Experience with Sporadic Server Scheduling in Linux: Theory vs. Practice

Mark J. Stanovich, Theodore P. Baker*, An-I Andy Wang
Florida State University
Department of Computer Science, Florida, USA
{stanovic,baker,awang}@cs.fsu.edu

Abstract

Real-time aperiodic server algorithms were originally devised to schedule the execution of threads that serve a stream of jobs whose arrival and execution times are not known a priori, in a way that supports schedulability analysis. Well-known examples of such algorithms include the periodic polling server, deferrable server, sporadic server, and constant bandwidth server.

The primary goal of an aperiodic-server scheduling algorithm is to enforce a demand bound for each thread - that is, an upper bound on the amount of CPU time a thread may compete for in a given time interval. Bounding the demand of a given thread limits the interference that thread can inflict on other threads in the system experience in the competition for CPU time. Isolating the CPU-time demands of threads, known as temporal isolation, is an essential requirement for guaranteed resource reservations and compositional schedulability analysis in open real-time systems. A secondary goal of an aperiodic server is to minimize the worst-case and/or average response time while enforcing the demand bound. The theoretical aperiodic server algorithms meet both goals to varying degrees.

An implementation of an aperiodic server can yield performance significantly worse than its theoretical counterpart. Average response time is often higher, and even temporal isolation may not be enforced due to factors not found or considered in the theoretical algorithm. These factors include context-switching overheads, imprecise clocks and timers, preemption delays (e.g., overruns), and limits on storage available for bookkeeping.

This paper reports our experience implementing, in Linux, variations of the sporadic-server scheduling algorithm, originally proposed by Sprunt, Sha, and Lehoczky. We chose to work with sporadic-server scheduling because it fits into the traditional Unix priority model, and is the only scheduling policy recognized by the Unix/POSIX standard that enforces temporal isolation. While this paper only considers sporadic server, some lessons learned extend to other aperiodic servers including those based on deadline scheduling.

Through our experience, we show that an implemented sporadic server can perform worse than less complex aperiodic servers such as the polling server. In particular, we demonstrate the effects of an implementation's inability to divide CPU time into infinitely small slices and to use them with no overhead. We then propose and demonstrate techniques that bring the performance closer to that of the theoretical sporadic-server algorithm. Our solutions are guided by two objectives. The primary objective is that the server enforce an upper bound on the CPU time demanded. The secondary objective is that the server provide low average-case response time while adhering to the server's CPU demand bound. In order to meet these objectives, our solutions restrict the degree to which the server's total CPU demand can be divided. Additionally, we provide mechanisms to increase the server's ability to provide more continuous allocations of CPU demand.

Through a network packet service example, we show that sporadic server can be effectively used to bound CPU demand. Further, the efficiency of jobs served by sporadic server can be improved in terms of both reduced average-case response time and increased throughput.

*Dr. Baker's contributions to this paper are based on work supported by the National Science Foundation, while working at the Foundation.

1 Introduction

The roots of this paper are in experiments we did in 2007 on trying to schedule Linux device-driver execution in a way that conforms to an analyzable real-time scheduling model[3]. We found that the Unix *SCHED_SPORADIC* scheduling policy is a potential improvement over *SCHED_FIFO* at any constant scheduling priority. Then, in subsequent studies we discovered that we needed to correct some technical defects in the POSIX definition of *SCHED_SPORADIC*, which are reported in [5]. The paper describes our more recent efforts to deal with another phenomenon, having to do with preemption overhead and trade-offs between server throughput, server response time, and the ability to guarantee deadlines of other real-time tasks.

In a broader sense, this paper is about narrowing a gap that has developed between real-time operating systems and real-time scheduling theory. While a great deal is known about real-time scheduling in theory, very little of the theory can be applied in current operating systems. We feel that closer integration of operating systems implementation and scheduling theory is needed to reach a point where one can build open systems that reliably meet real-time requirements.

After some review of real-time scheduling theory and aperiodic servers, we discuss our experiences with implementing sporadic server scheduling, the problem of properly handling preemption overhead, and how we addressed the problem. We compare the performance of serving aperiodic workload by a polling server, a sporadic server, and a hybrid polling-and-sporadic server, using our implementation of the three scheduling algorithms in Linux. We conclude with a brief discussion of lessons learned and some further work.

2 Background

Any implementor of real-time operating systems needs to understand the basics of real-time scheduling theory, in order to understand the implications of implementation decisions. While this paper is not primarily about scheduling theory, we try to establish some theoretical background as motivation for our discussion of implementation issues.

Real-time scheduling theory provides analysis techniques that can be used to design a system to meet timing constraints. The analyses are based on abstract scheduling algorithms and formal models of workload and processing resources. The theory can guarantee that a set of timing constraints will always be satisfied, but *only* if an actual system conforms to the abstract models on which the analysis is based.

Real-time operating systems provide a run-time platform for real-time applications, including the mechanisms and services that schedule the execution of tasks on the processing unit(s). For timing guarantees based on real-time scheduling theory to apply to an application implemented using an operating system, there must be a close correspondence between the virtual execution platform provided by the OS and the abstract models and scheduling algorithms of the theory. The burden of achieving this correspondence falls on the OS and application developers.

The OS must provide mechanisms that allow development of applications that conform to the abstract models of the theory within bounded tolerances. In the case of a general-purpose operating system that supports the concept of open systems, such as Linux, the OS must go further, to provide firewall-like mechanisms that preserve conformance to the models when independently developed applications or components run alongside one another.

