# Defects of the POSIX Sporadic Server and How to Correct Them

Mark Stanovich
Theodore P. Baker
An-I Wang
Florida State University, USA

Michael González Harbour
Universidad de Cantabria, Spain

## Abstract

*The specification of the sporadic server real-time scheduling policy in the* IEEE POSIX *standard is defective, and needs to be corrected. Via experiments using a* POSIX *sporadic server implementation under Linux, as well as simulations, we have shown and confirmed previously unreported defects. We propose and demonstrate a corrected sporadic server formulation that eliminates these defects without changes to the syntax of the API or any significant increase in implementation complexity.*

## 1 Introduction

During the late 1980's and early 1990's, a major initiative was undertaken to disseminate then-recent technological developments in real-time systems through programming language and operating system standards. One success of this effort was the inclusion of support for preemptive fixed-task-priority scheduling policies in the IEEE POSIX standard application program interface (API) for operating system services. That standard has since been rolled into the Unix standard of the Open Group[10] and is implemented by Linux and many other operating systems. However, as advances have continued to be made in the understanding of real-time scheduling, very little has been done to update the POSIX standard.

In this paper, we make a case for the need to correct the SCHED_SPORADIC scheduling policy specification in the existing POSIX real-time scheduling standard. We show that the current specification has several critical technical flaws, argue for the importance of correcting these flaws, and provide specific suggestions for how they may be corrected.

The SCHED_SPORADIC policy is important because it is the only scheduling policy supported by the POSIX standard that enforces an upper bound on the amount of high-priority execution time that a thread can consume within a given time interval. As such, it is the only standard scheduling policy that is potentially suitable for compositional schedulability analysis of an "open" real-time system in the sense of [6], and the only one that is suitable as the basis for a virtual computing resource abstraction for compositional analysis of a hierarchical scheduling scheme such as those studied in [19, 16, 13, 5, 2, 17].

The SCHED_SPORADIC policy is a variation on the *sporadic server* scheduling concept, originally introduced by Sprunt, Sha, and Lehoczky [18]. Conceptually, a sporadic server has execution time *budget*, which it consumes while it executes at a given *server priority*, and which is *replenished* according to a rule that approximates the processor usage of a conceptual set of sporadic tasks with a given *period*. The intent is that the worst-case behaviors of the server – both the minimum level of service it provides and the maximum amount of processor time it consumes – can be modeled by an equivalent periodic task, whose worst-case execution time is equal to the server budget and whose period is equal to the server period. We call this property the *periodic task analogy*.

This alleged equivalence of a sporadic server to a periodic task is often cited in the literature. For example, [5] says that "Sprunt proved that in the worst-case the interference due to a Sporadic Server is equivalent to that of a simple Periodic Server", and [16] says "in the worst case, a child reserve [implemented as a sporadic server] behaves like a classical Liu and Layland periodic task".

Unfortunately, the original formulation of the sporadic server scheduling algorithm published in [18] – commonly called the *SpSL* sporadic server – violates the above assertions. A defect in the replenishment rules allows a thread to consume more processor time than the allegedly-equivalent periodic task. We do not know for certain who first discovered this defect. One of us, cited as a source in [1], first learned of it from Raj Rajkumar. It is also described in [15].

Several proposals for correcting this defect have been published, including one in [1], several variations in [15], and an adaptation for deadline scheduling in [9]. In particular, the formulation of the sporadic server scheduling policy in the POSIX standard was widely believed to have corrected this defect. For example, [1] says: "The POSIX sporadic server algorithm (PSS) provides an effective and safe solution that does not allow any budget overruns".

Believing the POSIX sporadic server to be correct, we proposed in prior work [12] that the device driver processing of incoming and outgoing network traffic be executed by a thread that is scheduled using the SCHED_SPORADIC policy. Our original experiments with a tick-based implementation of the scheduling policy suggested that the performance would be improved by finer-grained management of time. However, in follow-up experiments using finer-grained time measurement, we were surprised to see that the server's actual processor utilization was significantly higher than that of a periodic task with the same budget and period. When we looked for the cause of this anomalous behavior, we discovered two flaws in the POSIX specification, which we believe need urgent attention.

To that end, this paper demonstrates the following facts:

1. The POSIX sporadic server algorithm's replenishment rules suffer from an effect that we call "premature replenishment". We provide an example in which this defect allows a server to use an average of 38 percent more execution time than the analogous periodic task.
2. The POSIX sporadic server algorithm also suffers from an unreported defect, which we call "budget amplification". This defect allows a server to use arbitrarily close to 100 percent of the processor time, regardless of how small the server's budget may be.
3. These defects can be corrected by modifications to the POSIX sporadic server specification, which are described in this paper.

In support of the above, we report experiences with an implementation of the POSIX sporadic server in the Linux kernel, which clearly demonstrate the budget amplification effect on a task. We also report on simulations using pseudo-random job arrivals that provide some insight into the likelihood of encountering the effects of the above two defects in practice.

We additionally propose a change to the POSIX sporadic server specification to address a practical deficiency relating to the inability to lower the priority of a sporadic server sufficiently when it is out of budget.

## 2 An Ideal Sporadic Server Model

The preemptive scheduling of periodic task systems is well understood and has been studied extensively, starting with the pioneering work of [14] and the recursive response-time analysis technique of [11].

A *periodic server* is a mechanism for scheduling an aperiodic workload in a way that is compatible with schedulability analysis techniques originally developed for periodic task systems. Aperiodic requests (jobs) are placed in a queue upon arrival. The server *activates* at times $t_1, t_2, \ldots$ such that $t_{i+1} - t_i = T_s$, where $T_s$ is the nominal server *period*, and executes at each activation for up to $C_s$, where $C_s$ is the server *budget*. If the server uses up its budget it is preempted and its execution is suspended until the next period. If the server is scheduled to activate at time $t$ and finds no queued work, it is deactivated until $t + T_s$. In this way the aperiodic workload is executed in periodic bursts of activity; *i.e.*, its execution is indistinguishable from a periodic task.

