Technical Report
CMU/SEI-90-TR-6
ESD-90-TR-207

Implementing Sporadic Servers in Ada

Brinkley Sprunt
Lui Sha

May 1990

Technical Report
CMU/SEI-90-TR-6
ESD-90-TR-207
May 1990

Implementing Sporadic Servers in Ada

Brinkley Sprunt
Lui Sha

Department of Electrical and Computer Engineering
Software Engineering Institute

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler
SEI Joint Program Office

Implementing Sporadic Servers in Ada

Abstract. The purpose of this paper is to present the data structures and algorithms for implementing sporadic servers [13] in real-time systems programmed in Ada. The sporadic server algorithm is an extension of the rate monotonic scheduling algorithm [6]. Sporadic servers are tasks created to provide limited and usually high-priority service for other tasks, especially aperiodic tasks. Sporadic servers can be used to guarantee deadlines for hard-deadline aperiodic tasks and provide substantial improvements in average response times for soft-deadline aperiodic tasks over polling techniques. Sporadic servers also provide a mechanism for implementing the Period Transformation technique [9] that can guarantee that a critical set of periodic tasks will always meet their deadlines during a transient overload. Sporadic servers can also aid fault detection and containment in a real-time system by limiting the maximum execution time consumed by a task and detecting attempts to exceed a specified limit. This paper discusses two types of implementations for the sporadic server algorithm: (1) a partial implementation using an Ada task that requires no modifications to the Ada runtime system and (2) a full implementation within the Ada runtime system. The overhead due to the runtime sporadic server implementation and options for reducing this overhead are discussed. The interaction of sporadic servers and the priority ceiling protocol [10] is also defined.

1. Introduction

The purpose of this paper is to present two high-level designs, in the form of data structures and algorithms, for implementing sporadic servers [13] in real-time systems programmed in Ada. The first design presented is a partial implementation of sporadic servers using an application-level Ada task. This implementation requires no runtime system modifications. The second sporadic server design is a full implementation of the sporadic server algorithm within an Ada runtime system. The Real-Time Scheduling in Ada project at the Software Engineering Institute (SEI) has designed a prototype implementation of the full sporadic server algorithm using a commercially available Ada runtime system. This technical report summarizes our implementation experiences to date and should be of interest to both Ada runtime implementors and real-time Ada application developers.

The sporadic server algorithm, developed by the Advanced Real-Time Technology project at Carnegie Mellon University, was designed as an extension of the rate monotonic algorithm for periodic tasks [6] to provide general support for both soft- and hard-deadline aperiodic tasks. In addition to providing a general scheduling solutions for aperiodic tasks, sporadic servers can be used for other real-time scheduling problems. Sporadic servers provide a mechanism for implementing the Period Transformation technique [9] for guaranteeing that a critical set of periodic tasks will always meet their deadlines during a transient overload. Producer/Consumer problems in real-time systems can be handled in a straightforward manner using sporadic servers. Sporadic servers can also aid fault detection and containment in a real-time system by limiting the execution time consumed by a task to a maximum value and detecting attempts to exceed the specified limit.

1.1. Background

It is common practice to place a real-time task into one of four categories based upon its deadline and its arrival pattern. If meeting a given task's deadline is critical to the system's operation, then the task's deadline is considered to be *hard*. If a quick response for a real-time task is desirable, but not absolutely necessary for correct system operation, then the task's deadline is considered to be *soft*. Tasks in real-time systems that have no timing constraints, such as data logging and backup, are classified as *background* tasks. Tasks with regular arrival times are *periodic* tasks. Periodic tasks are commonly used to process sensor data and update the current state of the real-time system on a regular basis. Periodic tasks used in control and signal processing applications typically have hard deadlines. Tasks with irregular arrival times are *aperiodic* tasks. Aperiodic tasks are used to handle the processing requirements of events with nondeterministic request patterns, such as operator requests. Aperiodic tasks typically have soft deadlines, but some aperiodic tasks can have hard deadlines. Aperiodic tasks with hard deadlines are *sporadic* tasks. In summary, we have:

- **Hard-Deadline Periodic Tasks.** A periodic task consists of a sequence of requests arriving at regular intervals. A periodic task's deadline coincides with the end of its period.
- **Soft-Deadline Aperiodic Tasks.** An aperiodic task consists of a stream of requests arriving at irregular intervals. Soft deadline aperiodic tasks typically require a fast average response time.
- **Sporadic Tasks.** A sporadic task is an aperiodic task with a hard deadline and a *minimum* interarrival time (the amount of time between two requests) [7].
- **Background Tasks.** A background task has no timing requirements and no particular arrival pattern. Background tasks are typically assigned the lowest priority in the system and, as such, the scheduling of background tasks will not be considered in this paper.

A well understood algorithm for scheduling hard-deadline periodic tasks is Liu and Layland's rate monotonic scheduling algorithm [6]. The rate monotonic algorithm assigns fixed priorities to tasks based upon the rate of their requests (i.e., a task with a relatively short period is given a relatively high priority). Under the assumptions of negligible context switching overhead and independent tasks (i.e., tasks that require no synchronization with one another), Liu and Layland proved that this algorithm is the optimum, fixed-priority pre-emptive scheduling algorithm for periodic tasks with hard deadlines. The rate monotonic algorithm provides real-time system designers with a well defined algorithm for determining *a priori* the timing correctness of the system, a quality that potentially offers a great reduction in the costs of system development, testing, and maintenance. The rate monotonic algorithm has the advantage of low scheduling overhead since priorities are statically assigned. The rate monotonic algorithm has also been shown to have a high average schedulable utilization bound of 88% [4], and has been extended to handle transient overloads [9], periodic tasks that share resources [10], and multiprocessors [3, 2, 8].

To handle soft-deadline aperiodic tasks, Lehoczky, Sha, and Strosnider created the the *deferrable server* (DS) and *priority exchange* (PE) algorithms [5]. These algorithms are extensions of the basic rate monotonic algorithm and they operate by creating a pool of high priority utilization that can be shared by soft-deadline aperiodic tasks. These algorithms have been shown to greatly improve the average response time performance of soft-deadline aperiodic tasks over polling and background service techniques [15]. However, Sprunt, Sha, and Lehoczky have shown that these algorithms have some limitations [14]. The PE algorithm was shown to require a prohibitively complex implementation. The schedulability bound for periodic tasks was shown to be lower when using the DS algorithm than the PE algorithm. In other words, for a given periodic task set, the maximum server size (the ratio of the server's maximum execution time to the server's period) for the DS algorithm is typically smaller than the maximum PE server size. It was also shown that although these algorithms can be used to provide support for some sporadic tasks, no technique had been developed for guaranteeing the deadline for a sporadic task that is shorter than the sporadic task's minimum interarrival time.

The Sporadic Server (SS) algorithm [14] was developed to overcome these limitations of previous aperiodic server algorithms. Specifically, the SS algorithm was originally developed to meet the following goals:

- provide good responsiveness for soft-deadline aperiodic tasks and guaranteed deadlines for sporadic tasks while not compromising the timing constraints of any hard-deadline periodic tasks
- attain a high degree of schedulable utilization
- allow for low implementation complexity and low runtime overhead
- provide a scheduling framework for real-time systems that are primarily composed of aperiodic tasks

Briefly, the specification and operation of a sporadic server is as follows (a complete description of the SS algorithm is presented in Section 2). A sporadic server is specified by its period, execution time, and priority. During system operation, a sporadic server preserves its execution time until one of the tasks it is servicing becomes ready to execute, at which point the sporadic server uses its available execution time to service the task. Service can continue as long as the sporadic server has capacity available. Once the sporadic server's execution time is exhausted, sporadic service is suspended until the consumed execution time is replenished. In its simplest form, the sporadic server replenishes consumed execution time one sporadic server period after the execution time is initially consumed.

Although the SS algorithm was developed for servicing soft and hard-deadline aperiodic tasks, sporadic servers can be also used to implement solutions for several other real-time scheduling problems. The major uses of sporadic servers are:

- **Improving Average Response Times for Soft-Deadline Aperiodic Tasks.** A high priority sporadic server can be created to service a set of soft-deadline aperiodic tasks. Since the sporadic server preserves its high priority execution time until it is needed, the sporadic server can provide immediate service for aperiodic tasks as long as it has available execution time. This is a great improvement over polling or background service techniques in which aperiodic tasks are either serviced at regular intervals or whenever no other tasks are being serviced. Unlike polling or background service, a sporadic server can usually provide service "on demand," and therefore, provide a much better average response time.

- **Guaranteeing Response Times for Sporadic Tasks.** To guarantee the response time for a sporadic task, a sporadic server can be created to provide exclusive service to the sporadic task. The sporadic server's period and execution time are set equal to the minimum interarrival time and worst case execution time, respectively, of the sporadic task. In this manner, the sporadic server will always have execution time available to service the sporadic task, even when the sporadic task arrives at its maximum rate. Usually the deadline of the sporadic task is equal to or greater than its minimum interarrival time and a rate monotonic priority is assigned to the sporadic server (i.e., the priority is based upon the minimum interarrival time of the sporadic task). For the cases when the sporadic task's deadline is shorter than its minimum interarrival time, the priority of the sporadic server must be based upon the deadline of the sporadic task, not its minimum interarrival time (i.e., a deadline monotonic priority assignment should be used). The necessity of a deadline monotonic priority assignment and its associated schedulability analysis is discussed in [14].

- **Scheduling Producer/Consumer Tasks.** In a real-time system a typical producer/consumer scheduling problem occurs when a device can produce items at a burst rate that is faster than the average rate at which those items can be consumed. Since the burst production rate is greater than the average consumption rate, items are placed in a queue as they are produced. The consumer removes items from the queue as soon as possible. The bursty production rate is characterized by an event density, which is the maximum number of events that can occur during any interval of time of a specified duration.

  Two sporadic servers can be used to schedule the producer and consumer tasks. The execution time of the sporadic server servicing the producer is set equal to the maximum number of items that can arrive in a burst, multiplied by the time required to queue each item. The execution time of the sporadic server servicing the consumer is set equal to the the maximum number of items that can arrive in a burst multiplied by the time required to consume each item. The period of both sporadic servers is set equal to duration of time used to define the event density. The priority of the consumer sporadic server is based upon its period (i.e., it is assigned its rate monotonic priority). However, the priority of the producer sporadic server is based up the minimum interarrival time of items. Thus, the producer sporadic server is given a priority high enough to ensure that no items are lost and the consumer sporadic server has a priority sufficient to guarantee that items are consumed at a quick enough rate to prevent the queue from overflowing. With the characterization of the bursty arrivals (event density) and the specification of the producer and consumer sporadic servers, one can then bound the maximum queue length and the maximum time to consume any item.