In real-time scheduling theory the arrival of a request for some amount of work is known as a *job*, and a logical stream of jobs is called a *task*. Some jobs have deadlines. The goal is to find a way to schedule all jobs in a way that one can prove that *hard* deadlines will always be met, *soft* deadlines will be met within a tolerance by some measure, and all tasks are able to make some progress at some known rate. To succeed, the theory must make some assumptions about the underlying computer platform and about the *workload*, *i.e.* the times at which jobs may arrive and how long it takes to execute them.

The best-behaved and best understood task model is a *periodic* task, whose jobs have a known *worst-case execution time* (WCET) and a known fixed separation between every pair of consecutive jobs, called the *period*. A periodic task also has an associated *deadline*, the point in time, relative to the arrival of a job, by which the job must complete execution. These workload parameters, along with others, can be used to determine whether all jobs can meet their timing constraints if executed according to certain scheduling algorithms, including strict preemptive fixed-task priority scheduling.

A key concept in the analysis of preemptive scheduling is *interference*. The nominal WCET of a job is based on the assumption that it is able to run to completion (*i.e.*, until the corresponding thread

suspends itself) without interference from jobs of any other task. Showing that a job can complete within a given time window in the presence of other tasks amounts to bounding the amount of processor time the other tasks can steal from it over that interval, and then showing that this *worst-case interference* leaves enough time for the job to complete. The usual form of interference is preemption by a higher priority task. However, lower priority tasks can also cause interference, which is called *priority inversion* or *preemption delay*. Preemption delays may be caused by critical sections, imprecision in the OS timer mechanism, or any other failure of the kernel to adhere consistently to the preemptive fixed-priority scheduling model.

A system that supports the UNIX real-time API permits construction of threads that behave like a periodic task. The *clock_nanosleep()* function is one of several that provide a mechanism for suspending execution between one period and the next. Using the *sched_setscheduler()* function the application can request the *SCHED_FIFO* policy, and assign a priority. By doing this for a collection of periodic tasks, and choosing priorities sufficiently high to preempt all other threads,[1] one should be able to develop an application that conforms closely enough to the model of periodic tasks and fixed task-priority preemptive scheduling to guarantee the actual tasks meet deadlines within some bounded tolerance.

Unfortunately, that is not enough. To support a reasonable range of real-time applications one needs to be able to handle a wider range of tasks. For example, a task may request CPU time periodically but the execution time requested may not be bounded, or the arrival of work may not be periodic. If such a task has high enough priority, the interference it can cause for other tasks may be unpredictable or even unbounded, causing other tasks to miss deadlines.

Aperiodic tasks typically have performance requirements that are soft, meaning that if there is a deadline it is stochastic, or occasional deadline misses can be tolerated, or under temporary overload conditions load shedding may be acceptable. So, while the CPU time allocated to the service of aperiodic tasks should be bounded to bound worst-case interference for other tasks, it should be provided in a way that allows the aperiodic task to achieve fast average response time under expected normal circumstances.

One example of an aperiodic task that requires fast average response time can be found in the paper by Lewandowski, et. al [3]. In this paper, a real-time task uses the network in its time-critical path to gather information. While it is desirable to receive all network packets, missing a few packets is not catastrophic. The difficulty lies in that the network receive path is shared by other tasks on the system, some with different deadlines and others with no explicit deadlines.

Assuming a fixed-task-priority model, a priority must be chosen for the bottom level of network packet service. Processing the packets at a low or background priority does not work well because processing the packets may be delayed arbitrarily. Extended delay in network packet processing means that a real-time task waiting for the packets may miss an unacceptably large number of packets. Another option is to schedule the network packet processing at a high priority. However, the network packet processing now can take an unbounded amount of CPU time, potentially starving other tasks on the system and thereby causing missed deadlines. Therefore, a scheduling scheme is needed that provides some high-priority time to serve the aperiodic jobs; however, the high-priority time should be limited, preventing the packet processing from monopolizing the CPU. The bound on CPU time ensures other tasks have access to the CPU in a timely manner.

The key to extending analysis techniques developed for periodic tasks to this broader class of workloads is to ration processor time. It must be possible to force even an uncooperative thread to be scheduled in a way that the worst-case interference it causes other tasks can be modeled by the worst-case behavior of some periodic task. A number of scheduling algorithms that accomplish this have been studied, which we refer to collectively as *aperiodic servers*.

Examples of well-known aperiodic server scheduling algorithms for use in a fixed-task-priority scheduling environment include the *polling* and *deferrable* servers [18], and the *sporadic server* [2]. There are also several examples for use with deadline scheduling, among which the *constant bandwidth server* has received considerable attention[17].

All these algorithms bound the amount of CPU time an aperiodic task receives in any time interval, which bounds the amount of interference it can cause other tasks, guaranteeing the other tasks are left a predictable minimum supply of CPU time. That is, aperiodic servers actively enforce *temporal isolation*, which is essential for an open real-time execution platform.

The importance of aperiodic servers extends beyond the scheduling of aperiodic tasks. Even

[1] Of course, careful attention must be given to other details, such as handling critical sections.

the scheduling of periodic tasks may benefit from the temporal isolation property.[2] Aperiodic server scheduling algorithms have been the basis for a rather extensive body of work on open real-time systems, appearing sometimes under the names virtual processor, hierarchical, or compositional scheduling. For example, see [4, 9, 10, 11, 12, 13, 14].

In this paper, we limit attention to a fixed-task-priority scheduling environment, with particular attention to sporadic- server scheduling. The primary reason is that Linux for the most part adheres to the UNIX standard and therefore supports fixed-task-priority scheduling. Among the well-known fixed-task-priority aperiodic-server scheduling algorithms, sporadic-server scheduling is theoretically the best. It also happens to be the only form of aperiodic-server scheduling that is recognized in the UNIX standard.

