an efficient multiprocessor version of such server that will be implemented as a patch on the Linux kernel. Although there exist many commercial and academic fully real-time kernels, such as VxWorks 1, Erika 2, SHark 3, MarteOS 4, etc., we think that enabling the support for real-time workloads in general purpose operating systems is much more attractive. This is because a large number of off-the-shelf applications and libraries can thus be used, e.g., for user interface or other non real-time activities.

Among the various existing GPOSs, Linux is definitely one of the preferred choices, thanks to the open source license, the huge community supporting it, the portability to many hardware architectures and the astonishing number of existing applications already running on it. Furthermore, many real-time oriented modification of Linux exist, such as RTLinux 5, RTAI 6 or Xenomai 7. They basically introduce a software layer between the original OS and the hardware, to provide real-time applications with a minimum latency and a highly predictable environment. Finally, also the mainstream Linux kernel is being continuously enriched with real-time capabilities, proposed as separate patches and progressively integrated into the main branch. Such features include priority inheritance enabled mutexes, interrupts handled in schedulable threads, non-preemptable code sections with reduced lengths, and many others.

From an implementation point of view, the major real-time and general purpose operating systems are based on fixed priority scheduling. Even if the achievable schedulable utilization is lower, fixed priority systems are often

---

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

1 http://www.windriver.com/
2 http://erika.sssup.it/
3 http://shark.sssup.it/
4 http://marte.unican.es/
5 http://www.rtlinuxfree.com/
6 https://www.rtai.org/
7 http://www.xenomai.org
easier to implement, with a smaller scheduling overhead. Moreover, many system and application developers are still more comfortable using static priorities rather than relying on more efficient dynamic schedulers. Coming to multiprocessor scheduling, fixed priorities are even more attractive, since the gap in the achievable utilization between fixed and dynamic priority approaches is reduced. Optimum algorithms, with respect to utilizations exist and use dynamic priorities, but their run-time overhead could be overly high [9, 2].

The real-time extensions of the IEEE 1003 family of standards (well known as POSIX [1]) includes the specification of fixed priority preemptive scheduling by means of three policies: SCHED_FIFO, SCHED_RR and SCHED_SPORADIC. Describing them in detail is out of the scope of this document. We will just say that the last policy mimics the Sporadic Server. This technique, proposed by Sprunt et al. in [29], 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. In the next sections, we will tackle these problems, presenting the design decisions that drove our implementation of a multiprocessor Sporadic Server in the Linux kernel.

Organization of the paper The remainder of the paper is organized as follows: Sec. 2 introduces background concepts and definitions. Sec. 3 compares our work with the existing solutions. Sec. 4 shows schedulability conditions for the Sporadic Server in a multiprocessor systems. Sec. 5 and Sec. 6 detail the modifications we propose to the original server, in order to improve the implementation on a real architecture. Sec. 7 illustrates the results of our simulations. Finally, in Sec. 8 we draw our conclusions.

2 System Model

We will consider a shared memory multiprocessor real-time system, with $m$ identical (unit-capacity) processors $P_1, \ldots, P_m$. Each activity is referred to as a task $\tau_i$. Each periodic or sporadic task $\tau_i$ 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 $\tau_i$ 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 $\tau_i$ 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 Either in fixed or dynamic priority systems, the Resource Reservation framework (RR) is 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) $S_i$ with budget $Q$ and period $P_i$, that will provide some scheduling guarantee. Typically, it will enforce the tasks execution for at least $Q$ every $P_i$ time-units.

This is usually called bandwidth isolation, since each task serviced by a RR server is guaranteed a processor share of at least $\frac{Q}{P_i}$. Thus, no reciprocal interference between tasks in different reservations is allowed.

Multiprocessor Scheduling There are basically two approaches for real-time multiprocessor scheduling: partitioned and global. In the partitioned approach each task is statically assigned to a processor, while in global scheduling tasks can dynamically migrate among CPUs. Both approaches have advantages and drawbacks, which are not detailed here due to space constraints. For this work, we are interested in Fixed Priority (FP) global scheduling.

For general task system scheduled with FP, an upper bound on the schedulable utilization is $\left(\frac{1}{P_i} + \frac{1}{T_i}\right)$, as shown in [3]. However, no known priority assignment allows a schedulable utilization equal to the above upper bound. For Rate Monotonic, there is no guarantee to positively schedule any task set with utilization greater than 1, due to Dhall’s effect [21]. We will use a particular priority assignment that gives highest priorities to heavy tasks with utilization above a specified threshold, and schedules the remaining ones with RM. This algorithm, called Rate Monotonic with Utilization Separation (RM-US) in [5, 4], allows a larger schedulable utilization than with classic RM.