- **Implementing the Period Transformation Technique.** Transient overloads occur when stochastic execution times for periodic tasks lead to a desired utilization greater than the schedulable utilization bound of the task set. Under the rate monotonic algorithm, periodic task priorities are assigned based upon their rate, not necessarily upon task importance. Thus, an important task may be assigned a low priority and, as such, may miss its deadline under transient overload conditions. For these cases, the period transformation technique can be used to guarantee that a set of critical periodic tasks will still meet their deadlines during a transient overload [9]. The basic idea of this technique is to force a critical task to be subdivided into tasks with smaller execution times that are executed in sequential order at a higher rate. The higher execution rate will give the task a high enough priority to allow it to execute even during transient overloads.

  A sporadic server can be used to implement the period transformation technique. The execution time and period of the sporadic server are set equal to the execution budget and period of the transformed periodic task. Since the sporadic server will suspend service once its available execution time is exhausted, the desired execution pattern for the periodic task is obtained. Note that this approach requires no special modifications to the code for the periodic task.

- **Fault Detection and Containment.** A hardware fault or programming bug can cause a task to execute longer than has been allowed for in the schedulability analysis. Typically, the only mechanism used to detect these types of faults checks if the task has not been completed by its deadline. However, this allows the task not only to exhaust its own budget of execution time but also to consume part of the execution time budgets of other lower priority tasks, possibly causing them to miss their deadlines. By servicing a periodic task using a sporadic server with an execution time, period, and priority identical to that of the periodic task, the periodic task will be restricted to consuming at most its execution time budget. If the sporadic server's budget is ever completely exhausted and the periodic task is still ready to execute, the sporadic server has detected a fault. Since the sporadic server will suspend the execution of the periodic task once all its execution time has been consumed, the errors resulting from any fault that has caused the periodic task execute longer than it should are contained to the faulty task and not allowed to influence the execution of other tasks.

Note that the use of sporadic servers to implement the period transformation technique or to improve fault detection and containment requires that the full sporadic server algorithm be implemented in the Ada runtime system. This is necessary because only within the runtime system can sporadic service be suspended once the server's execution time has been exhausted, regardless of whether or not the task being serviced has completed execution. For sporadic servers implemented at the application level (see Section 3), sporadic service can only be suspended once the task being serviced has completed execution.

The next section gives a complete description of the SS algorithm using examples for soft-deadline aperiodic tasks and discusses the interaction of the SS algorithm and the priority ceiling protocol [10] for scheduling periodic tasks that share data. Section 3 describes an application level implementation of the SS algorithm as an Ada task that requires no modifications to Ada or its runtime system. Section 4 presents the data structures and algorithms necessary for a full Ada runtime implementation of sporadic servers. Section 6 presents a summary.

2. The Sporadic Server Algorithm

The SS algorithm creates a high priority task for servicing aperiodic tasks. The SS algorithm preserves its server execution time at its high priority level until an aperiodic request occurs. The SS algorithm replenishes its server execution time after some or all of the execution time is consumed by aperiodic task execution. This method of replenishing server execution time sets the SS algorithm apart from the previous aperiodic server algorithms [5, 14] and is central to understanding the operation of the SS algorithm.

The following terms are used to explain the SS algorithm's method of replenishing server execution time:

$\mathbf{P}_S$	Represents the task priority level at which the system is currently executing.
$\mathbf{P}_i$	One of the priority levels in the system. Priority levels are consecutively

numbered in priority order with $\mathbf{P}_1$ being the highest priority level, $\mathbf{P}_2$ being the next highest, and so on.

Active	This term is used to describe a priority level. A priority level, $\mathbf{P}_i$, is considered to be *active* if the current priority of the system, $\mathbf{P}_S$, is equal to or higher than the priority of $\mathbf{P}_i$.
Idle	This term has the opposite meaning of the term *active*. A priority level, $\mathbf{P}_i$, is *idle* if the current priority of the system, $\mathbf{P}_S$, is lower than the priority of $\mathbf{P}_i$.
$\mathbf{RT}_i$	Represents the replenishment time for priority level $\mathbf{P}_i$. This is the time at which consumed execution time for the sporadic server of priority level $\mathbf{P}_i$ will be replenished. Whenever the replenishment time, $\mathbf{RT}_i$, is set, it is set equal to the current time plus the period of $\mathbf{P}_i$.

Determining the schedule for replenishing consumed sporadic server execution time consists of two separate operations: (1) determining the time at which any consumed execution time can be replenished and (2) determining the amount of execution time (if any) that should be replenished. Once both of these operations have been performed, the replenishment can be scheduled. The time at which these operations are performed depends upon the available execution time of the sporadic server and upon the active/idle status of the sporadic server's priority level. The rules for these two operations are stated below for a sporadic server executing at priority level $\mathbf{P}_i$:

1. If the server has execution time available, the replenishment time, $\mathbf{RT}_i$, is set when priority level $\mathbf{P}_i$ becomes active. Otherwise, the server capacity has been exhausted and $\mathbf{RT}_i$ cannot be set until the server's capacity becomes greater than zero and $\mathbf{P}_i$ is active. In either case, the value of $\mathbf{RT}_i$ is set equal to the current time plus the period of $\mathbf{P}_i$.
2. The amount of execution time to be replenished can be determined when either the priority level of the sporadic server, $\mathbf{P}_i$, becomes idle or when the sporadic server's available execution time has been exhausted. The amount to be replenished at $\mathbf{RT}_i$ is equal to the amount of server execution time consumed since the last time at which the status of $\mathbf{P}_i$ changed from idle to active.

2.1. SS Algorithm Examples

The operation of the SS algorithm will be demonstrated with four examples: a high priority sporadic server, an equal priority sporadic server (i.e., a sporadic server with a priority that is equal to the priority of another task), a medium priority sporadic server, and an exhausted sporadic server. Figures 1-4 present the operation of the SS algorithm for each of these examples. The upper part of these figures shows the task execution order and the lower part shows the sporadic server's capacity as a function of time. Unless otherwise noted, the periodic tasks in each of these figures begin execution at time = 0.

Figure 1 shows task execution and the task set characteristics for the high priority sporadic server example. In this example, two aperiodic requests occur. Both requests require 1 unit of execution time. The first request occurs at t = 1 and the second occurs at t = 8. Since the sporadic server is the only task executing at priority level $\mathbf{P}_1$ (the highest priority level), $\mathbf{P}_1$ becomes active only when the sporadic server services an aperiodic task. Similarly, whenever the sporadic server is not servicing an aperiodic task, $\mathbf{P}_1$ is idle. Therefore, $\mathbf{RT}_1$ is set whenever an aperiodic task is serviced by the sporadic server. Replenishment of consumed sporadic server execution time will occur one server period after the sporadic server initially services an aperiodic task.

The task execution in Figure 1 proceeds as follows. For this example the sporadic server begins with its full execution time capacity. At $t = 0$, τ_1 begins execution. At time = 1, the first aperiodic request occurs and is serviced by the sporadic server. Priority level $\mathbf{P}_1$ has become active and $\mathbf{RT}_1$ is set to $1 + 5 = 6$. At $t = 2$, the servicing of the first aperiodic request is completed, exhausting the server's execution time, and $\mathbf{P}_1$ becomes idle. A replenishment of 1 unit of execution time is set for $t = 6$ (note the arrow in Figure 1 pointing from $t = 1$ on the task execution timeline to $t = 6$ on the server capacity timeline). The response time of the first aperiodic request is 1 unit of time. At $t = 3$, τ_1 completes execution and τ_2 begins execution. At $t = 6$, the first replenishment of server execution time occurs, bringing the server's capacity up to 1 unit of time. At $t = 8$, the second aperiodic request occurs and $\mathbf{P}_1$ becomes active as the aperiodic request is serviced using the sporadic server's execution time. $\mathbf{RT}_1$ is set equal to 13. At $t = 9$, the servicing of the second aperiodic request completes, $\mathbf{P}_1$ becomes idle, and τ_2 is resumed. A replenishment of 1 unit of time is set for $t = 13$ (note the arrow in Figure 1 pointing from $t = 8$ on the task execution timeline to $t = 13$ on the server capacity timeline). At $t = 13$, the second replenishment of server execution time occurs, bringing the server's capacity back up to 1 unit of time.

Figure 2 shows the task execution and the task set characteristics for the equal priority sporadic server example. As in the previous example, two aperiodic requests occur and each requires 1 unit of execution time. The first aperiodic request occurs at $t = 1$ and the second occurs at $t = 8$. The sporadic server and τ_1 both execute at priority level $\mathbf{P}_1$ and τ_2 executes at priority level $\mathbf{P}_2$. At $t = 0$, τ_1 begins execution, $\mathbf{P}_1$ becomes active, and $\mathbf{RT}_1$ is set to 10. At $t = 1$, the first aperiodic request occurs and is serviced by the sporadic server. At $t = 2$, service is completed for the first aperiodic request and τ_1 resumes execution. At $t = 3$, τ_1 completes execution and τ_2 begins execution. At this point, $\mathbf{P}_1$ becomes idle and a replenishment of 1 unit of server execution time is set for $t = 10$. At $t = 8$, the second aperiodic request occurs and is serviced using the sporadic server, $\mathbf{P}_1$ becomes active, and $\mathbf{RT}_1$ is set to 18. At $t = 9$, service is completed for the second aperiodic request, τ_2 resumes execution, $\mathbf{P}_1$ becomes idle, and a replenishment of 1 unit of server execution time is set for $t = 18$. At $t = 10$, τ_1 begins execution and causes $\mathbf{P}_1$ to become active and the value of $\mathbf{RT}_1$ to be set. However, when τ_1 completes at $t = 12$ and $\mathbf{P}_1$ becomes idle, no sporadic server execution time has been consumed. Therefore, no replenishment time is scheduled even though the priority level of the sporadic server became active.