A *polling server* is one way of scheduling aperiodic workloads. The polling server is a natural extension to the execution pattern of a periodic task. Using a polling server, queued jobs are provided CPU time based on the polling server's budget, which is replenished periodically. If no work is available when the polling server is given its periodic allocation of CPU time, the server immediately loses its budget. Similarly, if the budget is partially used, and no jobs are queued, the polling server gives up the remainder of the budget.[3]

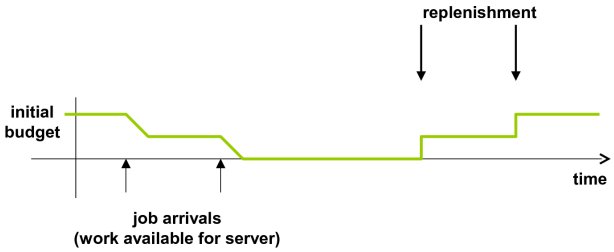

FIGURE 1: *Example usage and replenishment of sporadic server's budget.*

A *sporadic server* is a thread that is scheduled according to one of the variants of the original sporadic server algorithm introduced by Sprunt, Sha, and Lehoczky [2]. While many variants exist, the basic idea is the same. A sporadic server has a *budget*, *replenishment period*, and *scheduling priority*. When the sporadic server uses the CPU, the amount of time used is deducted from its budget. The amount of CPU time consumed is restored to the budget one replenishment period in the future, starting from the instant when the sporadic server requested CPU time and had budget. The operation to restore the budget at a given time in the future, based on the amount of time consumed, is known as a *replenishment*. Once the server uses all of its budget, it can no longer compete for CPU time at its scheduling priority.[4]

The objective of the sporadic-server scheduling algorithm is to limit worst-case system behavior such that the server's operation can be modeled, for schedulability analysis of other tasks, as if it were a periodic task. That is, in any given sliding time window, the sporadic server will not demand more CPU time than could be demanded by a periodic task with the same period and budget. A secondary goal of the sporadic server is to provide fast average response time for its jobs.

With regard to minimizing average response time, a sporadic server generally outperforms a polling server. The advantage with a sporadic server is that jobs can often be served immediately upon arrival, whereas with a polling server jobs will generally have to wait until the next period to receive CPU time. Imagine a job arrival that happens immediately after the polling server's period. The job must wait until the following period to begin service, since the polling server immediately forfeits its budget if there are no jobs available to execute. A sporadic server, on the other hand, can execute the job immediately given that its budget can be retained when the server's queue is empty. The ability to retain budget allows the server to execute more than once during its period, serving multiple jobs as they arrive. Aperiodic servers that can hold on to their budget until needed are known as *bandwidth-preserving* servers.

3 Implementation

Several variants of the original sporadic-server algorithm have been proposed, including the POSIX *SCHED_SPORADIC* [7], and more recently two variants that correct defects in the POSIX version [5, 8]. Differences include how they handle implementation constraints such as limited space to store replenishment operations, overruns, and preemption costs.

[2] Even nominally periodic tasks may be subject to faults that cause them to over-run their predicted WCET.

[3] A polling server cannot be implemented as a *SCHED_FIFO* periodic task, because there is no enforcement of the processor time budget.

[4] This does not describe the original Sporadic Server algorithm completely, nor does it address a subtle defect in the original algorithm which was corrected in subsequent work. Further, there are many sporadic-server variants, each with their own nuances. These details are omitted to simplify the discussion.

The scheduling algorithm followed by our implementation is described in [15], which is an updated version of [5] including corrections for errors in the pseudo-code that were identified by Danish *et al.* in [4].

Correct operation of a sporadic server results in bounded interference experienced by lower-priority tasks. In order to measure the interference, we used Regehr's "hourglass" technique [6], which creates an application-level process that monitors its own execution time without requiring special operating system support. The hourglass process infers the times of its transitions between executing and not executing by reading the clock in a tight loop. If the time between two successive clock values is small, the assumption is that the process was not preempted. However, if the difference is large, the thread was likely preempted. This technique can be used to find preemption points and thereby determine the time intervals when the hourglass process executed. From this information, the hourglass process can calculate its total execution time.

Using the hourglass approach, we were able to evaluate whether an implemented sporadic server actually provides temporal isolation. That is, if we schedule the hourglass task with a priority below that of the sporadic server (assuming there are no other higher-priority tasks in the system), the hourglass task should be able to consume all of the CPU time that remains after the sporadic server used all of its budgeted high-priority time. The CPU time available to the hourglass task should, *ideally*, be one hundred percent minus the percentage budgeted for the sporadic server, viewed over a large enough time window. Therefore, if we schedule a sporadic server with a budget of 1 millisecond and a period of 10 milliseconds, and there are no other tasks with priority above the sporadic server and hourglass tasks, the hourglass task should be able to consume at least 90% of the CPU time, *i.e.*, 9 milliseconds in any window of size 10 milliseconds. In reality, other activities such as interrupt handlers may cause the interference experienced by the hourglass task to be slightly higher.