A *primitive sporadic server* is obtained from a periodic server by replacing the periodic constraint $t_{i+1} - t_i = T_s$ by the *sporadic constraint* $t_{i+1} - t_i \geq T_s$. That is, the period is interpreted as just a lower bound on the separation between activations. The sporadic constraint guarantees that the worst-case preemption caused by a sporadic task for other tasks is not greater than that caused by a periodic task with the same worst-case execution time and period. In other words, the processor demand function (and therefore a worst-case residual supply function for other tasks) of the server will be no worse than a periodic task with period $T_s$ and worst-case execution time $C_s$. That is, the periodic task analogy holds.

A primitive sporadic server has an advantage over a periodic server in greater *bandwidth preservation*; that is, it is able to preserve its execution time budget under some conditions where a periodic server would not. If there are no jobs queued for a sporadic server at the time a periodic server would be activated, the sporadic server can defer activation until a job arrives, enabling the job to be served earlier than if it were forced to wait to be served until the next period of the periodic server.

An *ideal sporadic server* is a generalization based on a conceptual swarm of unit-capacity sporadic tasks, called "*unit servers*" or just "*units*", for short. The basis for this generalization is the observation that the worst-case analysis techniques of [14] and [11] allow a set of periodic or sporadic tasks with the identical periods to be treated as if they were a single task, whose execution time is the sum of the individual task execution times. That is, the worst-case interference such a swarm of identical sporadic tasks can cause for other tasks occurs when all the tasks are released together, as if they were one task. Although the worst-case interference for lower-priority tasks caused by such a swarm of sporadic servers remains the same as for a single periodic server task, the average response time under light workloads can be much better. Indeed, studies have shown that sporadic servers are able to achieve response times close to those of a dedicated processor under light workloads, and response times similar to those of a processor of speed $u_s = C_s/T_s$ under heavy loads.

Since the overhead of implementing a server as a swarm of literal unit-capacity sporadic servers would be very high, published formulations of sporadic server scheduling algorithms attempt to account for processor capacity in larger chunks of time, called *replenishments*. Each replenishment $R$ may be viewed as representing a cohort of $R.amt$ unit servers that are eligible to be activated at the same replenishment time, $R.time$. For such a sporadic server formulation to satisfy the periodic task analogy, the rules for combining unit servers into replenishments must respect the sporadic constraint.

**Observation 1** *If $R$ represents a cohort of unit servers that were activated together at some time $t$ and executed during a* busy interval *containing $t$* [1]*, the sporadic constraint will be satisfied so long as $R.time \geq t + T_s$.*

**Observation 2** *The sporadic constraint is preserved if $R.time$ is advanced to any later time.*

**Observation 3** *The sporadic task constraint is preserved if a replenishment $R_1$ is merged with a replenishment of $R_2$ to create a replenishment $R_3$ with $R_3.amt = R_1.amt + R_2.amt$ and $R_3.time = R_1.time$, provided that $R_1.time + R_1.amt \geq R_2.time$.*

**Proof**
Suppose cohorts corresponding to $R_1$ and $R_2$ are activated at $R_1.time$. Since unit servers are indistinguishable within a cohort, we can assume that those of $R_1$ execute first and so cannot complete sooner than $R_1.time + R_1.amt$. Since $R_1.time + R_1.amt \geq R_2.time$, by the time $R_1$ completes the replenishment time $t_2$ will have been reached. So, none of the unit servers in the combined $R_3$ can activate earlier than if $R_1$ and $R_2$ are kept separate. □

## 3 The POSIX Sporadic Server

The POSIX sporadic server policy specified in [10] superficially resembles the ideal model. A thread subject to this policy has a native priority, specified by the parameter *sched_priority*, a budget $C_s$ specified by the parameter *sched_ss_init_budget*, and a period $T_s$ specified by the parameter *sched_ss_repl_period*. The thread has a numerical attribute, called the *currently available execution capacity*, which abstracts a set of unit servers that are eligible for activation at a given time (because their last activations are all at least $T_s$ in the past), and a set of pending replenishments, which abstract sets of unit servers that are not yet eligible for activation (because the last activation is less than $T_s$). If the POSIX specification were in agreement with the ideal model, each replenishment $R$ would correspond to a cohort of units that executed within a busy interval of the server and $R.time$ would be earliest time consistent with Observation 1. However, the POSIX rules for handling replenishments fail to enforce the sporadic constraint at the unit server level, and so break the periodic task analogy.

In this section we compare the POSIX sporadic server policy and its terminology to the ideal model described previously [2]. These comparisons will be used to explain how the resulting defects occur.

### 3.1 Budget Amplification

POSIX differs from the ideal model by limiting a server's execution "to at most its available execution capacity, *plus the resolution of the execution time clock used for this scheduling policy*". Some such allowance for inexact execution budget enforcement is essential in a practical implementation. Typically budget enforcement latency can vary from zero to the maximum of the timer latency and the longest non-preemptable section of the system calls that a server may perform. POSIX seems to err in stating that when "the running thread ... reaches the limit imposed on its execution time ... the execution time consumed is subtracted from the available execution capacity (*which becomes zero*)." The specified one-tick enforcement delay mentioned above allows the server budget to become negative by one tick, and in reality, larger overruns must be expected. POSIX allows for this elsewhere by stating that "when the running thread with assigned priority equal to *sched_priority* becomes a preempted thread ... and the execution time consumed is subtracted from the available execution capacity ... If the available execution capacity would become negative by this operation ... *it shall be set to zero*". POSIX attempts to compensate for the downstream consequences of forgiving such overruns by specifying that if as a result of a replenishment "the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*." However, this is an oversimplification, which cannot be translated into the ideal model.