Figure 2 illustrates two important properties of the sporadic server algorithm. First, $\mathbf{RT}_i$ can be determined from a time that is *earlier* than the request time of an aperiodic task. This occurs for the first aperiodic request in Figure 2 and is allowed because $\mathbf{P}_1$ became active before *and* remained active until the aperiodic request occurred. Second, the amount of execution time replenished to the sporadic server is equal to the amount consumed.

Figure 3 shows the task execution and the task set characteristics for the medium priority

Task	Exec Time	Period	Utilization
SS	1	5	20.0%
τ_1	2	10	20.0%
τ_2	6	14	42.9%

Figure 1: High Priority Sporadic Server Example

sporadic server example. In this example, two aperiodic requests occur and each requires 1 unit of execution time. The first request occurs at $t = 4.5$ and the second at $t = 8$. The sporadic server executes at priority level $\mathbf{P}_2$, between the priority levels of τ_1 ($\mathbf{P}_1$) and τ_2 ($\mathbf{P}_3$). At $t = 0$, τ_1 begins execution. At $t = 1$, τ_1 completes execution and τ_2 begins execution. At $t = 0$, $\mathbf{RT}_2$ is set to 10 but, since no sporadic server execution time is consumed before $\mathbf{P}_2$ becomes idle at $t = 1$, no replenishment is scheduled. At $t = 4.5$, the first aperiodic request occurs and is serviced using the sporadic server making priority level $\mathbf{P}_2$ active. At $t = 5$, τ_1 becomes active and pre-empts the sporadic server. At this point all priority levels are active since $\mathbf{P}_1$ is active. At $t = 6$, τ_1 completes execution, $\mathbf{P}_1$ becomes idle, and the sporadic server is resumed. At $t = 6.5$, service for the first aperiodic request is completed, τ_2 resumes execution, and $\mathbf{P}_2$ becomes idle. A replenishment of 1 unit of sporadic server execution time is scheduled for $t = 14.5$. At $t = 8$, the second aperiodic request occurs and consumes 1 unit of sporadic server execution time. A replenishment of 1 unit of sporadic server execution time is set for $t = 18$.

Figure 3 illustrates another important property of the sporadic server algorithm. Even if the sporadic server is pre-empted and provides discontinuous service for an aperiodic request (as occurs with the first aperiodic request in Figure 3), only one replenishment is necessary. Pre-emption of the sporadic server does not cause the priority level of the sporadic server to become idle, thus allowing several separate consumptions of sporadic server execution time to be

Task	Exec Time	Period	Utilization
SS	2	10	20.0%
τ_1	2	10	20.0%
τ_2	6	14	42.9%

Figure 2: Equal Priority Sporadic Server Example

replenished together. Note that one replenishment for the consumption of sporadic server execution time resulting from both aperiodic requests in Figure 3 is not permitted because the priority level of the sporadic server became idle between the completion of the first aperiodic request and the initial service of the second aperiodic request.

The final sporadic server example, presented in Figure 4, illustrates the application of the replenishment rules stated in Section 2 for a case when the sporadic server's execution time is exhausted. Figure 4 shows that even though the sporadic server's priority level may be active before the sporadic server actually begins servicing an aperiodic request, the replenishment time must be determined from the time at which the sporadic server's capacity becomes greater than zero. In Figure 4, the sporadic server has a priority less than periodic task τ_1 and greater than periodic task τ_2. The initial period for τ_1 begins at time = 2 and the initial period for τ_2 begins at t = 0.

Task execution in Figure 4 proceeds as follows. At t = 0, τ_2 becomes ready and begins execution. At t = 1, an aperiodic request occurs that requires 3 units of execution time. The sporadic server pre-empts τ_2 and begins servicing the aperiodic request. While the aperiodic request is being serviced τ_1 becomes ready at t = 2 and pre-empts the sporadic server. At t = 3, τ_1 completes execution and servicing of the aperiodic request continues. At t = 4, the sporadic

Task	Exec Time	Period	Utilization
τ_1	1.0	5	20.0%
SS	2.5	10	25.0%
τ_2	6.0	14	42.9%

Figure 3: Medium Priority Sporadic Server Example

server exhausts its execution time capacity and τ_2 resumes execution. A replenishment of 2 units of sporadic server execution time is scheduled for t = 11. At t = 6, τ_1 pre-empts τ_2 and executes until t = 7 when τ_2 resumes execution. At t = 10, τ_1 again pre-empts τ_2 and begins execution. Note that at t = 10 (as was also the case for t = 6), all priority levels become active because the highest priority task is now executing. At t = 11, τ_1 completes execution and the replenishment of 2 units of sporadic server execution time occurs allowing the servicing of the aperiodic request to continue. The aperiodic request is completed at t = 12 and τ_2 resumes execution. A second replenishment for consumed sporadic server execution time must now be scheduled. However, the replenishment time is not determined from t = 10, the point at which the sporadic server's priority level became active, because at t = 10 the sporadic server's capacity was zero. The replenishment time is instead determined from the t = 11, the point at which the sporadic server's capacity became greater than zero.

2.2. Sporadic Servers and Priority Inheritance Protocols

In this section we define the interaction of sporadic servers and the priority inheritance protocols developed by Sha, Rajkumar, and Lehoczky in [10]. The schedulability impact of servicing aperiodic tasks that share data using priority inheritance protocols is also discussed.

Mok [7] has shown that the problem of determining the schedulability of a set of periodic tasks

Task	Exec Time	Period	Utilization
τ_1	1	4	25%
SS	2	10	20%
τ_2	10	40	25%

Figure 4: Exhausted Sporadic Server Replenishment

that use semaphores to enforce exclusive access to shared resources is NP-hard. The semaphores are used to guard critical sections of code (e.g., code to insert an element into a shared linked list). To address this problem for rate monotonic scheduling, Sha, Rajkumar, and Lehoczky [10] have developed priority inheritance protocols and derived sufficient conditions under which a set of periodic tasks that share resources using these protocols can be scheduled. The priority inheritance protocols require that the priority of a periodic task be temporarily increased if it is holding a shared resource that is needed by a higher priority task. Since both sporadic servers and the priority inheritance protocols manipulate the priorities of tasks, it is necessary to define the interaction of these two scheduling techniques.

The priority inheritance protocols manipulate the priorities of tasks that enforce mutual exclusion using semaphores in the following manner.[1] Consider the case of a task, τ_A, that is currently executing and wants to lock a semaphore and enter a critical section. The priority inheritance protocols will select one of the following two sequences of task execution:

1. Task τ_A is allowed to lock the semaphore and enter the critical section. During the critical section τ_A executes at its assigned priority.

[1]The description here of the operation of the priority inheritance protocols is very simplistic but sufficient for describing the interaction of sporadic servers and the priority inheritance protocols. For a better description of the priority inheritance protocols, the reader is referred to [10].

2. Task τ_A is not allowed to lock the semaphore and is blocked from executing. The lower priority task, τ_L, that is causing the blocking then inherits the priority of τ_A and continues execution. The lower priority task executes until the lock is released and then τ_A gets the lock and executes.

We are concerned with the problem of the interaction of priority inheritance protocols and a sporadic server for the case when an aperiodic task that is using its sporadic server wants to lock a semaphore and enter a critical section. The interactions to be defined concern the inheritance of the sporadic server's priority and the consumption of sporadic server execution time. In order to preserve the benefits of the priority inheritance protocols it is necessary to retain its rules of operation without modification. Thus, a lower priority task that is blocking an aperiodic task from entering its critical section inherits the priority of the aperiodic task's sporadic server. However, two possibilities exist for the consumption of sporadic server execution time:

1. Allow the task that inherits the sporadic server's priority to consume the sporadic server's execution time.
2. Do not allow the task that inherits the sporadic server's priority to consume the sporadic server's execution time.

The policy selection affects the efficiency and complexity of sporadic server implementations.

A comparison of the implementation effects of these two policy choices shows that the first policy results in a more complex implementation that requires more overhead to manage sporadic server execution time. The first policy requires that the implementation maintain more state to manage the sporadic server's execution time. With the first policy, any task that can block the execution of the aperiodic task can then consume the execution time of the aperiodic task's sporadic server. This expands the execution time of potential users of the sporadic server beyond the set of aperiodic tasks associated with the sporadic server, and this makes the conditional tests in the implementation more complex and less efficient. The first policy choice would also require that the implementation handle the case when the sporadic server's execution time is exhausted by a task that has inherited the priority of the sporadic server. In this case, all tasks that have inherited the sporadic server's priority must return to the priority they had before inheriting the sporadic server's priority. Changing these priorities can be a complex operation especially when nested critical sections are involved. Also, once these priority changes have been made it may be necessary to re-evaluate the priority inheritance protocols because the priority of one or more tasks has changed during the middle of a critical section.

The better choice for the policy governing the consumption of sporadic server execution time is not to allow tasks that inherit the sporadic server's priority to consume the sporadic server's execution time. This allows the implementation of support for sporadic servers to be largely independent of the implementation of support for the priority inheritance protocols. The resulting independence of the sporadic server and priority inheritance protocol implementations avoids the problems associated with the first policy choice.

Now that the interaction of sporadic servers and the priority inheritance protocols has been defined, we need to discuss the schedulability impact of using a sporadic server to service an aperiodic task that shares data with a periodic task. To describe the schedulability impact we will use two examples, each describing the schedulability of one periodic task and one aperiodic task. The periodic and aperiodic tasks share data using the priority ceiling protocol developed by Sha, Rajkumar, and Lehoczky [10]. In the first example, the aperiodic task executes at a priority lower than the periodic task. In the second example, a high-priority sporadic server is created to service the aperiodic tasks.

To demonstrate the schedulability impact of the sporadic server we will use the following equation developed in [10]:

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1). \tag{1}$$

where C_i and T_i are respectively the execution time and period of task τ_i and B_i is the worst-case blocking time for task τ_i. This equation was derived using the worst case utilization bound equation for scheduling periodic tasks developed by Liu and Layland in [6] which, under the absolute worst case conditions, provides a sufficient condition for determining schedulability of a rate monotonic priority assignment.