To evaluate the response time characteristics of our sporadic server, we measured the response time of datagram packets sent across a network. The response time of a packet is measured by the time difference between sending the packet on one machine, m_1, and receiving the packet by another, m_2. More specifically, the data portion of each packet sent from m_1 contains a timestamp, which is then subtracted from the time the packet is received by the UDP layer on m_2.[5] In our setup, m_1 periodically sends packets to m_2. The time between sending packets is varied in order to increase the load experienced by the network receive thread on m_2. The receive thread on m_2 is scheduled using either the polling server, sporadic server, or *SCHED_FIFO* [7] scheduling policies.[6] In our experiments, m_2 is running Linux 2.6.38 with a ported version of softirq threading found in the 2.6.33 Linux real-time patch. m_2 has a Pentium D 830 processor running at 3GHz with a 2x16KB L1 cache and a 2x1MB L2 cache. 2GB of RAM are installed. The kernel was configured to use only one core, so all data gathered is basically equivalent to a uniprocessor system.

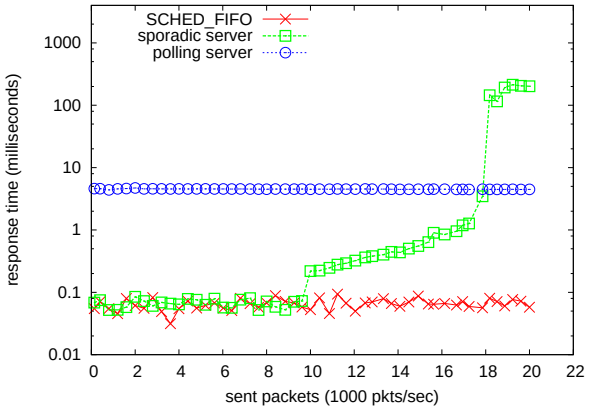

FIGURE 2: *Response time using different scheduling policies.*

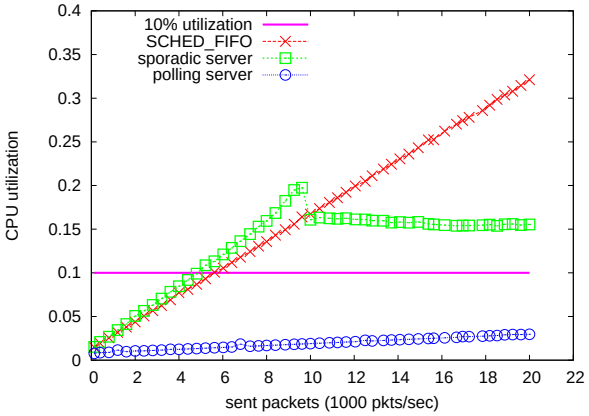

FIGURE 3: *sirq-net-rx thread CPU utilization using different scheduling policies.*

[5] The clocks for the timestamps on m_1 and m_2 are specially synchronized using a dedicated serial connection.

[6] *SCHED_FIFO* differs from the other in allowing thread of sufficiently high priority to execute arbitrarily long without preemption.

Scheduling the Linux network receive thread (i.e., *sirq-net-rx*) using various scheduling policies affects the average response time of received network packets. One would expect that the polling server would result in higher average response times than *SCHED_FIFO* or sporadic server and that sporadic server and *SCHED_FIFO* should provide similar average response times until sporadic server runs out of budget.

In our experiment, sporadic server and polling server are both given a budget of 1 millisecond and a period equal to 10 milliseconds. The sporadic server's maximum number of replenishments is set to 100. The hourglass task is scheduled using *SCHED_FIFO* scheduling at a real-time priority lower than the priority of the network receive thread. Each data point is averaged over a 10 second interval of sending packets at varied rates. The CPU utilization and response time for the described experiment are shown in Figures 2 and 3.

One would expect that if the sporadic server and polling server both were budgeted 10% of the CPU, the lower-priority hourglass task should be able to consume at least 90% of the CPU time regardless of the load. However, the data for the experiment shows that the sporadic server is causing much greater than 10% interference. The additional interference is the consequence of preemptions caused by the server. Each time a packet arrives the sporadic server preempts the hourglass task, thereby causing two context switches for each packet arrival. Given that the processing time for a packet is small (2-10 microseconds) the server will suspend itself before the next packet arrives. In this situation, the aggregate time for context switching and other sporadic server overhead such as using additional timer events and running the sporadic-sever-related accounting becomes significant. For instance, on the receiving machine the context-switch time alone was measured at 5-6 microseconds using the *lat_ctx* LMbench program[1].

The overhead associated with preemption causes the additional interference that is measured by the lower-priority hourglass task.[7]

A snapshot of CPU execution time over a 500 microsecond time interval was produced using the Linux Trace Toolkit(LTTng)[16] and is shown in Figure 4. The top bar is the *sirq-net-rx* thread and the bottom bar is the lower-priority hourglass measuring task. This figure shows that the CPU time of both tasks is being finely sliced. The small time slices cause interference for both the lower-priority and sporadic server thread that would not be experienced if the threads were able to run to completion.

3.1 Accounting for Preemption Overhead

To ensure that no hard deadlines are missed, and even to ensure that soft deadlines are met within the desired tolerances, CPU time interference due to preemptions must be included in the system's schedulability analysis. The preemption interference caused by a periodic task can be included in the analysis by adding a preemption term to the task's worst-case execution time ($WCET$) that is equal to twice the worst-case context switch cost – one for switching into the task and one for switching out of the task.[8] Assuming all tasks on the system are periodic, this is at least a coarse way of including context-switch time in the schedulability analysis.

A sporadic server can cause many more context switches than a periodic task with the same parameters. Rather than always running to completion, a sporadic server has the ability to self-suspend its execution. Therefore, to obtain a safe $WCET$ bound for analysis of interference[9], one would have to determine the maximum number of contiguous "chunks" of CPU time the sporadic server could request within any given period-sized time interval. The definition of sporadic- server scheduling given in scheduling theory publications does not place any such restriction on the number of CPU demand chunks and thus imposes no real bound on the $WCET$. In order to bound the number of preemptions, and thereby bound the time spent context switching, most implemented variations of sporadic server limit the maximum number of pending replenishments, denoted by max_repl. Once max_repl replenishments are pending, a sporadic server will be prevented from executing until one of the future replenishments arrives.