Regarding the schedulability analysis of multiprocessor sporadic task systems globally scheduled with Fixed Priority, no exact test is known. Sufficient schedulability tests have been presented in [5, 4, 13, 7, 12, 14].

Resource Contention Arbitration Real-time systems are often designed as a set of different tasks interacting through shared memory areas. When mutual exclusive access is required, proper arbitration throughout mutex semaphore is mandatory. Thus, in order to avoid unbounded priority inversion, many solutions have been proposed along the years. Some examples are the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP) [26], and the Stack Resource Policy (SRP) [6].

2.1 Fixed Priority Reservation Algorithms

An open and dynamic real-time system has often to deal with both periodic and aperiodic tasks. In such systems, common requirements are:
• guaranteed behavior for hard (periodic) activities;
• bandwidth isolation for soft (sporadic) activities;
• fast response times for non real-time activities.

Assuming hard real time workload to be periodic and guaranteed off-line, we need a mechanism that provides isolation and fast response time to, respectively, soft and aperiodic task, without jeopardizing the guaranteed behavior of hard tasks. Both these goals are usually obtained by using RR and servers. A server is a schedulable entity to whom one or more sporadic or aperiodic tasks are associated (or attached). When the server is scheduled, it selects one of the associated tasks for execution. We say that a server is backlogged if it has pending workload to execute.

In the field of fixed priority preemptive scheduling, many server based approaches have been proposed. The following paragraphs give a brief resume of some of the most relevant ones.

Polling Server (PS) [25] The server is implemented by a periodic task, activated at regular intervals and with a maximum service budget. In case there is no task to serve, the budget is depleted to zero until the next period. This causes a quite large response time for the served activities. From the schedulability point of view, this server can be treated as a normal periodic task.

Deferrable Server (DS) [30] It is basically a PS that preserves its budget even if there is no task waiting to run. Thus, service could start anytime, and proceed until the budget is exhausted. DS can provide a better response time than PS, but it introduces a larger schedulability penalty.

Sporadic Server (SS) [29] It uses a particular capacity planning mechanism. The budget of a SS 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 follows칠 (see Figure 1):

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 undergoes transition (1) at time \( t_1 \), the recharging time is set to \( t_1 + P \);
4. when the server undergoes transition (2) 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 \( [t_1, t_2) \).

Note that the recharging time and the amount of the recharge are set in two different instants; the first one when the server activates, the second one when it becomes idle.

Since the capacity is not discharged when the server is idle, SS is a bandwidth preserving server like DS, providing a good response time as reported by many studies [29, 17, 16]. Moreover, SS improves over DS for two main reasons: (i) it does not have the schedulability penalties associated to DS, and (ii) it does not need to replenish the budget when no task has executed, reducing the system overhead.

Other fixed priority servers, like the Priority Exchange server (PE) [30], are not described here due to space reason. They however have either more complex implementations or larger schedulability penalties.

2.2 Sporadic Server and POSIX

POSIX real-time profiles include features like asynchronous I/O, shared memory, timers, threads, priority inheritance and protection, etc. As for real-time CPU scheduling, the SCHED_SPORADIC policy present some analogies with the Sporadic Server algorithm. The main difference from the original SS is that when a server runs out of budget, it is not suspended, but its priority is downgraded to a low level. This makes it possible to implement a sort of basic reclaiming mechanism.

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 [10] that, although based on global EDF, is also applicable to every work-conserving algorithm, as is global FP. 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 [27], Sha et al. thoroughly studied the applicability of RM scheduling on multiprocessor distributed systems, using SS to schedule aperiodic activities. However, the main focus was on hardware and network-level real-time support for distributed or loosely coupled multiprocessor systems.
The adopted mechanism is similar to partitioned scheduling and, therefore, quite different from our perspective.

In [19], 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. The analysis is based on a Polling Server in a single processor hierarchical environment. Our work will instead address Sporadic Servers in a multiprocessor system.