For our examples, the blocking term B_i will be used to represent the maximum amount of time that the aperiodic task can block the execution of the periodic task, due to the possibility that the aperiodic task may have already obtained exclusive access to the shared data when the periodic task makes its request for the shared data. Note that "blocking time" is different from "pre-emption time". Blocking occurs when a lower priority task blocks the execution of a higher priority task. Pre-emption occurs when a higher priority task prevents the execution of a lower priority task. The above equation provides a sufficient test to determine if the sum of the pre-emption time, blocking time, and execution time for each task is less than its deadline. The examples will show that the addition of a high-priority sporadic server to service the aperiodic task increases the pre-emption time imposed upon the periodic task and does *not* decrease the amount of blocking time possible for the periodic task (unless special provisions are made as described later).

For the examples, we assume the following:

- The operations to be performed by either the periodic or aperiodic task upon the shared data take, at most, 2 units of time.
- The periodic task has a maximum execution time of 8 units ($C_P = 8$) and a period of 20 units ($T_P = 20$). This maximum execution time includes the time to operate upon the shared data, assuming no blocking occurs. The periodic task is the only task with a hard deadline.
- The execution time and arrival pattern of the aperiodic task are not important for these examples. However, whenever the aperiodic task executes it requires access to the shared data and, once access is obtained, the aperiodic task may block the periodic task from executing.
- The sporadic server (used only in the second example) has an execution time of 3 units ($C_{SS} = 3$) and a period of 10 units ($T_{SS} = 10$).

The first example is composed of the periodic task executing at a higher priority than the aperiodic task. The schedulability criterion for the periodic task using equation 1 is shown below.

$$\frac{C_P}{T_P} + \frac{B_P}{T_P} \le 1(2^1-1)$$
$$\frac{8}{20} + \frac{2}{20} \le 1$$
$$\frac{10}{20} \le 1$$

As can be seen from the above evaluation, the periodic task can be guaranteed to meet its deadline. This evaluation can be more simply described by noting that the maximum interval from the initiation of the periodic task to its completion consists of the its maximum execution time of 8 units plus the maximum amount of time that it can be blocked waiting for access to the shared data. Thus, the periodic task can take no more than 10 units of time to complete, and, therefore, will always meet its deadline of 20 units. Note that in this example, the only effect the aperiodic task has upon the schedulability of the periodic task is due to blocking.

The second example is composed of a sporadic server with a high priority, the periodic task executing at a medium priority, and the aperiodic task executing either at the high priority of the sporadic server or at a low priority. To determine the schedulability of a task set using a sporadic server, we treat the sporadic server as an equivalently sized periodic task. The use of Equation 1 to determine the schedulability of the second example proceeds as follows:

$$\frac{C_{SS}}{T_{SS}} \le 1(2^1-1)$$
$$\frac{3}{10} \le 1$$
$$\frac{C_{SS}}{T_{SS}} + \frac{C_P}{T_P} + \frac{B_P}{T_P} \le 2(2^{\frac{1}{2}}-1).$$
$$\frac{3}{10} + \frac{8}{20} + \frac{2}{20} \le 0.83$$
$$\frac{8}{10} \le 0.83$$

Since both the inequality for the sporadic server and the inequality for the periodic task are satisfied, the task set is schedulable. However, notice that the inequality for the periodic task involves a term for pre-emption by the aperiodic task (for the case when it is executing at the high priority of its sporadic server) and a term for blocking by the aperiodic task (for the case when the aperiodic task is executing at low priority and has locked the shared data). This is necessary because it is possible for the aperiodic task to be executing at its sporadic server's high priority to exhaust the sporadic server's execution time just after after locking the shared data. In this case, the aperiodic task can both pre-empt *and* block the execution of the periodic task. A periodic task that shares data with an aperiodic task that uses a sporadic server must be able to withstand both the pre-emption time of the sporadic server and the blocking of time of the aperiodic task execution at low priority. Referring to the equations from both examples, one can see that the addition of a high-priority sporadic server can increase the pre-emption time imposed upon a periodic task while not decreasing the blocking time. This "double hit" in terms of schedulability for the periodic task is a drawback of using sporadic servers to provide high priority service to aperiodic tasks that share data with periodic tasks.

Earlier we mentioned that one could use a sporadic server to decrease the amount of blocking time experienced by a periodic task that shares data with an aperiodic task. This can be accomplished if the aperiodic task is not allowed to execute at low priority *and* always completes and releases the shared data before its sporadic server runs out of execution time. If the aperiodic task only executes at its sporadic server's high priority and the sporadic server never suspends aperiodic service when the shared data is locked by the aperiodic task, then the aperiodic task can never block the periodic task. If the application characteristics allow the creation of a sporadic server that can make these guarantees, then the blocking term can be removed from the schedulability inequality for periodic task, improving its schedulability. Implementation considerations for such a sporadic server are discussed in Section 5.5.

3. Implementing a Sporadic Server with an Application-Level Ada Task

This section describes an implementation of a sporadic server as an Ada task. This implementation requires no changes to the Ada runtime system. However, since an Ada task cannot directly monitor the execution time it consumes or alter its priority, the sporadic server algorithm must be simplified and the following assumptions and restrictions must be made:

1. The worst-case execution time of each aperiodic task must be known. Each time an aperiodic task uses its sporadic server, it is assumed that the aperiodic task consumes an amount of sporadic server execution time equal to the aperiodic task's worst-case execution time.
2. Aperiodic tasks that use sporadic servers must rely exclusively upon the sporadic server to execute. In other words, an aperiodic task cannot execute both as a low priority task when the sporadic server's capacity is exhausted or when the processor is idle and then as high priority task when some sporadic server capacity is replenished (as is possible in the full Ada runtime implementation described in Section 4).
3. A sporadic server is not allowed to service an aperiodic task unless it has an available execution time greater than or equal to the worst-case execution time of the aperiodic task. When the sporadic server does not have enough capacity to completely service an aperiodic task, the aperiodic task must wait until the sporadic server's available execution time is replenished to a value greater than or equal to the worst-case execution time of the aperiodic task. This is necessary because once the sporadic server task begins servicing an aperiodic task, it has no way of suspending that service when its available execution time is exhausted.
4. Since it is not possible for the sporadic server task to track the active/idle status of the priority levels in the system, it is necessary to use a simplified version of the sporadic server's replenishment policy (discussed in Section 5.4). This policy requires that consumed execution time be replenished one sporadic server period after sporadic service is initiated.

The basic mechanism used to implement a sporadic server task is Ada's *selective wait with a delay alternative* (SWDA). Each *accept* statement of the SWDA corresponds to one of the aperiodic tasks that can request service from the sporadic server. To request service, an aperiodic task executes a *call* to the corresponding *accept* statement in the sporadic server. Each of the *accept* statements in the sporadic server's SWDA has a *guard* that compares the available execution time of the sporadic server to the worst-case execution time of the aperiodic task. If the sporadic server has enough execution time to completely service the aperiodic task, then the corresponding *select* alternative is *open*. The *delay* statement of the SWDA is used to schedule replenishments for consumed sporadic server execution time. If any replenishments are pending for the sporadic server, then the delay alternative is *open* and the delay will expire when the next replenishment should occur. If upon exiting the *select* statement, it is determined that an aperiodic task has consumed some of the sporadic server's execution time, then a replenishment is scheduled to replenish the consumed execution time.

The sporadic server's replenishments are managed using the record variable, Next_Rep, and a queue of replenishment records referred to as the replenishment queue. If any replenishments are pending, then Next_Rep holds the replenishment time and amount for the next replenishment. If the sporadic server has more than one pending replenishment, all the pending replenishments except Next_Rep are stored in FIFO order on the replenishment queue.

The pseudo-code for the Ada task implementation of a sporadic server is presented in Figure 5. The Sporadic_Service package body relies upon an application-level Ada package describing the worst-case execution times and service procedures for the aperiodic tasks and an application-level Ada package for the sporadic server's replenishment queue management. The specification for these packages, Aperiodic_Tasks and SS_Replenishment_Queue_Manager, are presented in Figure 6.

The task *Sporadic_Server* presented in Figure 5 consists of an infinite loop. The loop contains the SWDA (as described above) and code to manage the replenishing of the sporadic server's execution time. The SWDA supports **N** aperiodic tasks with **N** *accept* statements. Each *accept* statement has a guard that will open it if the sporadic server has enough execution time to service its corresponding aperiodic task. The body of each *accept* statement of the SWDA performs the following operations:

1. The time at which service begins for the aperiodic task is remembered by setting Exec_Begin_Time to the current time.
2. The aperiodic task is serviced by the sporadic server.
3. Consumed_Exec_Time is set to the maximum execution time of the aperiodic task.

The *delay* alternative of the SWDA in Figure 5 is used to replenish the sporadic server's execution time. If any replenishments are pending for the sporadic server (indicated by the boolean variable, Reps_Are_Pending), the *delay* alternative will be open. When replenishments are pending for the sporadic server, the record variable, Next_Rep, holds the amount and replenishment time of the next replenishment. When body of delay alternative is executed a replenishment of consumed sporadic server execution time is due. If the replenishment queue is empty, then the replenishment due is the only outstanding replenishment and, therefore, the sporadic server is brought to full capacity and the Reps_Are_Pending is set to FALSE. If the replenishment queue is not empty, the replenishment amount in Next_Rep is added to Available_Exec_Time and the next replenishment is dequeued from the replenishment queue and stored in Next_Rep.

The code after the SWDA is used to decrement the sporadic server's available execution time and schedule replenishments for the consumed execution time. This code is executed after one of the *accept* alternatives is taken because the value of Consumed_Exec_Time will then be greater than zero. This code first decrements the Available_Exec_Time of the sporadic server by Consumed_Exec_Time. It is then necessary to schedule a replenishment for the consumed execution time. If any replenishments are pending, then the record variable, Next_Rep, already holds the information for the next replenishment and, therefore, the replenishment for the most recently consumed sporadic server execution time must be placed in the replenishment queue. If no replenishments are pending, then the information for this replenishment is placed in Next_Rep and the Reps_Are_Pending boolean variable is set to TRUE. The last part of this code resets the value of Consumed_Exec_Time to zero.

The operation of the *delay* alternative of the SWDA can be different from what is desired. If the evaluation of the *delay* expression is pre-empted after reading the clock but before executing the *delay*, the effect will be to make the delay longer than desired [1]. This will, at best, result in wasted server capacity because the server will be replenished at a later time than desired. At worst, a pre-emption during the evaluation of the delay expression will result in missing a desired response time because server capacity that should be available at a given time will not be. A solution to this problem is to support a *delay_until* [1] capability for the selective wait statement. The *delay_until* statement presents an absolute time to the runtime system instead of a relative duration, and, therefore, does not suffer from the pre-emption problem described above.

