Aperiodic Task Scheduling for Real-Time Systems

Ph.D. Dissertation

Brinkley Sprunt

Department of Electrical and Computer Engineering
Carnegie Mellon University
August 1990

Ph.D. Dissertation
Submitted in Partial Fullfillment of the Requirements for the
Doctor of Philosohpy Degree in Computer Engineering

This work was done as part of the ART (Advanced Real-time Technologies) project, which is sponsored in part by the Office of Naval Research under contract N00014-84-K-0734, in part by the Naval Ocean System Center under contract N66001-87-C-0155, in part by Southeastern Center for Electrical Engineering Education, and in part by the Federal Systems Division of IBM Corporation under University Agreement YA-278067.

Abstract

This thesis develops the Sporadic Server (SS) algorithm for scheduling aperiodic tasks in real-time systems. The SS algorithm is an extension of the rate monotonic algorithm which was designed to schedule periodic tasks. This thesis demonstrates that the SS algorithm is able to guarantee deadlines for hard-deadline aperiodic tasks and provide good responsiveness for soft-deadline aperiodic tasks while avoiding the schedulability penalty and implementation complexity of previous aperiodic service algorithms. It is also proven that the aperiodic servers created by the SS algorithm can be treated as equivalently-sized periodic tasks when assessing schedulability. This allows all the scheduling theories developed for the rate monotonic algorithm to be used to schedule aperiodic tasks. For scheduling aperiodic and periodic tasks that share data, this thesis defines the interactions and schedulability impact of using the SS algorithm with the priority inheritance protocols. For scheduling hard-deadline tasks with short deadlines, an extension of the rate monotonic algorithm and analysis is developed. To predict performance of the SS algorithm, this thesis develops models and equations that allow the use of standard queueing theory models to predict the average response time of soft-deadline aperiodic tasks serviced with a high-priority sporadic server. Implementation methods are also developed to support the SS algorithm in Ada and on the Futurebus+.

Acknowledgments

Without the support of many people, this work would not have been possible. I am grateful to them all, not just for their technical support, but more importantly, for the experiences I have shared with them. I will remember these experiences always, for they are much more important to me than any of the work described in this thesis.

I would first like to thank the members of my committee: John Lehoczky, Lui Sha, Jay Strosnider, and Dan Siewiorek. My sincere thanks to John, Lui, and Jay for scooping me up a few years ago and introducing me to the ART project. I am especially grateful to John and Lui who helped me create my research project and guide it to completion. From them I learned how to conduct research and to always strive for a better solution. Special thanks to Jay, who began as an officemate, became a friend, and eventually my advisor. Finally, I gratefully thank Dan Siewiorek for taking me as his student under unusual circumstances, and for the helpful advice and direction he has given me.

I thank Craig Meyers for giving me a glimpse of the complexity of real-time systems and for arranging the funding for this work.

I am grateful to many members of the CMU community who have made this trip much more enjoyable than I could have hoped for. Thanks to Bob Colwell and Charlie Hitchcock for their encouragement and support when I was a green graduate student. Thanks to Andre van Tilborg for his advice and concern during a stressful time. Thanks to Ragunathan Rajkumar who was always ready to help me with my research problems. I envy his technical skill and understanding and I am grateful for his friendship. Special thanks to Tom Marchok who always managed to make me smile and remember the many things that are more important than my work. Thanks to the members of the ART project: Hide Tokuda, Cliff Mercer, and especially Joan Maddamma. Thanks to John Goodenough, Mark Borger, Mark Klein, Bob Page, and Tom Ralya of the SEI for all of the technical discussions and advice. Special thanks to the B52’s and the members of J-Team (Ron Mraz, C.J. Paul, Gowthami Rajendran, Sandy Ramos Thuel, George Sosnowski, and Dave Kirk) for their friendship and our many lively and amusing discussions. Thanks to my former officemates Ron Bianchini, Steve Kruse, Tim Stanley, and Dorothy Setliff for many adventures and unusual conversations.

Extra special thanks to the other vintage ’83 graduate students: Ron Bianchini, Beth Dirkes Lagnese, Frank Feather, Steve Kruse, Jim Quinlan, and Ted Lehr. It was a *very* good year.