This oversimplification of the ideal model leads to the defect we refer to as *budget amplification*. That is, the size of a replenishment can grow as it is consumed and rescheduled over time.

[1] A *busy interval* is a time interval during which the processor is continually busy executing the server and tasks that cannot be preempted by the server.

[2] We regret that the page limit prevents us from reproducing the full specification from the POSIX standard [10], but it is available for free access at the website of The Open Group (http://www.opengroup.org/bookstore/catalog/c082.htm).

When an overrun occurs, the POSIX specification states that the available execution capacity should be set to zero and that a replenishment should be scheduled for the amount of the time used since the *activation_time*. At this point, the sporadic server has used more units than it had. This increased amount is scheduled as a future replenishment. This would not be a big problem as long as the sporadic server were charged for this amount of time. However, setting the execution capacity to zero means that the overrun amount is never charged, thereby increasing the total capacity the server can demand within its period.

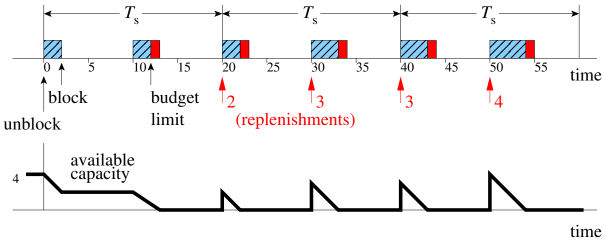


**Figure 1.** Budget amplification anomaly.

While POSIX attempts to diminish such effects by rounding the currently available execution capacity down to the initial budget, this effort is not sufficient. Consider the example illustrated in Figure 1. Here the resolution for the execution time clock is 1 time unit. At time 0, the server executes for two time units and schedules a replenishment of two time units at time 20. At time 10, the server again begins execution, but at time 12 it has not completed executing and therefore is scheduled to stop running at its high priority. The server is able to execute an additional time unit before actually being stopped, as permitted in the POSIX specification. At time 13, a replenishment is scheduled at time 30 for the amount of capacity consumed, which in this case is 3, and the available execution capacity is set to zero. Now, the sum of pending replenishments is greater than the initial budget of 4, but within the leeway provided by the specification. This sequence of receiving one time unit of additional execution capacity repeats with the intervals of server execution beginning at 20, 30, 40, and 50. By the time the replenishment for the execution interval beginning at 30 is scheduled, the sum of pending replenishments is 2 time units greater than the initial budget. If this scenario continues each overrun will contribute an increase to the total execution time available to the sporadic server. As long as each replenishment is below the maximum budget, this amplification may continue. In this case, each replenishment can grow to at most 5 time units (4 due to the initial budget limit and 1 for the permitted clock resolution).

With this defect, by breaking the budget into small enough fragments a server can achieve an execution capacity arbitrarily close to 100%.

| Task | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 10 | 200 | 20 |
| $\tau_2$ | 20 | 50 | 50 |
| $\tau_3$ | 49 | 200 | 100 |

**Table 1. Periodic task set for premature replenishment example.**

### 3.2 Premature Replenishments

POSIX specifies that "a replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time". This has the effect of maintaining a single activation time for all currently available units. This is inconsistent with the ideal model, because it fails to preserve the minimum replenishment time (earliest next activation time) of a replenishment (cohort of server units) if the server is in a busy period when a replenishment arrives. A consequence is that a replenishment can arrive earlier than its required minimum offset from the previous arrival, resulting in what we refer to as a premature replenishment.

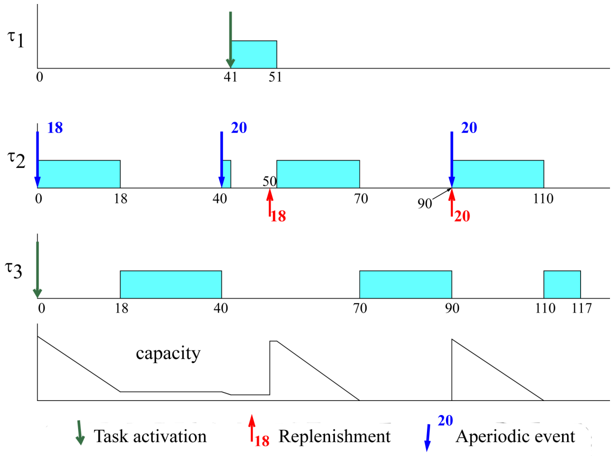


**Figure 2.** Execution sequence showing a replenishment that occurs prematurely.

The following example illustrates the premature replenishment defect. Consider a scenario with three independent periodic tasks, given a deadline-monotonic priority ordering and parameters (worst-case execution time, period, relative deadline) shown in Table 1. Response time analysis [11] obtains a worst-case response time for task $\tau_3$ of 99 time units:

$$R_3 = \left\lceil \frac{99}{200} \right\rceil C_1 + \left\lceil \frac{99}{50} \right\rceil C_2 + C_3 = 10 + 2 \cdot 20 + 49 = 99$$

Suppose task $\tau_2$ is a sporadic server serving aperiodic events. The sporadic server is given an execution capacity

$C_2 = 20$, and a replenishment period $T_2 = 50$. Under the ideal sporadic server model the worst-case response time of $\tau_3$ would be 99. However, in the execution sequence shown in Figure 2, the response time of $\tau_3$ is 117 (and therefore its deadline is missed). In this sequence, aperiodic events arrive at times $\{0, 40, 90\}$, with respective execution-time demands of $\{18, 20, 20\}$. Task $\tau_3$ is activated at $t = 0$, while task $\tau_1$ is activated at $t = 41$. We can see that when the second aperiodic event arrives at $t = 40$, the execution capacity of the sporadic server is above zero (its value is 2), so the activation time is recorded as 40, and the aperiodic event starts to be processed. At time 41, $\tau_1$ preempts the execution of the sporadic server. When the replenishment of the first chunk of execution time occurs at $t = 50$, 18 is added to the available execution capacity (1 unit at that time), and the activation time remains unchanged (because the server is still active). This violates the ideal model, by effectively merging a cohort of 18 units not permitted to activate until time 50 with a cohort of two units that activated at time 40. When the aperiodic event is fully processed, a replenishment of 20 time units is scheduled to happen at $t = 90$. This allows the service of three long aperiodic events to preempt task $\tau_3$, instead of the two that would happen in the ideal model.