Figure 5: Application-Level Sporadic Server

Figure 6: Specifications for the Aperiodic_Tasks and SS_Replenishment_Queue_Manager Packages

4. A Full Implementation of Sporadic Servers in an Ada Runtime System

This section describes an implementation of sporadic servers within an Ada runtime system. This is a *full* implementation in that no simplification of the algorithm described in Section 2 is used, as was necessary for the Ada task implementation described in the previous section. This section begins with a brief description of how sporadic servers can be implemented within the existing semantics of Ada. This is followed by a discussion of the runtime data structures that are assumed to be available within an existing Ada runtime system. Next the discussion of the sporadic server implementation is broken into two parts. First, the data structures and procedures used to schedule aperiodic tasks that use sporadic servers are presented. Second, the data structures and procedures for scheduling sporadic server replenishments are described. The next section describes the sources of overhead in this implementation of sporadic servers and discusses implementation options for reducing the overhead.

4.1. Sporadic Servers and Ada Semantics

An Ada runtime system can support sporadic servers for aperiodic tasks within the semantics of Ada. Ada does not specify any specific scheduling discipline for tasks with undefined priorities. As discussed by Sha and Goodenough in [12], if the priority of a task is not assigned using pragma PRIORITY then the Ada runtime system is free to employ any algorithm for deciding which eligible task to run. Thus, an Ada runtime system can use the sporadic server algorithm to provide high priority service for aperiodic tasks. Implementation dependent pragmas and/or runtime calls can be used to specify scheduling priorities for tasks and the information necessary to create and use sporadic servers.

4.2. Runtime Data Structures for Sporadic Servers

The implementation of sporadic servers within an Ada runtime relies upon these existing data structures in the Ada runtime:

- *Task_Control_Block (TCB)* - a record containing all the information necessary to schedule and execute an Ada task.
- *Task_Ready_Queue* - a priority-ordered list of tasks that are ready to execute. The task at the head of the Task_Ready_Queue is always the currently executing task. Whenever a scheduling decision is made that changes the task at the head of the Task_Ready_Queue, the task placed at the head of the Task_Ready_Queue is selected as the next task to execute. Tasks of equal priority are managed using a FIFO policy.
- *Delay_Queue* - a time-ordered queue of tasks that are suspended waiting for a timing event to occur (the Delay_Queue is typically used to implement the Ada *delay* statement).

In addition to the above data structures, new data structures and modifications to existing data structures are necessary to support a complete implementation of sporadic servers. These runtime modifications are needed to support the two primary operations of sporadic servers: (1) scheduling aperiodic tasks that will consume sporadic server execution time and (2) scheduling replenishments for consumed sporadic server execution time. This section breaks the discussion of an Ada runtime implementation of sporadic servers into these two categories. As the data structures for each category have been defined, pseudo-code for the corresponding procedures that manipulate the data structures is presented and discussed.

4.3. Data Structures for Scheduling Aperiodic Tasks that Use Sporadic Servers

4.3.1. Sporadic Server Queues

A full implementation of sporadic servers in an Ada runtime system allows multiple sporadic servers to be created and used concurrently. To manage multiple sporadic servers and aid in the scheduling aperiodic tasks that will use their sporadic server's execution time, several sporadic server queues are maintained by the runtime system. The elements of these queues are pointers to *Sporadic Server Control Blocks* (SSCBs) which are discussed in Section 4.3.3. The sporadic server queues are listed below:

- *SS_Ready_Queue* - a priority-ordered queue of SSCBs that have both aperiodic tasks ready to execute and execution time available to service the aperiodic tasks.
- *SS_Enabled_Queue* - a priority-ordered queue of all SSCBs that are currently enabled. If the SS_Enabled_Queue is empty, then no sporadic servers have been created for use (i.e., either none have ever been created or all that have been created have been terminated).

4.3.2. Aperiodic Task Queues

A sporadic server is created to service one or more aperiodic tasks. The following queues are used to manage the aperiodic tasks associated with a sporadic server:

- *Aperiodic_Ready_Queue* - a queue of ready-to-execute aperiodic tasks. An Aperiodic_Ready_Queue exists for each sporadic server. If an aperiodic task is ready to execute, then the runtime system places the aperiodic task's TCB upon the Aperiodic_Ready_Queue associated with the sporadic server assigned to the aperiodic task. Similarly, if the aperiodic task is ever not ready to execute, then the runtime system removes the aperiodic task from the Aperiodic_Ready_Queue. The Aperiodic_Ready_Queue queue is managed with a FIFO queueing discipline. If preferential service for some aperiodic tasks is desired, a separate sporadic server with a higher priority can be used.
- *Registered_Aperiodics_List* - a list of aperiodic tasks that are registered to use the sporadic server. The information on this list is used to unregister aperiodic tasks from the sporadic server if sporadic service is ever terminated (e.g., during a mode change [11]).

4.3.3. The Sporadic Server Control Block (SSCB)

To support sporadic servers in an Ada runtime system, a new data type is needed to contain the information about each sporadic server created by the runtime system. This new data type is the Sporadic Server Control Block (SSCB). The following fields in the SSCB are used to schedule aperiodic tasks that will consume sporadic server execution time (other SSCB fields will be discussed in Section 4.5.4):

- *Period* - the period of the sporadic server
- *Priority* - the priority of the sporadic server
- *Max_Exec_Time* - the maximum execution time of the sporadic server
- *Avail_Exec_Time* - the execution time the server has available for aperiodic service
- *SS_Ready_Queue_Link* - a pointer to the next SSCB on the SS_Ready_Queue

- *On_SS_Ready_Queue* - a boolean value that indicates whether or not the sporadic server is present on the SS_Ready_Queue
- *SS_Enabled_Queue_Link* - a pointer to the next SSCB on the Enabled_SS_Queue
- *Exhausted_Task* - a pointer to the TCB of a dummy task that is used to suspend the processing of a task when the sporadic server it is using exhausts its available execution time (the use of an Exhausted_Task is discussed in Sections 4.3.7 and 4.4)
- *Aperiodic_Ready_Queue_Head* - a pointer to the first task on the sporadic server's Aperiodic_Ready_Queue
- *Registered_Aperiodics_Head* - a pointer to the head of the Registered_Aperiodics_List

Each sporadic server's period, priority, and maximum execution time are specified by either implementation-dependent pragmas or runtime calls. Pragmas or runtime calls are also necessary to register each aperiodic task with its sporadic server.

4.3.4. Task Control Block Extensions

Implementation of sporadic servers requires some information to be added to the TCB of each task. These additions are summarized below:

- *Base_Priority* - the default execution priority of the task.
- *Current_Priority* - the priority at which the task can currently execute. Both Base_Priority and Current_Priority are necessary since the sporadic server algorithm adjusts the priority of an aperiodic task depending upon whether or not it is using its sporadic server. The Current_Priority is the priority used to queue TCBs on the Task_Ready_Queue.
- *Task_Category* - the category to which this task is associated. A task that can execute is in either the *Normal_Task* category or in the *Aperiodic_Task* category. Only tasks in the Aperiodic_Task category can use a sporadic server. To prevent an aperiodic task from consuming more than the available sporadic server capacity a special task category is used: *Exhausted_Task*. A task in this special category is never executed as an actual task; it is merely added to the Delay_Queue when appropriate. When the delay for a task in the Exhausted_Task category expires, the available execution time for the corresponding sporadic server has been exhausted and sporadic service must be suspended. An Exhausted_Task exists for each sporadic server. Another special task category, the *Replenish_Task*, is defined later in Section 4.5.
- *My_Sporadic_Server* - a pointer to the SSCB used by this task. The pointer is set to null if the type of the task is Normal_Task.
- *Using_Sporadic_Server* - a boolean value that indicates whether or not the task is currently consuming its sporadic server's execution time.
- *Aperiodic_Queue_Link* - a pointer to the TCB of the next aperiodic task on the Aperiodic_Ready_Queue associated with this task's sporadic server.
- *On_Aperiodic_Queue* - a boolean value that indicates whether or not this task is on its sporadic server's Aperiodic_Ready_Queue.
- *Registered_Aperiodic_Link* - a pointer to the TCB of the next aperiodic task on the Registered_Aperiodic_List.

4.3.5. Sporadic Server Data Structure Example

Figure 7 presents an example of the sporadic server data structures. In this section we will be discussing the data structures used to schedule aperiodic tasks that lie outside the gray box in Figure 7. The data structures enclosed in the gray box are used to manage the replenishment of consumed sporadic execution time and will be defined and discussed later in Sections 4.5 and 4.6.

In Figure 7, five sporadic servers, SSCB-1 through SSCB-5, have been created and placed on the SS_Enabled_Queue in priority order (SSCB-1 having the highest priority and SSCB-5 having the lowest priority). Although five sporadic servers have been created, only two of them are "ready" in the sense that they have execution time available and aperiodic tasks ready to consume the execution time. These two "ready" sporadic servers (SSCB-3 and SSCB-5) have been placed on the SS_Ready_Queue in priority order.

Referring to the control block of the third sporadic server (SSCB-3) in Figure 7 we can confirm that it should be on the SS_Ready_Queue because its Avail_Exec_Time is greater than zero and its Aperiodic_Ready_Queue has two ready-to-execute aperiodic tasks, TCB-1 and TCB-2. By following the links from the Registered_Aperiodics_Head we can see that three aperiodic tasks (TCB-1, TCB-2, and TCB-3) are registered to use this sporadic server. Also associated with this sporadic server is its Exhausted_Task.

4.3.6. Modification of the Task_Ready_Queue Support Routines

As described above, the runtime system maintains two queues for "ready-to-execute" sporadic servers and aperiodic tasks. The runtime system maintains an SS_Ready_Queue that is updated whenever the readiness of a sporadic server or an aperiodic task changes. Also, each sporadic server has an Aperiodic_Ready_Queue that is updated whenever the readiness changes for one of the sporadic server's aperiodic tasks.