[7] The lower-priority thread does not measure much of the cache eviction and reloading that other applications may experience, because its code is very small and typically remains in the CPU's cache. When cache effects are taken into account, the potential interference penalty for each preemption by a server is even larger.

[8] This is an intentional simplification. The preemption term should include all interferences caused by the sporadic server preempting another thread, not only the direct context-switch time, but also interferences such as the worst-case penalty imposed by cache eviction and reloading following the switch. For checking the deadline of a task, both "to" and "from" context switches need to be included for potentially preempting task, but only the "to" switch needs be included for the task itself.

[9] From this point on we abuse the term $WCET$ to stand for the maximum interference that a task can cause for lower-priority tasks, which includes not just the maximum time that the task itself can execute, but also indirect costs, such as preemption overheads.

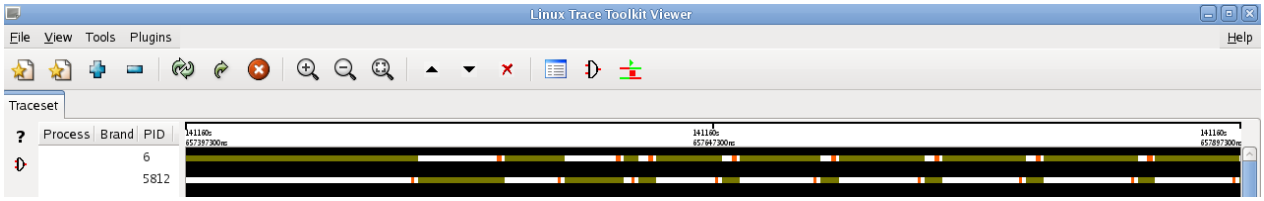

FIGURE 4: *LTTng visualization of CPU execution.*

Using *max_repl*, the maximum context-switching time per period of a sporadic server is two times the *max_repl*. Using this logic, and assuming that the actual context-switch costs are added on top of the servers budget, a worst-case upper bound on the interference that can be caused by a sporadic server task could be written as:

$$SS_{budget} + 2 * max\_repl$$

Accounting for the cost due to preemptions is important in order to ensure system schedulability; however, adding preemption cost on top of the server's budget as above results in over-provisioning. That is, if a sporadic server does not use *max_repl* number of replenishments in a given period a worst-case interference bound derived in this way is an over-estimate. At the extreme, when a sporadic server consumes CPU time equal to its budget in one continuous chunk, the interference only includes the cost for two context switches rather than two times *max_repl*. However, the server cannot make use of this windfall to execute jobs in its queue because the context switch cost was not added to its actual budget.

We believe a better approach is to account for actual context-switch costs while the server is executing, charging context switch costs caused by the server against its actual budget, and doing so only when it actually preempts another task. In this approach the SS_{budget} alone is used as the interference bound for lower-priority tasks. Accounting for context-switching overhead is performed on-line by deducting an estimate of the preemption cost from the server's budget whenever the server causes a preemption. Charging the sporadic server for preemption overhead on-line reduces over-provisioning, and need not hurt server performance on the average, although it can reduce the effective worst-case throughput of the server if the workload arrives as many tiny jobs (as in our packet service experiment).

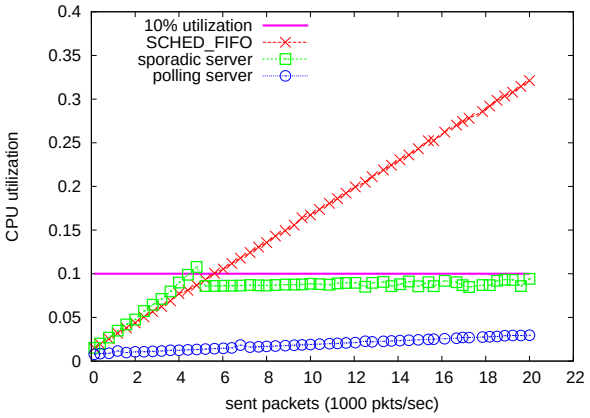

FIGURE 5: *Accounting for context-switching overhead.*

Charging for preemptions on-line requires that the preemption interference be known. Determining an appropriate amount to charge the server for preempting can be very difficult, as it depends on many factors. In order to determine an amount to charge sporadic server for a preemption, we ran the network processing experiment under a very heavy load and extracted an amount that consistently bounded the interference of sporadic server to under 10%. While such empirical estimation may not be the ideal way to determine the preemption interference, it gave us a reasonable value to verify that charging for preemptions can bound the interference.

The network experiment was performed again, this time charging sporadic server a toll of 10 microseconds each time it caused a preemption. Figure 5 shows the results for the experiment and demonstrates that time interference for other lower-priority tasks can be bounded to 10%, that is, the server's budget divided by the period.