### 3.3 Unreliable Temporal Isolation

In many real-time systems there is a need to provide *temporal isolation* between tasks or between sub-systems. That is, when one composes subsystems one wants a guarantee that if a task in one subsystem fails to complete within its allotted time budget it cannot cause a task in another subsystem to miss a deadline. As mentioned in the Introduction, sporadic server scheduling has been proposed in a number of papers on compositional and hierarchical scheduling for "open" real-time systems, as a means of achieving such temporal isolation.

The theoretical formulation of the sporadic server in [18] provides temporal isolation by requiring that when a task runs out of budget, it is not allowed to execute until its budget is replenished. The POSIX formulation differs in allowing a sporadic server to continue execution after it has exhausted its budget, albeit at a lower (*background*) priority, specified by the parameter *sched_ss_low_priority*. The apparent intent behind this feature is to allow a server to make use of otherwise-idle time. This feature is compatible with the ideal model so long as *sched_ss_low_priority* is below the priority of every critical task. However, POSIX also specifies that each scheduling policy has a range of valid priorities, which is implementation defined. Further, the statement that the SCHED_SPORADIC policy "is identical to the SCHED_FIFO policy with some additional conditions" has been interpreted by some to mean that the range of priorities for these two policies should be the same. For example, in Linux the priorities for SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR are identical, while priorities for SCHED_OTHER are strictly lower. This means that a thread under any real-time policy can lock out all SCHED_OTHER threads, breaking temporal isolation.

This problem has been recognized. The consequence is that the Linux kernel implements *real-time throttling* [20, 3], at the expense of breaking POSIX compliance. Real-time throttling ensures that in a specified time period, the non-real-time threads receive a minimum amount of time on the CPU. Once the budget for all real-time threads is consumed in the period the CPU is taken away from the real-time threads to provide CPU time to the non-real-time threads. This mechanism prevents real-time threads from locking up the system, but it is very coarse. There is only one budget and period, defined system wide. The default budget is 950 msec of real-time execution time per 1 second period. This means that any real-time thread can experience a (rather large and possibly fatal) preemption of 5 msec.

## 4 Corrected Sporadic Server Algorithm

In this section we provide a corrected version of the POSIX sporadic server. We then go on to explain how this new version corrects the defects mentioned previously.

Each server $S$ has a replenishment queue $S.Q$, which may contain a maximum of $S.max\_repl$ replenishments. Each replenishment $R$ has time $R.time$ and an amount $R.amt$. The queue $S.Q$ is ordered by replenishment time, earliest first. The sum of the replenishment amounts is equal to the server's initial budget

$$S.budget = \sum_{R \in S.Q} R.amt$$

A server is in foreground mode, competing for processor time at $S.foreground\_priority$ or in background mode, competing at $S.background\_priority$. Whenever $S$ is in foreground mode, its execution time is accumulated in the variable $S.usage$. The currently available execution capacity of the server is computed as

$$S.capacity = \begin{cases} 0 \text{ if } S.Q.head.time > Now \\ S.Q.head.amt - S.usage \text{ otherwise} \end{cases}$$

$S$ is in foreground mode whenever $S.capacity > 0$, and should perform a BUDGET_CHECK as soon as possible after the system detects that $S.capacity \leq 0$ (the server may change to background mode). To detect this condition promptly, event $S.exhaustion$ is queued to occur at time $Now + S.capacity$ whenever $S$ becomes a running task at its foreground priority. The system responds to event $S.exhaustion$ by updating $S.usage$ with the amount of execution time used at the foreground priority since the last update and then executing BUDGET_CHECK (Figure 3).

```
BUDGET_CHECK

if $S.Capacity \leq 0$ then
    while $S.Q.head.amt \leq S.usage$ do
        ▷ Exhaust and reschedule the replenishment
        $S.usage \leftarrow S.usage - S.Q.head.amt$
        $R \leftarrow S.Q.head$
        $S.Q.pop$
        $R.time \leftarrow R.time + S.Period$
        $S.Q.add(R)$
    if $S.usage > 0$ then ▷ $S.usage$ is the overrun amt.
        ▷ Budget reduced when calculating $S.capacity$
        ▷ Due to overrun delay next replenishment
        $S.Q.head.time \leftarrow S.Q.head.time + S.Usage$
        ▷ Merge front two replenishments times overlap
        if $S.Q.size > 1$ and $S.Q.head.time \geq S.Q.head.next.time$ then
            $a \leftarrow S.Q.head.amt$
            $S.Q.pop$ ▷ remove head from queue
            $S.Q.head.amt \leftarrow S.Q.head.amt + a$
    if $S.capacity = 0$ then ▷ $S.Q.head.time > Now$
        $S.priority \leftarrow S.background\_priority$
        if $\neg\, S.is\_blocked$ then
            $S.replenishment.enqueue(S.Q.head.time)$
```

**Figure 3.** Pseudo-code for budget over-run check.

The system also calls BUDGET_CHECK when $S$ is executing in foreground mode and becomes blocked or is preempted, after cancelation of event $Q.exhaustion$. If $S$ blocks while in foreground mode, after BUDGET_CHECK the system executes procedure SPLIT_CHECK (Figure 4).

```
SPLIT_CHECK

if $S.usage > 0$ and $S.Q.head.time \leq Now$ then
    $remnant \leftarrow S.Q.head.amt - S.Usage$
    if $S.Q.head.size = S.max\_Repl$ then
        ▷ Merge remnant with next replenishment
        $S.Q.pop$
        $S.Q.head.amt \leftarrow S.Q.head.amt + remnant$
    else
        ▷ Leave remnant as reduced replenishment
        $S.Q.head.amt \leftarrow remnant$
    $S.Q.add(S.usage, S.Q.head.time + S.period)$
```