Typically, an Ada runtime will use procedures to manipulate the Task_Ready_Queue. For a sporadic server implementation, these procedures must be modified to adjust the SS_Ready_Queue and/or the sporadic server's Aperiodic_Ready_Queue whenever appropriate. As an aperiodic task is added to or removed from the Task_Ready_Queue it should *also* be added to or removed from its sporadic server's Aperiodic_Ready_Queue. Also, as a sporadic server's Aperiodic_Ready_Queue is adjusted it is necessary to determine if the sporadic server should be added to or removed from the SS_Ready_Queue. The pseudo-code for these procedures is presented in Figure 8. It is also necessary to check if a sporadic server should be added to or removed from the SS_Ready_Queue whenever the sporadic server's execution time is exhausted or replenished. These checks are discussed in Section 4.5.

Figure 7: Sporadic Server Runtime Data Structures

```
procedure Add_Task_To_Ready_Queue(task : TCB) is
begin

  Add the task to the Task_Ready_Queue;

  if task.Task_Category = Aperiodic_Task then

      Add the task to its sporadic server's Aperiodic_Ready_Queue;

      if (not task.My_Sporadic_Server.On_SS_Ready_Queue) and then
         task.My_Sporadic_Server.Avail_Exec_Time > 0.0 then

         Add the task's sporadic server to the SS_Ready_Queue;

      end if;

  end if;

end Add_Task_To_Ready_Queue;
```

Figure 8: Add_Task_To_Ready_Queue and Remove_Task_From_Ready_Queue

4.3.7. Use of the Delay_Queue for Scheduling Aperiodic Tasks Using Sporadic Servers

An Ada runtime system typically has a Delay_Queue that is used to implement the Ada delay statement. When a task executes a delay statement, the task is removed from the Task_Ready_Queue and placed upon the Delay_Queue to be awakened (moved back to the Task_Ready_Queue) after some specified delay. A sporadic server implementation also uses the Delay_Queue as a mechanism to prevent the execution of an aperiodic task from consuming more sporadic server execution time than it is allocated. The Exhausted_Task category is used for this purpose.

As an aperiodic task is about to begin execution and use its sporadic server, the Exhausted_Task associated with the aperiodic task's sporadic server is placed on the Delay_Queue with a delay equal to the available execution time of the sporadic server. If the processing of the aperiodic task completes, is suspended, or pre-empted before the Exhausted_Task's delay expires, the Exhausted_Task is simply removed from the Delay_Queue. However, if the delay for the Exhausted_Task expires before the aperiodic task completes then the sporadic server has exhausted its available execution time. In this case, sporadic service for the aperiodic task is suspended, a replenishment is scheduled for the consumed execution time, and the Current_Priority of the aperiodic task is reset to its Base_Priority (these actions are taken in the sporadic server procedure *Mark_SS_Consumption* which is discussed in Section 4.6.1).

4.4. Scheduling Aperiodic Tasks Using Sporadic Servers

Now that the data structures used in scheduling aperiodic tasks that use sporadic servers have been presented, the conditions governing the execution of an aperiodic task with a sporadic server can be discussed.

The conditions under which an aperiodic task should be scheduled to execute using a sporadic server are as follows:

1. The aperiodic task must be ready to execute;
2. The aperiodic task's sporadic server must have execution time available to execute the aperiodic task;
3. The sporadic server must have a priority equal to or higher than all other tasks that are ready to execute;
4. Use of the sporadic server must be necessary in order to execute the aperiodic task. Otherwise, it would be wasteful to use a sporadic server's execution time for an aperiodic task that could execute without the server (e.g., when the processor would otherwise be idle).

The above rules are simply stated, but an application can have many sporadic servers with different priorities and each sporadic server could be supporting several aperiodic tasks. As such, a general test of the above conditions every time a new task becomes ready to execute could be expensive, slowing the system's response to external events and increasing the general runtime overhead of the system. To avoid these problems, a queue of sporadic servers is used. The SS_Ready_Queue is a priority-ordered queue of sporadic servers that have both at least one aperiodic task ready to use the sporadic server and execution time available to execute the aperiodic task. Using the SS_Ready_Queue greatly simplifies the check of the first two conditions above. If the head of the SS_Ready_Queue is null then none of the system's sporadic servers are ready to be used because either they have no execution time available or none of their aperiodic tasks are ready to execute. If the head of the SS_Ready_Queue is not null then it is necessary to check the third condition above. The sporadic server can be used only if it has a priority equal to or greater than the task that would normally execute next. Finally, if the third condition passes then the last check to be made determines whether or not the high priority of sporadic server is necessary to execute the aperiodic task. This method of checking for sporadic servers is efficient because no searches are necessary to find "ready" sporadic servers and aperiodic tasks.

Once it is determined that an aperiodic task can execute using its sporadic server the following operations are necessary. The aperiodic task's Current_Priority is set equal to the priority of its sporadic server and the aperiodic task's Using_Sporadic_Server boolean is set to TRUE. Next, the aperiodic task is moved to the head of the Task_Ready_Queue. Before execution of the aperiodic task begins it is necessary to add the sporadic server's Exhausted_Task to the Delay_Queue to prevent the aperiodic task from overrunning the sporadic server's available execution time. Finally, for proper monitoring of the execution time consumed by the aperiodic task, the time that execution begins must be recorded (this is used later by the procedure *Mark_SS_Consumption* discussed in Section 4.6.1). The time at which execution begins is recorded in the sporadic server's Replenishment_Control_Block (the RCB is discussed in Section 4.5).

Figure 9: Execute_Next_Task

Figure 9 presents the pseudo-code for the procedure *Execute_Next_Task*. The first half of Execute_Next_Task implements the operations described above for determining if a aperiodic task can execute using its sporadic server and then prepares the aperiodic task for execution. The second half of Execute_Next_Task shows the conditions that must be checked every time tasks are switched if sporadic servers are in use. These checks concern the accurate monitoring of the consumption of a sporadic server's execution time and the proper replenishing of any consumed sporadic server execution time and are discussed in Sections 4.5 and 4.6.

4.5. Data Structures For Scheduling Sporadic Server Replenishments

4.5.1. The Replenishment Data Type

A new data type, *replenishment*, is the basic unit of information that is managed by the sporadic server for replenishing consumed execution time. A replenishment has two fields:

- *Rep_Time* - the time at which a replenishment is to be performed.
- *Rep_Amount* - the amount of execution time to be added to the sporadic server's Avail_Exec_Time.

4.5.2. Sporadic Server Replenishment Queues

A sporadic server's execution time can be consumed during several distinct intervals of time, each requiring a separate replenishment. As such, a queue of outstanding replenishments, the *Rep_Queue*, must be maintained for each sporadic server. Each Rep_Queue is a FIFO queue of Replenishments whose Rep_Times have not been reached yet. For efficiency, the storage for the Rep_Queue should not be created dynamically, but instead preallocated at compile time.

4.5.3. The SS_Used_Queue

The runtime system maintains the *SS_Used_Queue*, a priority-ordered queue of sporadic servers that have had some of their execution time consumed but have not yet scheduled the replenishment for the consumed execution time. As priority levels become idle, the SS_Used_Queue is checked for any sporadic servers that have replenishments that need to be scheduled.

4.5.4. SSCB Fields for Managing Replenishments

An SSCB has the following additional fields that are used to manage the replenishment of consumed sporadic server execution time:

- *Replenish_Task* - a pointer to the TCB of a dummy task that is used to implement replenishments for a sporadic server. When a Replenish_Task is awakened and removed from the Delay_Queue, the corresponding sporadic server is replenished using the replenishment information at the head of the sporadic server's replenishment queue.
- *SS_Used_Queue_Link* - a pointer to the SSCB of the next sporadic server on the SS_Used_Queue.
- *On_SS_Used_Queue* - a boolean value that indicates whether or not the sporadic server is present on the SS_Used_Queue.
- *Replenish_Data* - a pointer to the Replenishment_Control_Block (RCB) for this sporadic server. The RCB, described in Section 4.5.5, contains all the information concerning the outstanding replenishments for a sporadic server.

4.5.5. The Replenishment Control Block

Each sporadic server has an RCB that contains the information about the outstanding replenishments for the sporadic server. The fields of the RCB are:

- *Rep_Queue* - a pointer to the head of a FIFO queue of Replenishments whose Rep_Times have not been reached yet.
- *Rep_Origin* - the time from which the Rep_Time of a Replenishment is determined (i.e., the actual replenishment time is equal to the Rep_Origin plus the period of the sporadic server). Usually, the Rep_Origin corresponds to the time at which the sporadic server's priority level becomes active (the exception occurs when the sporadic server has exhausted its execution time as shown in Figure 4 in Section 2).
- *Exec_Begin_Time* - the time at which the sporadic server begins servicing an aperiodic request. This value is used to compute the amount of sporadic server execution time that is consumed by an aperiodic task.
- *Pending_Replenishment* - a Replenishment that has not yet been placed in the Rep_Queue. Pending_Replenishment is used to accumulate the execution time that is consumed throughout one interval of time during which the sporadic server's priority level is active. If any sporadic server execution time is consumed, its Pending_Replenishment is placed on the Rep_Queue when the sporadic server's priority level becomes idle or when its execution time is exhausted. If the Rep_Queue ever becomes full, Pending_Replenishment is used to accumulate replenishments until an entry on the Rep_Queue becomes available. Note that for the cases when the Rep_Queue becomes full, the earliest allowed replenishment for consumed execution time may not occur.

4.5.6. Replenishment Data Structure Example

Now that the data structures used to manage the replenishment of consumed sporadic server execution time have been presented we can refer to Figure 7 to see an example of these data structures. Two sporadic servers, SSCB-3 and SSCB-5, are on the SS_Used_Queue indicating that some of their execution time has been consumed but that the replenishment for the consumed execution time has not yet been placed on their respective replenishment queues. The Rep_Queue for SSCB-3 is shown to have three outstanding replenishments that are waiting for their replenishment times to arrive. Also shown is SSCB-3's Replenish_Task that is placed upon the Delay_Queue to wake up at the time the next replenishment is due.

4.6. Scheduling Sporadic Server Replenishments

The following are the basic operations necessary for handling sporadic server replenishments:

1. The consumption of sporadic server execution time must be monitored and the replenishment amounts must be computed.
2. The times at which consumed sporadic server execution time are to be replenished must be determined.
3. Each outstanding replenishment for a sporadic server must be queued until the replenishment of sporadic server execution time is made.

The procedures for implementing these operations are presented in this section.