Implementations of the Sporadic Server or similar periodic server policies have been presented for existing operating systems, like RTLinux [16, 28]. There are also both research papers [24, 8, 15] and recent attempts to implement the POSIX SCHED_SPORADIC policy in the Linux kernel [11]. Among commercial operating systems, support for POSIX SCHED_SPORADIC is only claimed by QNX Neutrino Microkernel 9, by RTEMS 10 and, recently, by Xenomai 7.

4 Sporadic Server for Multiprocessors

A Sporadic Server implemented according to the rules described in Section 2.1 does not need any particular mechanism to work as well on a multiprocessor environment scheduled with global Fixed Priority. However, differently from the uniprocessor case, it is not so immediate to conclude that all schedulability tests for sporadic task systems scheduled with global FP apply as well to systems that include sporadic servers. In the uniprocessor case, the equivalence, from a schedulability point of view, between a sporadic server and a sporadic task with identical parameters is easily proved exploiting the concept of critical instant. The response time of a job in a given collection of jobs is maximized when all jobs are released synchronously. However, there is no known critical instant in the schedulability analysis of multiprocessor sporadic task systems. Nevertheless, we will show in this section that at least two previously proposed schedulability tests for multiprocessor sporadic task systems can be applied as well to systems including sporadic servers.

The schedulability problem for systems scheduled with FP has been first addressed by Andersson et al. in [5, 4], where utilization bounds have been presented for RM and RM-US priority assignments. These bounds have been later improved by Bertogna et al. in [13], where the following result was proved.

**Theorem 1 (from [13])** A set of periodic or sporadic tasks with implicit deadlines is schedulable with the Rate Monotonic priority assignment on \( m \geq 2 \) processors if

\[
U_{tot} \leq \frac{m}{2}(1 - U_{max}) + U_{max}.
\]

A corollary of Theorem 1 allows achieving a larger schedulable utilization using RM-US[1/3], i.e., assigning higher priorities to tasks having utilization greater than 1/3, and scheduling the remaining ones with Rate Monotonic.

**Corollary 1 (from [13])** A set of periodic or sporadic tasks with implicit deadlines is schedulable with RM-US[1/3] on \( m \geq 2 \) processors if

\[
U_{tot} \leq \frac{m + 1}{3}.
\]

A different result for globally scheduled fixed priority systems has been derived in [14]. The following upper bound on the workload of a task \( \tau_i \), with slack \( \geq s_i \), in a window \( L \) is provided:

\[
W_i(L, s_i) = \min(C_i, L + D_i - C_i - s_i - N_i(L, s_i)T_i) + N_i(L, s_i)C_i,
\]

with

\[
N_i(L, s_i) = \left\lfloor \frac{L + D_i - C_i - s_i}{T_i} \right\rfloor.
\]

**Theorem 2 (from [14])** A lower bound on the slack of a sporadic task \( \tau_k \) scheduled with Fixed Priority on a multiprocessor platform composed by \( m \) identical processors is given by

\[
s_k = D_k - C_k - \left( \sum_{1 \leq k} \min(W_i(D_k, s_i), D_k - C_k + 1) \right),
\]

where this term is positive.

The previous theorem suggests a simple schedulability test for a sporadic task set \( \Gamma \):

- initially, all slack lower bounds are set to \( s_i = 0 \);
- the slack lower bound \( s_k \) of each task \( \tau_k \in \Gamma \), is computed using Equation (3); slacks are updated in decreasing priority order;
- if, for a task \( \tau_k \), 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.

The above results (Theorems 1 and 2, and Corollary 1) apply as well to systems in which periodic and sporadic tasks are scheduled together with Sporadic Servers. Analyzing the proof of Theorem 2 in [14], and Theorem 1 in [13], we see that both results 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 can be upper bounded by the same expressions used for the workload produced by a sporadic task having WCET and period equal to, respectively, the server budget and period.
Theorem 3 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 in Section 2.1, $S_i$ must be in Idle state throughout $[t_1, t_2)$. Therefore, when the server will activate again undergoing transition (1), 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 in Idle state — if the server budget is exhausted — or in Active state — when there is some budget left. 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 underwent transition (1) 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. □

To find an upper bound on the workload of a SS in an interval $L$, we can then simply consider the situation in which the server is always backlogged in $L$. In this case, the largest workload is produced when the server executes for $Q_i$ time-units before being fully recharged, and then it executes for $Q_i$ time-units at the beginning of each period. Such situation is identical to the worst-case situation considered in the proofs of Theorems 1 and 2 in [13, 14] for a sporadic task having period $P$ and WCET $Q$. Therefore, the schedulability tests presented in this section can be applied as well to sporadic servers, treating each SS $S_k$ as a sporadic task $\tau_k$ having $D_k = T_k = P_k$ and $C_k = Q_k$.