**Figure 4.** Pseudo-code for conditionally splitting a replenishment.

If $S$ goes into background mode while it is not blocked (in BUDGET_CHECK) or if it becomes unblocked while in background mode (in SPLIT_CHECK) event $S.replenishment$ is queued to occur at time $S.Q.head.time$. The system responds to event $S.replenishment$ by setting $S.priority$ to $S.foreground\_priority$, which may result in $S$ being chosen to execute next.

If $S$ becomes unblocked the system executes procedure UNBLOCK_CHECK, shown in Figure 5.

```
UNBLOCK_CHECK
    ▷ Advance earliest activation time to now.
if $S.capacity > 0$ then
    if $S.priority \neq S.foreground\_priority$ then
        $S.priority \leftarrow S.foreground\_priority$
    $S.Q.head.time \leftarrow Now$
    ▷ Merge available replenishments
    while $S.Q.Size > 1$ do
        $a \leftarrow S.Q.head.amt$
        if $S.Q.head.next.time \leq Now + a - S.Usage$ then
            $S.Q.pop$ ▷ remove head from queue
            $S.Q.head.amt \leftarrow S.Q.head.amt + a$
        else
            exit
else
    $S.replenishment.enqueue(S.Q.head.time)$
```

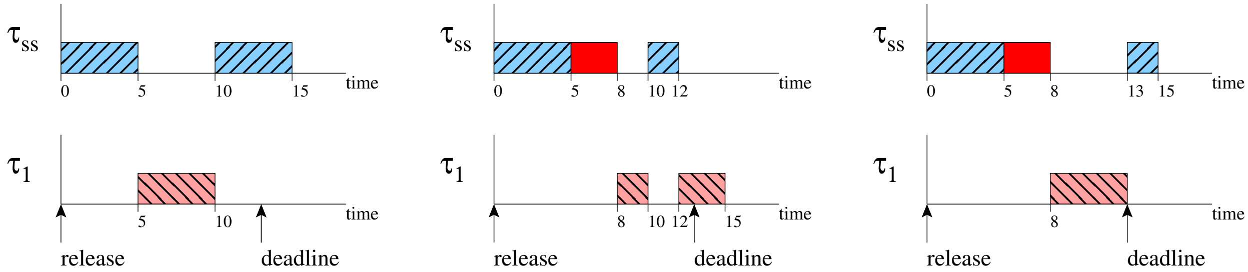
**Figure 5.** Pseudo-code for unblocking event.

## 4.1 Correcting for Budget Amplification

In [9] a solution for blocking effects caused by a deadline sporadic server is provided, where overruns will be charged against future replenishments. This mechanism is adapted to allow our modified sporadic server to handle overruns properly and inhibit the budget amplification effect.

Recall that amplification occurs when a replenishment is consumed in its entirety with some amount of overrun. This overrun amount is added to the replenishment and scheduled at a time in the future. The POSIX sporadic server is never charged for the overrun time because a negative budget is immediately set to zero.

A simple fix would be to just allow the currently available execution capacity to become negative. This prevents the amplification effect by keeping the capacity plus replenishments equal to the initial budget, thereby limiting execution within a server period to at most the initial budget plus the maximum overrun amount. However, since the replenishment is larger by the overrun amount, the server is still not being properly charged for the overrun.



(a) normal operation (b) overrun affecting deadline of $\tau_1$ (c) postponing replenishment time

**Figure 6.** Postponing replenishments after an overrun.

To handle this overrun properly, the time used will be charged against a future replenishment. To be clear, we are not preventing the overrun from occurring, only once it does occur, future overruns of the same size will be prevented from accumulating and only permitted to occur once per interval of continuous server execution.

In the BUDGET_CHECK procedure, the compensation for an overrun is handled. An overrun has occurred when the sum of the replenishment amounts with times less than or equal to the current time exceed the server's current usage ($S.usage$). This overrun amount is charged against future replenishments as if that time has already been used. The while loop starting at line 2 of BUDGET_CHECK iterates through the replenishments, charging as needed until the $S.usage$ is less than the replenishment at the head of the $S.Q$. At this point, the overrun amount will remain in $S.usage$. Therefore, when the next replenishment arrives it will immediately be reduced by the amount in $S.usage$ according to the calculation of $S.capacity$. This prevents the POSIX amplification effect by ensuring that overrun amounts are considered borrowed from future replenishments.

The intervals of sporadic server execution that correspond to a given replenishment are spaced apart by the server's period. This spacing allows lower priority tasks to receive execution time. However, when there is an overrun, the time between such intervals may shrink. For instance, consider Figure 6a. The execution of $\tau_1$ fits nicely between the two intervals of server execution. However, if an overrun occurs in the first execution interval of $\tau_{ss}$, the time meant for $\tau_1$ is occupied, forcing $\tau_1$ to start execution later. Since the next replenishment of $\tau_{ss}$ arrives during the execution of $\tau_1$, $\tau_{ss}$ is permitted to preempt $\tau_1$. This further postpones the completion of $\tau_1$ causing a missed deadline as illustrated in Figure 6b.

The overrun time used during the interval of execution corresponding to the first replenishment is borrowed from the second one. This time was used early. If this time is borrowed, the borrowed units should be taken from the *front* of the next available replenishment. That is, the time of the next replenishment should be postponed by the overrun amount. This is done in line 9 of the BUDGET_CHECK procedure. With this postponement we see that in Figure 6c, $\tau_1$ is able to meet its deadline.