4.6.1. Tracking the Consumption of Sporadic Server Execution Time

Each time the execution of an aperiodic task that is using its sporadic server can be stopped, it is necessary to keep track of the amount of the sporadic server's execution time that was consumed. The execution of an aperiodic task that is using its sporadic server can be stopped by one of the following events:

1. The aperiodic task execution is pre-empted by a higher priority task;
2. The aperiodic task suspends;
3. The aperiodic task completes execution; or,
4. The execution time of the aperiodic task's sporadic server is exhausted.

The first three cases above are detected in the second half of the procedure Execute_Next_Task (Figure 9) as task execution is switched from one task to another. If the previously executing task was using its sporadic server and the next task to execute is a different task, then one of the first three cases above holds. The fourth case above is checked by the procedure that processes tasks on the Delay_Queue as their delays expire. For each of the above cases, the procedure Mark_SS_Consumption is called. The pseudo-code for the Mark_SS_Consumption procedure is presented in Figure 10 and discussed below.

The first job that Mark_SS_Consumption must perform is to determine the reason the execution of the aperiodic task has stopped and, as necessary, adjust the sporadic server data in the aperiodic task's TCB and/or re-queue the aperiodic task on the Task_Ready_Queue. The sporadic server data in the aperiodic task's TCB must be adjusted if the aperiodic task is no longer ready to execute or if its sporadic server has exhausted its execution time. Thus, in cases 2, 3, and 4 the aperiodic task's Current_Priority must be set equal to its Base_Priority and its Using_Sporadic_Server boolean must be set to FALSE. In case 1, neither of these changes is necessary because when the higher priority activity ceases, the aperiodic task will still be ready to execute and its sporadic server will still have execution time available.

To determine if the aperiodic task should be re-queued on the Task_Ready_Queue it is necessary to determine its readiness. Even though the execution of the aperiodic task has stopped, the aperiodic task may still be ready to execute and, therefore, still be on the Task_Ready_Queue. In cases 1 and 4 above, the aperiodic task remains ready to execute but its priority is no longer high enough to execute because either a higher priority task can now execute or the aperiodic task's sporadic server has exhausted its execution time. In both of these cases, the aperiodic task should remain on the Task_Ready_Queue. For case 1, the aperiodic task can remain in its current position on the Task_Ready_Queue because its priority remains unchanged. However, in case 4 the aperiodic task must be re-queued on the Task_Ready_Queue because its priority has dropped from its sporadic server's priority to its base priority.[2]

[2]Note: Re-queueing of a task to the Task_Ready_Queue is different from placing a task on the Task_Ready_Queue that is not already there. When placing a task on the Task_Ready_Queue that is not already there, tasks of equal priority are queued in FIFO order. However, re-queueing of a task on the Task_Ready_Queue requires that tasks of equal priority be placed on the queue in LIFO order.

Next, the Delay_Queue must be checked for the presence of the sporadic server's Exhausted_Task. In each of the first three cases above, consumption of the sporadic server's execution time has stopped but the sporadic server's Exhausted_Task is still on the Delay_Queue and, therefore, must be removed.

The consumption of sporadic server execution time is then calculated and any necessary adjustments to the SS_Ready_Queue are made. The amount of execution time consumed is determined by subtracting Exec_Begin_Time from the current time. Exec_Begin_Time was set in the procedure Execute_Next_Task before the aperiodic task was scheduled to execute (see Section 4.6.1). Since some sporadic server execution time has been consumed, it is necessary to check if the sporadic server should remain on the SS_Ready_Queue as is done when adding or removing an aperiodic task from the Task_Ready_Queue (Section 4.3.6).

Next, the amount of consumed execution time must be used to update the sporadic server's state and be scheduled for replenishment. The sporadic server's execution time is decremented by the amount of execution time consumed. The total amount of consumed sporadic server execution time during this active period (which is stored in Pending_Replenishment) is incremented by the amount of consumed execution time. Finally, if the sporadic server's execution time is exhausted, then the Pending_Replenishment must be placed on the replenishment queue with a call to the procedure *Queue_Pending_Replenishment* described in Section 4.6.3. Note that this check for sporadic server exhaustion compares for an Avail_Exec_Time that is less than zero. This allows for accounting errors in the value of Avail_Exec_Time due to runtime overhead and clock granularity. If the sporadic server is not exhausted, then it is placed on the SS_Used_Queue from which the procedure Track_Active_Idle_Status (Section 4.6.2) will schedule the sporadic server's replenishment when its priority level becomes idle.

Figure 10: Mark_SS_Consumption

4.6.2. Tracking the Active/Idle Status of Sporadic Server Priority Levels

A full implementation of the sporadic server algorithm requires that the active/idle status of the sporadic server priority levels be tracked as different tasks are scheduled. This is essential because of the following two rules concerning sporadic server replenishments:

1. The time at which a sporadic server's consumed execution time is to be replenished is determined from the time at which the sporadic server's priority level becomes active.
2. The time at which the total amount of consumed execution time for a sporadic server can be determined occurs when the sporadic server's priority level becomes idle.

When a new task is executed that has a priority higher than the previous task, some sporadic server priority levels can become active. As the priority level for a sporadic server becomes active, its replenishment origin (Rep_Origin) must be set equal to the current time. The replenishment origins are set by using a priority-ordered list of enabled sporadic servers, the Enabled_SS_Queue. Similarly, when a new task is executed that has a priority lower than the previous task, then some sporadic server priority levels can become idle. As the priority level of a sporadic server becomes idle, the sporadic server's pending replenishment (if it has positive replenish amount) can be placed on the sporadic server's Rep_Queue. The pending replenishments are placed on the Rep_Queue by using the SS_Used_Queue to select the sporadic servers whose priorities lie in the range of the newly idle priority levels. These are the operations performed by the procedure *Track_Active_Idle_Status*. The variables *Previous_Priority_Level* and *Next_Priority_Level* are used by this procedure to track the active/idle status of sporadic server priority levels. The values of Previous_Priority_Level and Next_Priority_Level are set as different tasks are scheduled for execution by the procedure Execute_Next_Task (Figure 9). The pseudo-code for Track_Active_Idle_Status is presented in Figure 11. The first section of Track_Active_Idle_Status corresponds to the first rule above and the second section corresponds to the second rule above.

The two exceptions to the above sporadic server replenishment rules occur when the sporadic server's execution time has been exhausted. The first exception concerns the queueing of the replenishment for the consumed execution time. This case is tested by the procedure Mark_SS_Consumption (Section 4.6.1) which is called after each block of sporadic server execution time is consumed. When Mark_SS_Consumption detects that the sporadic server's execution time has been exhausted, the procedure Queue_Pending_Replenishment (Section 4.6.3) is called to add the pending replenishment to the replenishment queue. The second exception concerns the setting of a sporadic server's replenishment origin (Rep_Origin) after the sporadic server's execution time has been exhausted. As some execution time is replenished after the sporadic server's capacity is exhausted, the replenishment origin should be set equal to the current time to prevent any consumed execution time from being replenished too early (Figure 4 shows an example of this). The case is tested by the procedure Replenish_Sporadic_Server (discussed later in Section 4.6.4) which is called as each sporadic server's replenish task wakes up on the Delay_Queue. When Replenish_Sporadic_Server detects this case the replenishment origin is set to the current time.

Figure 11: Track_Active_Idle_Status

4.6.3. Queueing Sporadic Server Replenishments

Several conditions need to be checked when a request is made to add a replenishment to a sporadic server's replenishment queue. First, the pending replenishment cannot be added to the replenishment queue if the replenishment queue is full. The replenishment queue can become full if the consumption of execution time occurs during many distinct intervals of time (where each consumption requires a different replenishment time) such that the last consumption of execution time fills the replenishment queue. In this case of a full replenishment queue, Pending_Replenishment is left unchanged and will be added to the replenishment queue when space becomes available by the procedure *Replenish_Sporadic_Server* discussed in Section 4.6.4. Until space becomes available on the replenishment queue, additional replenishments may be accumulated in Pending_Replenishment. Note that in this case, the replenishment time of the last replenishment accumulated in Pending_Replenishment is used as the replenish time when Pending_Replenishment is placed on the replenishment queue. If the replenishment queue is not full, the pending replenishment is added to its replenishment queue and the replenish amount of Pending_Replenishment is reset zero. Next, since the sporadic server no longer has a pending replenishment, the sporadic server is removed from the SS_Used_Queue. Now that the replenishment queue has at least one entry, the sporadic server's replenish task must be placed upon the Delay_Queue if it is not already there. The pseudo-code for the procedure Queue_Pending_Replenishment is presented in Figure 12.

Figure 12: Queue_Pending_Replenishment

4.6.4. Replenishing Sporadic Servers

The procedure Replenish_Sporadic_Server (presented in Figure 13) is called whenever the delay expires for a sporadic server's replenish task. This procedure dequeues the replenishment at the head of the sporadic server's replenishment queue and increments the sporadic server's available execution time using the replenish amount from the dequeued replenishment.

The operation of Replenish_Sporadic_Server procedure proceeds as follows. First, the sporadic server's available execution time is checked and, if it is zero, the sporadic server's replenishment origin is set to the current time as discussed earlier in Section 4.6.2. Next, a replenishment is removed from the replenishment queue and the replenish amount is added to the sporadic server's available execution time. If the replenishment queue is not empty, the sporadic server's replenish task is then added to the Delay_Queue to wake up at the replenish time of the replenishment at the head of the replenishment queue. Now that an entry on the replenishment queue has been freed, the sporadic server's pending replenishment is checked and if it has a positive replenish amount, the pending replenishment is added to the replenishment queue. Since the sporadic server's available execution time has just been increased, the last operation of Replenish_Sporadic_Server is to check if the sporadic server should be added to the SS_Ready_Queue.

Figure 13: Replenish_Sporadic_Server

5. Implementation Options for Sporadic Servers in an Ada Runtime

This section discusses several replenishment and execution options for implementing sporadic servers in an Ada runtime. The replenishment options are concerned with controlling and/or reducing the overhead necessary to queue and schedule replenishments. The execution options concern providing high priority service to only those aperiodic requests that can be completely serviced without interruption. These options are not necessary for a fully functional implementation. However, depending upon the application, these options may provide a better solution than the full implementation described in Section 4.