5 Sporadic Server with Budget Overruns

Enforcing the execution of serviced tasks for at most the server budget $Q$, i.e., avoiding overruns, is a key aspect. However, there are situations where a budget overrun is either impossible to avoid, or it is desirable. For example:

- if the server budget is not a multiple of the time accounting resolution of the operating system;
- if the system is experiencing high latencies, e.g., during OS kernel debugging, or while running inside a Virtual Machine;
- if we explicitly want the serviced task to execute for more than the server budget, e.g., when it holds a lock shared with other tasks.

The following sections further describe these three scenarios, in order to show they are likely to occur in real physical systems, and to describe our proposed solutions.

5.1 Limited Timer Resolution

In a modern OS, the account of the execution time of a task and the enforcement of a maximum duration could be implemented by means of either a timer, or the periodic tick mechanism. Anyway, there will always be a minimum possible resolution that one could rely on. Often, such resolution depends on both the OS implementation and the hardware platform. For example, high resolution timers have been recently introduced in the Linux kernel, so that, if allowed by proper hardware, timer with $100 \mu s$ resolution may be programmed.

As for task execution accounting, the Linux kernel measures how long a task has been running at each one of the following events:

- a periodic system tick;
- an enqueue or dequeue event (which does not necessarily happen during a system tick).

Furthermore, the tick has its own frequency, configured at kernel build time selecting one of the following values: $100$, $250$, $300$ or $1000$ Hz. Moreover, the frequency is subject as well to clock precision and stability issues.

Whenever the server budget is not an integral multiple of the resolution of the accounting mechanisms, some problem arises. Unfortunately this happens almost always, since, as we mentioned, the time resolution depends on hardware features and on the OS configuration. A more precise model of how timer resolution affects budget enforcing is given by the following observation.

Observation 1 The worst possible error in budget accounting due to tick resolution is equal to the tick period $P_{\text{tick}}$.

In fact, suppose that at a particular tick the current budget of an executing server $S_i$ is found barely positive. Thus, $S_i$ is not suspended even if it exhausts the budget immediately after. The budget will not be checked again until the next tick instance, i.e., after $P_{\text{tick}}$ time units. During that time, the server may overrun its budget by the same amount of time, as shown in the example in Figure 2.

Furthermore, since the tick delivery may be subject to a jitter $\Delta$, the bound may increase to $P_{\text{tick}} + \Delta$. If the timing parameters of the server are close to the tick period, severe
issues happens in terms of overrun. This effect can be reduced if an OS timer is used for budget enforcing, since its time resolution is likely to be smaller that the resolution of a periodic tick. However, having one timer for each server would introduce too much overhead, decreasing system performances.

5.2 Widened OS Latency

Wide kernel latencies can be modeled as temporary reductions of tick and timer resolutions. A relevant example is given by virtualized systems, where a (guest) OS is executed as a common process of a host. The guest could experience long delays and imprecise time accounting. For example, a tick of the guest fired at time \( t_G \), could be served at a later time \( (t_G + t_H) \), if at \( t_G \) the guest VM is not a running process of the host. Moreover, \( t_H \) may dynamically vary and can be quite big.

5.3 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 critical shared data [18, 23, 20] . Two applicable solutions for this problem 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 it is holding a shared lock.

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\(^{11}\). Instead, the second solution does not require any a priori information on the CS lengths, but it may break the bandwidth isolation of admitted reservations. This paper is mainly related to the second approach, in which a Sporadic Server budget overrun is likely to occur.

In the next sections, we will show how to enhance a Sporadic Server with efficient mechanisms to deal with server overruns.

\(^{11}\) 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 [22]

5.4 Strategies and Solution

As explained in the previous sections, enforcing the limitations on the budget of a Sporadic Server in a real scenario could be either impossible — in the first two scenarios — or unwanted — in the last one. It is therefore necessary to design efficient mechanisms to deal with server overruns. A reasonable requirement could be to “restore some fairness” by making the overrunning server somehow payback in its subsequent execution instances.

Something similar has already been proposed for other reservation strategies [20]. When using a Polling or a Deferrable Server (in fixed priority systems), or a CBS (in dynamic priority ones), it is sufficient to decrease the budget of the next server instance by the amount of the overrun. In this way, the server automatically pays back for its unexpectedly longer execution.