## 4.2 Correcting the Premature Replenishments

Premature replenishments occur when one or more unit-capacity servers violate the *sporadic constraint*. The POSIX sporadic server experiences premature replenishments due to its simplified tracking of activation times. When a replenishment arrives, it is immediately merged with any replenishment that has an activation time less than or equal to the current time. This may result in invalid merging of replenishment cohorts, allowing a replenishment to be used earlier than the earliest activation time that would be consistent with the sporadic constraint.

To maintain the *sporadic constraint*, we must ensure that each cohort be separated by the server's period ($T_s$). In the corrected algorithm, replenishments can be totally or partially consumed. If the entire replenishment is consumed, the replenishment time is set to $R.time + S.Period$ (line 6 of BUDGET_CHECK). When only a portion of time is used, the replenishment must be split resulting in distinct replenishments. This is performed in the SPLIT_CHECK procedure. Only the used portion is given a new time, $T_s$ in the future (line 8). As these replenishments are maintained distinct and each usage is separated by at least $T_s$, the worst-case interference is correctly limited in the fashion consistent with the ideal model.

The number of replenishments to maintain over time can become very large. To help minimize this fragmentation the corrected algorithm allows merging of replenishments in accordance with Observation 3. This is done in lines 11-13 of the BUDGET_CHECK procedure. Here if the replenishment at the head of the queue overlaps with the next in the queue, they will be merged.

POSIX limits the number of replenishments into which the server capacity can be fragmented. The parameter *sched_ss_max_repl* defines the maximum number of replenishments that can be pending at any given time. There are very good pragmatic reasons for this limit. One is that it allows pre-allocation of memory resources required to keep track of replenishments. Another benefit is that it implicitly bounds the number of timer interrupts and context switches that replenishments can cause within the server period. This effect can be translated into the ideal model using Observation 2 as follows: When the pending replenishment limit is reached all server units with earliest-next-activation time prior to the next replenishment time are advanced to the next replenishment time, effectively becoming part of the next replenishment/cohort. This action is performed in the else block starting at line 6, of the SPLIT_CHECK procedure.

### 4.3 Improving Temporal Isolation

As explained in Section 3.3, the POSIX standard currently permits an interpretation that the sporadic server's background priority range is limited, and so there is no way to prevent an over-running sporadic server from starving tasks that are scheduled according to policies with priorities below this range.

To permit all other tasks on the system to be isolated from budget overruns of a sporadic server, we propose that the allowable range of priorities for a server's background priority should extend down to include the lowest system priority, and further include an extreme value so low that a thread with that value can never run. Alternatively one could introduce a new scheduling parameter or configurable system parameter to indicate that instead of switching to a background priority when a sporadic server is out of budget, it should wait until its next replenishment time. Adding either of these features would require very little change to an existing scheduler, and would provide at least one portable way to write applications with temporal isolation. If introducing such a new priority value or a new interface exceeds the latitude of the IEEE Standards interpretation process, the next best thing is to make it clear that implementations are permitted to define the range of priorities for SCHED_SPORADIC to extend below that of SCHED_FIFO.

To demonstrate the usefulness of priority ranges we extended our simulator to provide such functionality. This allowed us to implement SCHED_SPORADIC alongside a variety of other scheduling policies, including earliest-deadline-first (EDF) and deadline sporadic. We used a single range of 64-bit integer values to cover both static priorities and deadlines, reserving priority ranges at the extreme low and extreme high ends, and interpreting the largest expressible value as "do not execute". Of course, such an implementation model needs remapping of values at the API in order to comply with POSIX, which interprets numerically large values as higher fixed priorities, and requires a contiguous range of values for each policy.

## 5 Evaluation

This section presents our evaluation of the problems and the proposed solutions discussed above. This evaluation was performed using an implementation in the Linux-2.6.28 operating system and through simulation.

It is perhaps an indication of the seriousness of the budget amplification effect that we discovered it accidentally, as users. We were experimenting with a version of the Linux kernel that we had modified to support the SCHED_SPORADIC policy. The reason for using this policy was to bound the scheduling interference device driver threads cause other tasks [12]. Our implementation appeared to achieve the desired effect. We noticed that the server was consuming more execution time than its budget. We attributed these overruns to the coarseness of our sporadic server implementation, which enforced budget limits in whole ticks of the system clock. Since the clock tick was much larger than the network message interarrival and processing times, this allowed the execution behavior of the sporadic server under network device driver loads to be very similar to that of a periodic server, and it was able to run over significantly in each period. We hoped that by going to a finer-grained timer, we could both reduce the overrun effect and distinguish better between the sporadic and periodic servers. Therefore, we tried using a different Linux implementation of the sporadic server, developed by Dario Faggioli [7], which uses high-resolution timers. With a few refinements, we were able to repeat our prior experiments using this version, but the server continued to run significantly over its budget – sometimes nearly double its allocated CPU bandwidth. After first checking for errors in the time accounting, we analyzed the behavior again, and conjectured that the overruns might be due to budget amplification. To test this conjecture, we modified the scheduler to allow the currently available execution capacity to become negative, and observed the server CPU utilization drop down to the proper range.