5.1. Implementation Concerns

The options for managing replenishments in a sporadic server implementation address the following questions:

- How much execution time must be consumed before a replenishment is placed upon the rep_queue?
- When should replenishments be removed from the replenishment queue and when should the Replenish_Task be placed upon the delay queue?
- Can the tracking of the active/idle status of priority levels be eliminated?
- Can sporadic service for aperiodic tasks be restricted to only those aperiodic tasks that can be completely serviced before the sporadic server's available execution time is exhausted?

5.2. Specification of a Minimum Replenishment Amount

The overhead of queueing replenishments and adding replenish amounts to the sporadic server's available execution time can become a concern when the sporadic server's maximum execution time is much larger than the expected execution times of the aperiodic tasks that share the sporadic server. Under these conditions, many separate replenishments can sometimes be necessary because the sporadic server's execution time can be consumed in many separate, small amounts before any replenishments are made. As more replenishments are necessary, the overhead necessary to place each replenishment upon the replenishment queue and add each replenish amount to the available server execution time may be a concern.

To reduce this replenishment overhead, a *minimum replenishment amount* can be specified. As the sporadic server's execution time is consumed, it is accumulated in the pending replenishment. Only when the amount accumulated in the pending replenishment exceeds the minimum replenishment amount is the replenishment placed on the replenishment queue. When the replenishment is placed on the replenishment queue in this manner, the replenishment time of the most recent replenishment is used as the time at which this replenish amount is to be added to the sporadic server's available execution time. Thus, the replenishment overhead of several separate consumptions of sporadic server execution time is reduced to the overhead of one replenishment. Also, specification of a minimum replenishment amount limits the maximum length of the replenishment queue (i.e., the maximum queue length will be equal to the maximum server execution time divided by the minimum replenishment amount) and allows the implementation to eliminate all checks for replenishment queue overflow. However, this option does have the drawback that the times replenishments are made can be later than with the full replenishment policy.

An additional option, *flush*, can be used in conjunction with a minimum replenishment amount. The flush option specifies that, if a consumption of sporadic server execution time consumes less than the minimum replenishment amount, then assume that the minimum replenish amount has been consumed (i.e., *flush* the difference between the actual amount consumed and the minimum replenishment amount) and schedule the replenishment. This option provides an earlier replenishment schedule than specifying just a minimum replenishment, but it can waste the sporadic server's unused execution time.

5.3. Scheduling Replenishments Only Upon Server Exhaustion

To further reduce the replenishment overhead, replenishments can be removed from the replenishment queue only when the sporadic server's execution time is exhausted. Upon exhaustion of the server's execution time, the replenishment queue is checked for any replenishments whose replenish time has already passed. The replenish amounts for these replenishments are then added to the sporadic server's available execution time and sporadic service is continued. If no replenishments have replenish times earlier than the current time, then the Replenish_Task must be placed on the delay queue using the information at the head of the replenishment queue. This technique reduces the overhead of managing replenishments by removing them from the replenishment queue only when necessary (i.e. when no more sporadic server execution time remains). Also, since the Replenish_Task is only added to the delay queue when the server's execution time has been completely consumed, the frequency of adding and removing the replenish task to the delay queue can be greatly reduced. Using the replenish task less also reduces the timer interrupts the runtime system must process.

5.4. Eliminating the Tracking of Active/Idle Status of Priority Levels

A full implementation of sporadic servers requires that the active/idle status of sporadic server priority levels be tracked as different tasks are scheduled (Section 4.6.2). The time at which a sporadic server's priority level becomes active is used as the time origin to calculate the replenishment time for any consumed sporadic server execution time. Although this approach provides the the earliest replenishment schedule as shown in [13], it requires that the active/idle status of priority levels be tracked on every task switch even if no sporadic server execution time is consumed. Thus, the total overhead to track the active/idle status is directly related to the frequency of task switches.

A simpler replenishment policy can be used to eliminate the necessity of tracking the active/idle status of sporadic server priority levels. By definition, a sporadic server's priority level must be active if its execution time is being consumed. Therefore, using the time at which the sporadic server's execution time is initially consumed as the origin from which replenishment times are determined cannot provide an earlier replenishment schedule than tracking the active/idle status of priority levels does. This simple replenishment policy requires that the consumed sporadic server execution time be scheduled for replenishment one server period after sporadic service is initiated (as is done in the example presented in Figure 1). The drawback of this simple replenishment policy is that some replenishments do not occur as early as possible.

Using this simple replenishment policy allows the elimination of the procedure Track_Active_Idle_Status (Section 4.6.2). Replenishments must then be scheduled as each portion of sporadic server execution time is consumed. The procedure Mark_SS_Consumption (Section 4.6.1) can be modified to schedule replenishment for sporadic server execution time as it is consumed.

5.5. Avoiding Suspension of Aperiodic Service due to Sporadic Server Exhaustion

It may be desirable to only begin the execution of an aperiodic task if its sporadic server can completely service the aperiodic task before exhausting its available execution time. This requires that the sporadic server have available the maximum amount of execution time that the aperiodic task could require before granting service to the aperiodic task. This amount of execution time can be specified in two ways:

1. Declare a minimum amount of execution time a sporadic server must have before providing service to any aperiodic task.
2. Specify for each aperiodic task the maximum execution required to complete the aperiodic task (this information could be added to the aperiodic task's TCB).

The first option above is easily implemented by changing the checks for placing a sporadic server on the SS_Ready_Queue to require at least the minimum execution time instead of just an available execution time greater than zero (see the procedures presented in Figures 8 and 13).

The second option above would require much more overhead to determine when a sporadic server could service an aperiodic task, unless a shortest-job-first policy were adopted for ordering the service of all the aperiodic tasks registered to a particular sporadic server. In this case, the aperiodic tasks would be queued in shortest-job-first order on the sporadic server's Aperiodic_Ready_Queue. The check for placing a sporadic server on the SS_Ready_Queue would then compare the sporadic server's available execution time to the execution time required by the aperiodic task at the head of its Aperiodic_Ready_Queue. Note that servicing aperiodic tasks in shortest-job-first order may "starve" some long aperiodic tasks.

6. Summary

This paper described implementation approaches for the Sporadic Server (SS) algorithm. The SS algorithm provides a general solution for scheduling both soft and hard-deadline aperiodic tasks. The average response time performance of soft-deadline aperiodic tasks can be greatly improved by creating a sporadic server that is shared by the soft-deadline aperiodic tasks. The response time for hard-deadline aperiodic tasks (sporadic tasks) can be guaranteed by dedicating a sporadic server to exclusively service the sporadic task. The SS algorithm can also be used to implement the period transformation technique [9], to provide improved fault detection and containment capabilities for some execution errors, and as a general mechanism for implementing solutions for producer/consumer problems. The SS algorithm operates by creating a server with a given period and execution time. The server maintains its execution time until it is needed and replenishes any consumed execution time in a sporadic manner based upon when the execution time is consumed.

This paper has presented two approaches for implementing sporadic servers in a real-time system programmed in Ada. The first approach implements sporadic servers as a normal Ada task and, as such, requires no modifications to the Ada language or its runtime system. However, the implementation of a sporadic server as an Ada task requires a simplification of the sporadic server algorithm that may reduce its response time performance for servicing aperiodic tasks. The second approach is a full implementation of the sporadic server algorithm in an Ada runtime system. This implementation requires modifications to the Ada runtime system but it is argued that the addition of these new scheduling policies is permissible within the semantics of the Ada language.

Acknowledgments

The author would like to thank Ragunathan Rajkumar for his help with the design of the Ada runtime sporadic server implementation which borrowed several ideas from his prototype deferrable server implementation. The author would also like to thank members of the SEI's RTSIA project: Mark Borger, John Goodenough, and Mark Klein for reviewing drafts of this paper, offering many constructive comments, and improving the quality of the Ada code presented herein.

References

[1] Mark Borger, Mark Klein, Robert Veltre.
Real-Time Software Engineering in Ada: Observations and Recommendations.
In *Proceedings of TRI-Ada '89*, pages 554-569. ACM/SIGAda, Pittsburgh, PA, October, 1989.

[2] Sadegh Davari and Sudarshan K. Dhall.
An On line Algorithm For Real-Time Tasks Allocation.
In *Proceedings of the 7th Real-Time Systems Symposium*, pages 194-200. IEEE, New Orleans, Louisiana, December, 1986.

[3] S. K. Dhall and C. L. Liu.
On a Real-Time Scheduling Problem.
Operations Research 26(1):127-140, February, 1978.

[4] J. P. Lehoczky, L. Sha, and Y. Ding.
The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.
Technical Report, Department of Statistics, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1987.

[5] John P. Lehoczky, Lui Sha, Jay K. Strosnider.
Enhanced Aperiodic Responsiveness in Hard Real-Time Environments.
In *Proceedings of the Real-Time Systems Symposium*, pages 261-270. IEEE, San Jose, CA, December, 1987.

[6] C. L. Liu and James W. Layland.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[7] A.K. Mok.
Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.
PhD thesis, M.I.T., 1983.

[8] Ragunathan Rajkumar, Lui Sha, John P. Lehoczky.
Real-Time Synchronization Protocols for Multiprocessors.
In *Proceedings of the Real-Time Systems Symposium*, pages 259-269. IEEE, Huntsville, Alabama, December, 1988.

[9] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar.
Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.
In *Proceedings of the Real-Time Systems Symposium*, pages 181-191. IEEE, New Orleans, Louisiana, December, 1986.

[10] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report CMU-CS-87-181, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, November, 1987.

[11] Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K.
Mode Changes in a Prioritized Preemptive Scheduling Environment.
Accepted for Publication, Real-Time Systems Journal , 1989.
Also available as a Technical Report, Software Engineering Institute.

[12] Lui Sha and John Goodenough.
Real-Time Scheduling Theory in Ada.
Technical Report, Software Engineering Institute, Carnegie Mellon University, 1989.
CMU/SEI-TR-89-14, DTIC: ADA211397.

[13] Sprunt, B., Sha, L. and Lehoczky, J. P.
Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.
Technical Report, Software Engineering Institute, Carnegie Mellon University, 1989.
CMU/SEI-89-TR-11, DTIC: ADA211344.

[14] Sprunt, B., Sha, L. and Lehoczky, J. P.
Aperiodic Task Scheduling for Hard Real-Time Systems.
The Journal of Real-Time Systems 1:27-60, 1989.

[15] Jay Kurt Strosnider.
Highly Responsive Real-Time Token Rings.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August, 1988.

Table of Contents

List of Figures