Unfortunately, applying this technique to a Sporadic Server is not so immediate, because of the following reasons:

- replenishment times are not periodic, but are set every time a server reactivates;
- replenishment amounts are not equal to the initial server budget value, but to the capacity consumed since the last activation.

Looking at Fig 3, we can see how the simple presented method works for DS (in (a)) but not for SS (in (b)). When a SS with budget \( Q \) executes for \((Q + O_1)\) time units, a payback amount of \( O_1 \) is subtracted from the next replenishment. However, if the replenishment was of \((Q + O_1)\), then the budget would be recharged to \( Q \), without any payback. Since we want an overrunning server to pay back, we can try setting a replenishment of \( Q \) (i.e., the initial server budget value), as in Fig 3(b). When \( O_1 \) is subtracted, the second instance of the task is only able to execute for \((Q - O_1)\), seemingly proving the effectiveness of the payback mechanism. Unfortunately, if the task then executes for \((Q - O_1)\) time units, the next server instance will be replenished by only \((Q - O_1)\) units, and the same is true for every following instance, propagating a lower server budget.

Moreover, any additional overrun will cause a further budget reduction, eventually depleting the server capacity. Thus, the payback mechanism turned out to produce a permanent budget leakage, as visible by the fourth instance.

![Figure 2](image-url)  
**Figure 2.** \( \tau_i \) experiencing overruns (in dark gray) due to coarse grained tick period.

![Figure 3](image-url)  
**Figure 3.** “Naive” payback for DS (a) and SS (b).
**Our Solution** To overcome the limits of the above naive implementations, more efficient mechanisms are needed in case of budget overrun. We propose to enhance a Sporadic Server with the following rules:

- 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 to the initial server budget $Q$, i.e., if the server executed for $(Q + O)$ units, the recharge is set to $Q$;

- at each replenishment time $t$:
  1. the corresponding recharge is added to the current server budget $q$;
  2. if $O_{pool} > 0$, both the overrun pool $O_{pool}$ and the current budget $q$ are decreased by $\min\{q, O_{pool}\}$;
  3. a recharge of $\min\{q, O_{pool}\}$ units is set after one period, i.e., at time $t + P$.

With these modifications, the payback mechanism works even if the overrun $O$ is greater than $Q$. In that case, one or more server instances are skipped to properly recover from the overrun.

The main features of the proposed solution are:

- a “fair” payback mechanism that decreases as soon as possible the capacity of an overrunning server.

- the preservation of the server capacity, avoiding the budget leakage problem.

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 next replenishment time. This is an effective solution because it is consistent with the philosophy of the Sporadic Server, that is, (i) a replenishment is posted only after some budget is consumed, and (ii) the recharge corresponds to the consumed capacity.

**Main Properties of Our Solution** We present here a few valuable properties of the proposed solution.

As long as the server budget is not negative, our server behaves as the original SS. When instead there is an overrun, the budget will be decreased by $\min\{q, O_{pool}\}$ at the next replenishment time $t$, and a recharge of the same amount is posted at time $(t + P)$. Note that, when $q > O_{pool}$, the server has remaining budget and it may restart running at $t$. When it becomes Idle, a recharge equal to the amount of execution received since $t$ is posted at time $(t + P)$, that is, at the same time instant of the ancillary replenishment we have added. Therefore, there is one single cumulative replenishment at time $(t + P)$, so that no additional overhead is introduced.

Another, more important, property concerns the payback mechanism we implemented. Consider a SS with budget $Q$ and period $P$, in a system where the largest overrun experienced is equal to $\Delta$. A classic SS without any payback mechanism may experience subsequent overruns, executing for $(Q + \Delta)$ in each period. The cumulative overrun after $n$ periods can then be of $n\Delta$. Considering instead the payback mechanism we introduced, a smaller cumulative overrun is experienced.

**Theorem 4** Consider a SS implemented with our payback mechanism and let $\Delta$ be the largest overrun it may experience. In any interval $L$, the cumulative execution of SS cannot exceed by more than $\Delta$ time-units the cumulative execution of a non-overrunning server with identical budget and period.