As further verification, we conducted a simple structured experiment, using the Linux sporadic server implementation. A sporadic server is given a period of 10 msecs and a budget of 1 msec. Two jobs arrive, with execution times of one-half the budget and one-half the server period. The effect is to divide the budget into two replenishments. Immediately following the second job arrival, more jobs arrive, with the same execution times as the initial jobs, at a rate that maintains a server backlog for the duration of the experiment. The results are seen in the lower trace of Figure 7. Each replenishment, originally one-half the budget, is able to increase to the size of the full budget, allowing the server to achieve double its budgeted CPU utilization. The other traces in Figure 7 show what happens if the number of active replenishments before the start of the overload is

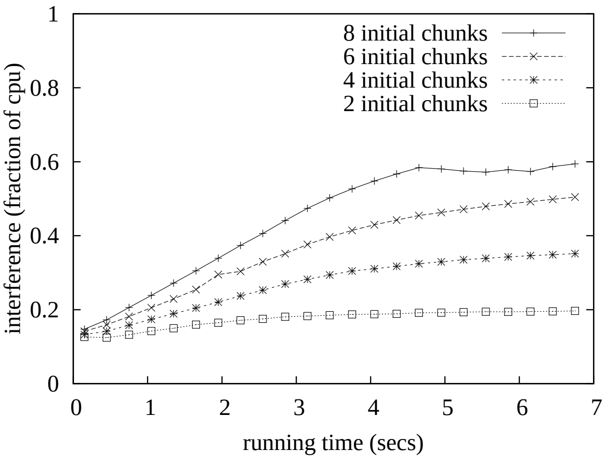


**Figure 7.** Budget amplification effect with varying number initial replenishments (empirical measurement).

4, 6, and 8. In principle, with sufficiently many initial fragments before the overload interval, the server CPU utilization could reach nearly 100%. However, in our experiment, the increase in server utilization did not climb so quickly, apparently due to the replenishments overlapping, causing merging of replenishments.

To further understand the sporadic server anomalies, a simulator was developed. With the simulator we were able to reduce the scheduling "noise" allowing us to focus on the problems associated with the POSIX definition of sporadic server rather than those introduced by the hardware and Linux kernel. Figures 8 and 9 are from this simulator.

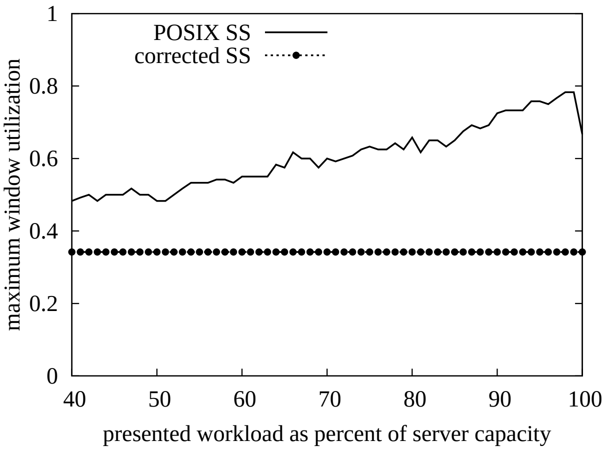


**Figure 8.** Utilization in a given period-sized window (simulation study).

The effects of budget amplification can not only increase the total utilization over an entire run, but also the maximum demand for execution time in any time window of a given size. (Here we consider "demand" to be the amount of time the server is allowed to compete at its foreground priority.) A correctly operating sporadic server should have the same worst-case demand as an equivalent periodic task.

So, if we consider a window of the server period in size, the maximum server demand in that window should not exceed the execution time divided by the period.

Due to the budget amplification effect, this is not true for sporadic server. Figure 8 shows the amount of execution time given to a sporadic server at its native priority (here the sporadic server is not allowed to run at background priority). This experiment was performed using an exponential distribution of job execution times with a mean job execution time of 10 time units. The server's period is 120 and the budget is 40. To demonstrate the budget amplification, there must be overruns. Here each job is permitted to overrun 1 time unit, corresponding to the resolution of the execution time clock as defined in the POSIX sporadic server. The interarrival times of jobs are also determined with an exponential distribution where the mean arrival rate is adjusted to create an average workload as a percent of server capacity. So, for a workload of 100% the mean interarrival time would be $\frac{120}{4} = 30$. The corrected sporadic server provides the expected maximum of 34% utilization in a given window ($\frac{40+1}{120}$). The POSIX implementation however exceeds the maximum utilization drastically, clearly not providing temporal isolation. (Over 100% of server capacity, it may be noticed that the maximum window utilization drops slightly, apparently due to more frequent overlapping and merging of replenishments.)

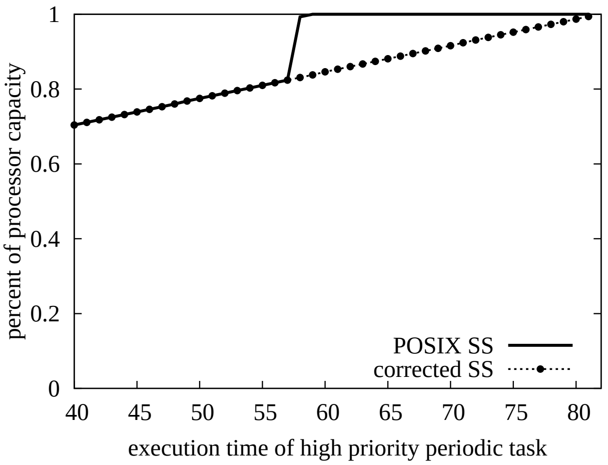


**Figure 9.** Effect of premature replenishments (simulation study).

To demonstrate the effect of premature replenishments Figure 9 graphs the combined capacity of the sporadic server and a higher priority periodic task. The periodic task has a period of 141 and execution time identified by the value along the x-axis. The sporadic server has a budget of 42 and a period of 100. The effect of the premature replenishment is not seen until the execution time of the high priority periodic task increases above 57 time units. At this point the effect hits abruptly, and the POSIX sporadic server is able to acquire 58 percent of the CPU. This is an increase of 38 percent from its budgeted 42 percent maximum and causes the CPU to become saturated. The corrected sporadic server is able to correctly limit the CPU utilization, thereby allowing other tasks to run, despite the server overload.

Attempts were made to demonstrate this premature replenishment effect on random arrivals and execution times, however, it appears that the effect does not occur often enough to be measured on a macroscopic scale. If, as this suggests, the premature replenishment anomaly has a very low probability, it may be that this anomaly would only be a concern in a hard real-time environment.