3.2 Preemption Overhead

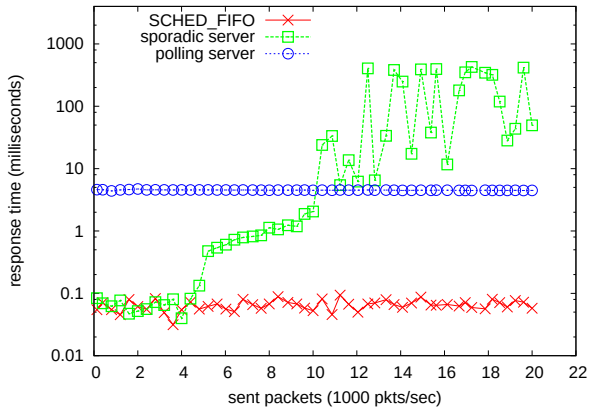

FIGURE 6: *Accounting for context-switching overhead.*

Bounding the interference that an aperiodic workload causes for other tasks is the primary objective of aperiodic server scheduling; however, one would also like to see fast average response time. Figure 6 shows that under heavy load, the average response time of packets when using sporadic-server scheduling is actually worse than that of a polling server with the same parameters. For this experiment, not only is the sporadic server's average response time higher, but as the load increases up to 45% of the packets were dropped.[10]

The poor performance of the sporadic server is due to a significant portion of its budget being consumed to account for preemption costs, leaving a smaller budget to process packets. If all of the packets arrived at the same time, the processing would be batched and context switching would not occur nearly as often. However, due to the spacing between packet arrivals, a large number of preemptions occur. A polling server on the other hand has a much larger portion of its budget applied to processing packets, and therefore does not drop packets and also decreases the average response time.

Based on the poor performance of a sporadic server on such workloads one might naïvely jump to the conclusion that, in general, a polling server is a much better choice. Actually, there is a trade-off, in which each form of scheduling has its advantage. Given the same budget and period, a sporadic server will provide much better average-case response time under light load, or even under a moderate load of large jobs, but can perform worse than the polling server for certain kinds of heavy or bursty workloads.

[10]No packets were dropped by the other servers.

It turns out that the workload presented by our packet service example is a bad one for the sporadic server, in that a burst of packet arrivals can fragment the server budget, and then this fragmentation becomes "locked in" until the backlog is worked off. Suppose a burst of packets arrives, and the first *max_repl* packets are separated by just enough time for the server to preempt the running task, forward the packet to the protocol stack, and resume the preempted task. The server's budget is fragmented into *max_repl* tiny chunks. Subsequent packets are buffered (or missed, if the device's buffer overflows), until the server's period passes and the replenishments are added back to its budget. Since there is by now a large backlog of work, the server uses up each of its replenishment chunks as it comes due, then suspends itself until the next chunk comes due. This results in a repetition of the same pattern until the backlog caused by the burst of packets has been worked off. During this overload period, the sporadic server is wasting a large fraction of its budget in preemption overhead, reducing its effective bandwidth below that of a polling server with the same budget and period. There is no corresponding improvement in average response time, since after the initial *max_repl* fragmentation, the reduced bandwidth will cause the response times to get worse and worse.

3.3 Reducing the Impact of Preemption Overhead

A hybrid server combining the strengths of polling and sporadic servers may be a better alternative than choosing either one. In this approach, a sporadic server is used to serve light loads and a polling server to serve heavy loads.

Sporadic-server scheduling supports a polling-like mode of operation. When the *max_repl* parameter value is one, only one preemption is permitted per period. Switching to the polling-like mode of operation is just a matter of adjusting *max_repl* to 1.

When changing modes of operation of the sporadic server in the direction of reducing *max_repl*, something must be done if the current number of pending replenishments would exceed *max_repl*. One approach is to allow the number of pending replenishments to exceed *max_repl* temporarily, reducing it by one each time a replenishment comes due. Another approach is to implement the reduction at once, by coalescing pending replenishments. This is

similar to the classical mode-change scheduling problem, in that one must be careful not to violate the assumptions of the schedulability analysis during the transition. In the case of a sporadic server the constraint is that the server cannot cause any more interference within any time window than would be caused by a periodic task with execution time equal the server budget and period equal to the server's budget replenishment period, including whatever adjustments have been made to the model to allow for context-switch effects. We call this the *sliding window constraint* for short.

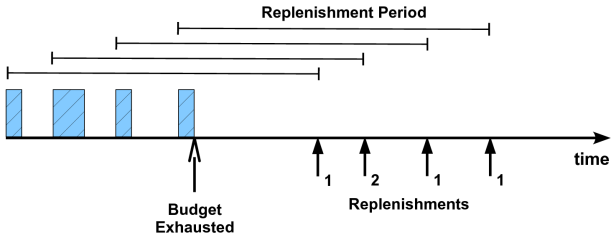

FIGURE 7: *Sporadic server with max_repl ≥ 4, before switch to polling-like server.*

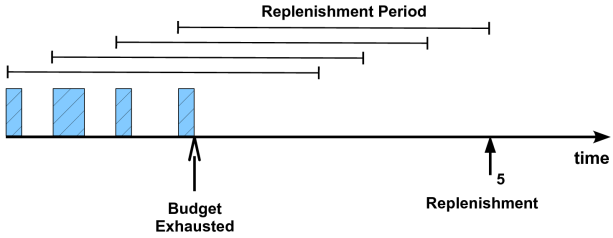

FIGURE 8: *After switch to poll-like server, with max_repl = 1 and replenishments coalesced.*

In order to maintain the sliding-window constraint during the mode change, one can think in terms of changing the times associated with pending replenishments. Consolidating the replenishment times would allow the creation of a single replenishment with an amount equal to the server's initial budget. To guard against violating the sliding-window constraint, the replenishment time of any replenishment must not be moved earlier in time. One approach is to coalesce all replenishments into the replenishment with a time furthest in the future, resulting into a single replenishment with an amount equal to the server's initial budget as shown in Figures 7 and 8.