**Proof.**
For Theorem 3, the largest workload in $L$ can be found when the SS is continuously backlogged throughout $L$. If an overrun of $\Delta$ occurs, the subsequent instance of SS is subject to a budget reduction of $\Delta$. 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 subsequent overrunning instance. Therefore, the overall execution of a SS with payback mechanism cannot exceed the ideal behavior ($Q$ every $P$) by more than $\Delta$ time-units. $\square$

Thanks to the payback mechanism, our SS can then significantly reduce the schedulability penalties associated to budget overruns. To highlight this effect, we hereafter present how the schedulability test of Theorem 2 can be modified to take into account server overruns. Let $\Delta_i$ be the largest overrun that a SS $S_i$ — having period $P_i$ and budget $Q_i$ — may experience. We will separately consider the cases with and without the payback mechanism.

When no payback mechanism is present, it can be shown (see [14]) that the largest workload of an overrunning server $S_i$ is produced in the situation of Figure 4(a). In this case, the server executes for $(Q_i + \Delta_i)$ time units just $s_i$ time-units before a full replenishment, and then it executes for $(Q_i + \Delta_i)$ time units at the beginning of each subsequent period. A sufficient schedulability test easily follows from (1), (2) and (3), using $D_i = T_i = P_i$ and replacing each $C_i$ occurrence with $(Q_i + \Delta_i)$. We will denote such test as $SS_{Original}$.

When instead the payback mechanism is introduced, a continuously backlogged SS may execute for up to $(Q_i + \Delta_i)$ time units only for one instance. All subsequent instances may execute for at most $Q_i$ time units. So, an upper bound $W_i^p(L)$ on the execution allowed in $L$ can again be found applying techniques from [14]. It is possible to show that a worst-case condition is given by the situation of Fig-
6.1 Reducing Power and System Overhead

As showed by the example above, performing a replenishment exactly at the scheduled instant may be detrimental to system performances. A better solution would be to schedule a single cumulative budget replenishment when the server becomes backlogged again.

Formally speaking, our modified replenishment posting rules read as follows:

- when the server becomes non backlogged, each timer associated to a replenishment event is stopped;
- when the server becomes backlogged,
  - each pending recharge associated to a past replenishment event is immediately issued;
  - each timer associated to a future replenishment event is armed.

7 Simulations

Simulation based study has been conducted to demonstrate the effectiveness of the proposed approach. We considered a platform composed of \( m \) processors, with \( m \) equal to 2, 4 or 8 processors, upon which a set of sporadic servers are scheduled using RM. A randomized distribution of sporadic servers \( SS \) 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 \([10000, 1000000]\);
- utilizations \( U_i \) from an exponential distribution with mean \( \lambda = 0.25 \), and budgets accordingly computed as \( Q_i = U_i P_i \).

If test \( SS_{\text{Payback}} \) 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. The above procedure continues until \( 10^6 \) sets have been positively considered for evaluation.

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 \( SS_{\text{Payback}} \) and \( SS_{\text{Original}} \). The maximum overrun has been set equal to the OS tick, which can be 10000, 4000 or 1000 time units. What we measured is that without the payback mechanism there are significant losses. With
<table>
<thead>
<tr>
<th>$m$</th>
<th>$P_{\text{tick}} = 1000$</th>
<th>$P_{\text{tick}} = 4000$</th>
<th>$P_{\text{tick}} = 10000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>44%</td>
<td>45%</td>
<td>47%</td>
</tr>
<tr>
<td>4</td>
<td>48%</td>
<td>49%</td>
<td>51%</td>
</tr>
<tr>
<td>8</td>
<td>53%</td>
<td>54%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Figure 6. Schedulability loss of SS$_{\text{Original}}$ with respect to SS$_{\text{Payback}}$.

$m = 2$ processors and $P_{\text{tick}} = 1000$, 44% of the generated task sets are schedulable only with SS$_{\text{Payback}}$ and not with SS$_{\text{Original}}$. Increasing $P_{\text{tick}}$ to 4000 and 10000, the schedulability loss of SS$_{\text{Original}}$ 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$_{\text{Original}}$ with 4 and 8 processors increases to, respectively, 48% and 53%. All results are summarized in Figure 6.

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 $10^7$ time-units.

Figure 7 shows the number of useless replenishment the system undergoes if running the original SS algorithm. As we can see, a marked bursty behavior and a skip probability up to 50%, may cause a large number of spurious wake-ups, which are completely avoided by our modified version of the SS. Note that the number of useless wake-ups starts decreasing for skip probabilities larger than 50%. This is because, since fewer jobs are executed, less replenishing events are posted as well.

8 Conclusions and Future Works

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.

References