## 6 Related Work

Other works have proposed to improve the original sporadic server algorithm in [18], and at least one other proposed a correction for the error in the original sporadic server definition. However, to the best of our knowledge, this work is the first to point out and provide corrections for the defects in the POSIX sporadic server scheduling policy that are mentioned in this paper.

In [15], Jane Liu gives an example of a defect in the original sporadic server defined in [18], which allows a chunk of time to be replenished too soon. However, she does not discuss the POSIX sporadic server, which has different replenishment rules. In fact, the POSIX sporadic server handles her example correctly. Liu provides an alternate version of the sporadic server algorithm, that is conceptually similar to ours and that of [9] in maintaining the budget in separate chunks, each with their own replenishment times, and using the old replenishment time to compute the next replenishment time for the chunk. Liu's algorithm appears to always avoid premature replenishments, even on the example that we found caused trouble for the POSIX sporadic server. The differences between that algorithm and ours are in details that affect implementation overhead. Liu's algorithm applies a more aggressive and computationally more complex rule for computing replenishment times, based on keeping track of the starting times of busy intervals for each priority. It is less aggressive in attempting to merge chunks and so may require more storage and timer interrupts. It also fails to address the practical matter of budget overruns.

In [8], the importance of handling overruns is addressed. The authors propose a 'payback' mechanism that limits the amount of time a server can accumulate when its budget is exceeded.

In addition, the authors of [8] introduce an optimization to reduce the number of timer events by only arming a replenishment timer when the server has work to perform. We have observed that one can further reduce the number of timers events by not only checking if the server has work to perform, but also checking whether the server has budget. If the server has budget, then the need for a replenishment event is not necessary.

While they provide optimizations for the sporadic server algorithm, they do not address the defects we have mentioned in the current paper.

Davis and Burns [5] evaluated the use of periodic, deferrable, and sporadic servers in fixed priority pre-emptive systems. They provide response time analysis for real-time tasks scheduled under those servers. In a later paper, Davis and Burns [4] define a Hierarchical Stack Resource Policy (HSRP) for global resource access. That policy allows for server overruns and proposes to use the payback mechanism described in [9]. The approach in our paper also adapts the same overrun and payback mechanism.

## 7 Conclusion

We have shown that the POSIX formulation of the SCHED_SPORADIC scheduling policy suffers from several defects, making it inadequate for its intended purposes. If a critical system is trusted to meet deadlines, based on a schedulability analysis in which a SCHED_SPORADIC server is modeled as periodic server model, the consequences could be serious.

One possible reaction to the existence of these defects is to dismiss the POSIX SCHED_SPORADIC policy entirely. Some have argued that POSIX should be extended to include other fixed-task-priority budget-enforcing policies [1] that have lower implementation complexity. Others may argue that POSIX should be extended to include deadline-based scheduling policies, which potentially allow deadlines to be met at higher processor utilization levels.

We do not believe that SCHED_SPORADIC should be dismissed. There is a definite need for a standard scheduling policy that enforces time budgets, now. This capability is essential for the safe composition of applications in an open system. POSIX has no other such policy. The API for SCHED_SPORADIC exists, and with proper semantics can serve the originally intended purpose.

There is also a matter of time. The POSIX standard revision process is on a five-year cycle, and does not allow standardization of specifications that have not already been tested in existing practice. Therefore, the addition of any such new policies would be about five years off. In the mean time, there is an "interpretation" process for the existing standard that can be applied to correct the SCHED_SPORADIC specification sooner, perhaps within one year.

Therefore, we urge members of the real-time research and development community to support a corrective re-interpretation of the semantics of the POSIX SCHED_SPORADIC specification.

## Acknowledgment

We thank our anonymous reviewers for their invaluable insights. We also would like to thank Dario Faggioli for his Linux implementation of the POSIX SCHED_SPORADIC scheduling policy. This work is sponsored by NSF CNS-0509131. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, or the U.S. government.

## References

[1] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proc. 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[2] R. J. Bril and P. J. L. Cuijpers. Analysis of hierarchical fixed-priority pre-emptive scheduling revisited. Technical Report CSR-06-36, Technical University of Eindhoven, Eindhoven, Netherlands, 2006.

[3] J. Corbet. SCHED_FIFO and realtime throttling. `http://lwn.net/Articles/296419/`, Sept. 2008.

[4] R. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 257–270, Dec. 2006.

[5] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. 26th IEEE Real-Time Systems Symposium*, pages 376–385, 2005.

[6] Z. Deng and J. W. S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[7] D. Faggioli. POSIX SCHED_SPORADIC implementation for tasks and groups. `http://lkml.org/lkml/2008/8/11/161`, Aug. 2008.

[8] D. Faggioli, M. Bertogna, and F. Checconi. Sporadic server revisited. In *SAC '10: Proceedings of 25th ACM Symposium On Applied Computing*. ACM, Mar. 2010.

[9] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

[10] IEEE Portable Application Standards Committee (PASC). *Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifations, Issue 7*. IEEE, Dec. 2008.

[11] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[12] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A.-I. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, pages 57–68, Apr. 2007.

[13] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proc. 15th EuroMicro Conf. on Real-Time Systems*, pages 151–158, July 2003.

[14] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[15] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[16] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS '02: Proceedings of the 14th Euromicro Conf. on Real-Time Systems*, page 173, Washington, DC, USA, 2002. IEEE Computer Society.

[17] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–39, 2008.

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

[19] Y. C. Wang and K. J. Lin. The implementation of hierarchical schedulers in the RED-Linux scheduling framework. In *Proc. 12th EuroMicro Conf. on Real-Time Systems*, pages 231–238, June 2000.

[20] P. Zijlstra. sched: rt time limit. `http://lkml.org/lkml/2007/12/30/258`, Dec. 2007.