Switching from sporadic server to a polling-like server should be performed if the server is experiencing heavy load. The ideal switching point may be difficult to detect. For instance, a short burst may be incorrectly identified as the onset of a heavy load and the early switching may cause the server to postpone a portion of its budget that could have been used sooner. Conversely, delaying the switch may mean that time that could have been used to serve incoming jobs is wasted on preemption charges.

While an ideal switching point may not be possible to detect beforehand, one reasonable indicator of a heavy load is when sporadic server uses all of its budget. That is the point when a sporadic server is blocked from competing for CPU time at its scheduling priority. At this point the server could switch to its polling-like mode of operation.

A possible event to indicate when to switch back to the sporadic server mode of operation is when a sporadic server blocks but still has available budget. This point in time would be considered as entering a period of light load and the *max_repl* could be reinstated.

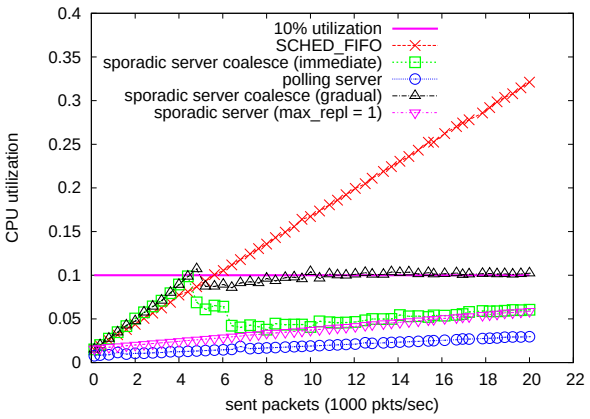

FIGURE 9: *Coalescing replenishments under heavy load.*

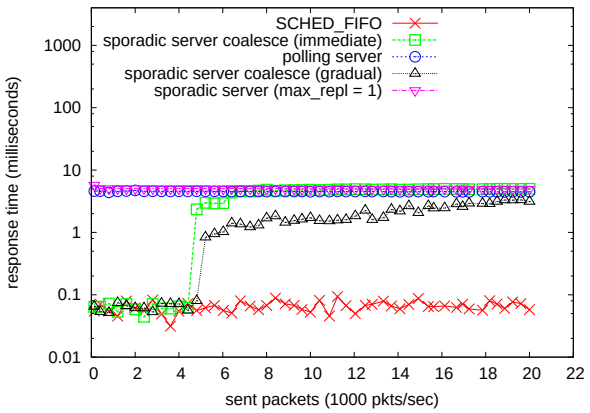

FIGURE 10: *Coalescing replenishments under heavy load.*

Implementation of the switching mechanism described above is relatively simple. The replenishments are coalesced into one when the server runs out of budget but still has work. The single replenishment limit will remain enforced until the sporadic server is suspended and has budget, a point in time considered to be an indication of light load. So, the polling-like mode of operation will naturally transition back to the original sporadic server mode of operation.

Immediately coalescing all replenishments may be too eager. Loads that are between light and heavy may experience occasional or slight overloads that require only slightly more CPU time. In this case, converting all potential preemption charges, by delaying replenishments, into CPU time to serve packets is too extreme. Therefore, to perform better under a range of loads one approach is to coalesce only two replenishments for each overload detection. Using this method allows the sporadic server to naturally find an intermediate number of replenishments to serve packets efficiently without wasting large portions of its budget on preemption charges.

The performance data for the two coalescing methods, immediate and gradual, are shown in Figures 9 and 10. These figures show the advantage of transitioning between sporadic-server and polling-like mode of operation. Under light load until approximately 4500 pkts/sec, the sporadic server has the same response time as *SCHED_FIFO* scheduling. However, once the load is heavy enough the sporadic server is forced to limit the amount of CPU demand and therefore the response time begins to increase to that of a polling server. There is not enough CPU budget to maintain the low average response time with *SCHED_FIFO* scheduling. The difference between the immediate and gradual coalescing is seen when the restriction on CPU demand begins. The gradual coalescing provides a gradual transition to polling-like behavior where as the immediate coalescing has a much faster transition to the polling server's response time performance. The better performance of the gradual coalescing is due to the server making better use of the available budget. With immediate coalescing, when the server transitions to the polling-like mode the CPU utilization drops, as one would expect of sporadic server where the *max_repl* is set to 1. However, with gradual coalescing the server continues to use its available budget to pay for preemption costs and serve some jobs earlier, which results in lower response times.

4 Conclusion

Any open real-time operating system needs to provide some form of aperiodic-server scheduling policy, in order to permit temporal isolation of tasks, and to provide a real-time virtual processor abstraction that can support fault-tolerant compositional schedulability analysis. The only standard Unix scheduling policy with these properties is Sporadic Server. [11]

We have described our experiences implementing and using a variation of the Sporadic Server scheduling algorithm in Linux. Our experience demonstrates that sporadic server scheduling can be an effective way to provide a predictable quality of service for aperiodic jobs while bounding the interference that the server thread can cause other tasks, thereby supporting schedulability analysis. However, this goal cannot be achieved without consideration of some subtle implementation issues that are not addressed in the theoretical formulations of sporadic server that have been published.

Neither the published theoretical versions of Sporadic Server nor the POSIX/Unix formulation consider all the interference effects we found on a real implementation. In particular, fine grained time slicing degrades the performance of the sporadic server thread and can cause interference for other threads on the system to significantly exceed the assumptions of the theoretical model. This interference is mainly due to a sporadic server being able to use CPU time in arbitrarily small time slices. Such fine time slicing not only increases the interference that the server inflicts on tasks that it preempts, but also degrades the throughput of the server itself. Through network service experiments we showed that the interference caused by a sporadic server can be significant enough to cause other real-time threads to miss their deadlines.

Charging a sporadic server for preemptions is an effective means to limit the CPU interference. The charging for preemptions can be carried out in several ways. We chose an on-line approach where the server is charged when it preempts another thread. Charging the server only when it actually preempts not only bounds the CPU time for other tasks, but allows the server to use its budget more effectively. That is, rather than accounting for the additional interference by inflating the nominal server budget (over the implemented server budget) in the schedulability analysis, we charge the server at run time for the actual number of preemptions it causes. In

[11] While a literal implementation of the abstract description of *SCHED_SPORADIC* in the Unix standard is not practical and would not support schedulability analysis, we feel that the corrections described in this paper and [15] fall within the range of discretion over details that should be permitted to an implementor.

this way the server's actual interference is limited to its actual CPU time budget, and we do not need to use an inflated value in the schedulability analysis. Since the preemption charges come out of the server's budget, we still need to consider preemption costs when we estimate the worst-case response time of the server itself. However, if we choose to over-provision the server for worst-case (finely fragmented) arrival patterns it actually gets the time and can use it to improve performance when work arrives in larger chunks.

The ability to use small time slices allows a sporadic server to achieve low average response times under light loads. However, under a load of many small jobs, a sporadic server can fragment its CPU time and waste a large fraction of its budget on preemption charges. A polling server, on the other hand, does not experience this fragmentation effect, but does not perform as well as sporadic server under light load. To combine the strengths of both servers, we described a mechanism to transition a sporadic server into a polling-like mode, thereby allowing sporadic server to serve light loads with good response time and serve heavy loads with throughput similar to a polling server. The data for our experiments show that the hybrid approach performs well on both light and heavy loads.

Our recent experiences reinforce what we learned in prior work with sporadic-server scheduling in Linux [5]. There are devils in the details when it comes to reducing a clever-looking theoretical algorithm to a practical implementation. To produce a final implementation that actually supports schedulability analysis, one must experiment with a real implementation, reflect on any mismatches between the theoretical model and reality, and then make further refinements to the implemented scheduling algorithm until there is a match that preserves the analysis. This sort of interplay between theory and practice pays off in improved performance and timing predictability.

We also believe our experience suggests a potential improvement to the "NAPI" strategy employed in Linux network device drivers for avoiding unnecessary packet-arrival interrupts. NAPI leaves the interrupt disabled so long as packets are being served, re-enabling it only when the network input buffer is empty. This can be beneficial if the network device is faster than the CPU, but in the ongoing race between processors and network devices the speed advantage shifts one way and another. For our experimental set-up, the processor was sufficiently fast that it was able to handle the interrupt and the *sirq-net-rx* processing for each packet before the next arrived, but the preemption overhead for doing this was still a problem. By waiting for several packets to arrive, and then processing them in a batch, the polling server and our hybrid server were able to handle the same workload with much less overhead. However, the logical next step is to force a similar waiting interval on the interrupt handler for the network device.

While we have not experimented with deadline-based aperiodic servers in Linux, it appears that our observations regarding the problem of fitting the handling of context switch overheads to an analyzable theoretical model should also apply to the constant bandwidth server, and that a similar hybrid approach is likely to pay off there.

In future work, we hope to explore additional variations on our approach to achieving a hybrid between polling and sporadic server, to see if we can improve performance under a range of variable workloads. We are considering several different mechanisms, including stochastic, for detecting when we should change modes of operation as the system moves between intervals of lighter and heavier load. We also plan to explore other aperiodic servers and determine how much interference preemptions cause. For example, it appears that a constant bandwidth server would suffer the same performance problems as a sporadic server when the workload causes budget fragmentation. We also plan to investigate the preemption interference due to cache eviction and reloading. The threads used in our experiments access relatively small amounts of data and therefore do not experience very large cache interferences. This is not true for all applications, and the cache effects on such applications will need to be bounded. While limiting the number of replenishments does reduce the cache effect, better mechanisms are needed to reduce the ability of sporadic server to cause cache interferences.

Other questions we are considering include whether it is practically feasible to schedule multiple threads using a single sporadic-server budget, and how well sporadic-server scheduling performs on a multi-core system with thread migration.

References

[1] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, Jan. 1996.

[2] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic

task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[3] 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.

[4] M. Danish, Y. Li, and R. West. Virtual-cpu scheduling in the quest operating system. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:169–179, 2011.

[5] M. Stanovich, T. P. Baker, A.-I. A. Wang, and M. G. Harbour. Defects of the posix sporadic server and how to correct them. In *Real Time and Embedded Technology and Applications Symposium, 2010. RTAS '10. 16th IEEE*, pages 35–45, Stockholm, Sweden, Apr. 2010. IEEE Computer Society.

[6] J. Regehr. Inferring scheduling behavior with Hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, Monterey, CA, June 2002.

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

[8] D. Faggioli, M. Bertogna, and F. Checconi. Sporadic server revisited. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 340–345, Sierre, Switzerland, 2010. ACM.

[9] 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.

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

[11] 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.

[12] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierar hical 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.

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

[14] 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.

[15] M. Stanovich, T. P. Baker, A.-I. A. Wang, and M. G. Harbour. Defects of the posix sporadic server and how to correct them. Technical Report TR-091026 (revised), Florida State University Department of Computer Science, *http://www.cs.fsu.edu/research/reports/TR-100315.pdf*

[16] Linux Trace Toolkit Next Generation, *http://lttng.org/*

[17] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, Florence, Italy, June 1999.

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