I am grateful to my parents and family for always encouraging me to pursue my ideas and desires and for their unfailing support and love throughout this long journey that now seems to have passed too quickly.

Lastly, I am eternally grateful for the love and support of my wife Diane. She helped me endure the many frustrations associated with the completion of this work. But most of all, she makes me very happy.

Brinkley Sprunt
July 31, 1990

To my parents

1. Introduction

1.1 The Real-Time Scheduling Problem

Hard real-time systems are used to control physical processes that range in complexity from automobile ignition systems to controllers for flight systems and nuclear power plants. Such systems are referred to as *hard* real-time systems because the timing requirements of system computations must always be met or the system will fail. The scheduler for these systems must coordinate resources to meet the timing constraints of the physical system. This implies that the scheduler must be able to predict the execution behavior of all tasks within the system. This basic requirement of real-time systems is ***predictability***. Unless the behavior of a real-time system is predictable, the scheduler cannot guarantee that the computation deadlines of the system will be met.

It is the requirement of predictability that differentiates real-time systems from conventional computing environments and makes the scheduling solutions for conventional systems inappropriate for real-time systems. Conventional computing environments, such as time sharing systems, assume that tasks and their resource and computation requirements are not known in advance. As such, no effort is made in these systems to determine, *a priori*, the execution behavior of the tasks. It is also the case in these systems that an unknown waiting time for computation results is acceptable. None of these assumptions hold for a real-time system. For a real-time system to be predictable, all tasks and their resource and computation requirements must be known in advance. Also, for hard real-time systems, unknown delays waiting for computation results are unacceptable. A scheduling solution that captures the timing behavior of the system is needed for a real-time system.

The common scheduling solution for the predictability requirement of real-time systems is to use *cyclical executives* [Hood 86]. Cyclical executives provide a deterministic schedule for all tasks and resources in a real-time system by creating a static timeline upon which tasks and resources are assigned specific time intervals. This technique has been able to meet the timing requirements of many real-time systems. However, the cyclical executive approach has several serious drawbacks. Although a cyclical executive provides predictability through the use of a deterministic schedule, it is the requirement of determinism that is source of these drawbacks. A real-time system typically has many different timing constraints that must be met and no formal algorithm exists for creating timelines to meet these constraints. Therefore, each real-time system requires a different, handcrafted cyclical executive that requires extensive testing to verify its correctness. Also, because of the many *ad hoc* decisions made to create the timelines, changes to the system have unpredictable effects upon the correctness of the timelines and thus, a complete and new set of testing is required. Thus, the cyclical executive approach is typically expensive to create, verify, and update. As real-time systems become more complex, the problems associated with cyclical executives only become more difficult. A better scheduling solution is needed that provides predictability without the requirement of determinism.

An alternative solution to the real-time scheduling problem is to use priority-driven, preemptive scheduling algorithms to schedule tasks and resources. These algorithms assign priorities to each task in the system and always select the highest priority task to execute, preempting lower priority tasks as necessary. Such algorithms can describe the worst-case timing behavior of the system under all circumstances, and thus provide predictability by capturing the timing behavior of the system. Using well defined algorithms to schedule tasks and resources in a real-time system yields an understandable scheduling solution that is void of the *ad hoc* decisions typically found in cyclical executives. Thus, the resulting real-time system is easier to maintain and upgrade. Also, since the algorithmic approach accurately models the timing behavior of the system, the resulting schedule is easier to test. Because of these benefits, we will be investigating algorithmic based scheduling methods in this thesis as a means of creating efficient, predictable real-time systems.

1.2 Background and Related Research

A real-time task is generally placed into one of three categories based upon its arrival pattern and its deadline [Mok 83]. 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 it is desirable to meet a task's deadline but occasionally missing the deadline can be tolerated, then the deadline is considered to be *soft*. Tasks with regular arrival times are called periodic tasks. A common use of periodic tasks is to process sensor data and update the current state of the real-time system on a regular basis. Periodic tasks, typically used in control and signal processing applications, have hard deadlines. Tasks with irregular arrival times are aperiodic tasks. Aperiodic tasks are used to handle the processing requirements of random events such as operator requests. An aperiodic task typically has a soft deadline. Aperiodic tasks that have hard deadlines are called *sporadic* tasks. We assume that each task has a known worst case execution time. In summary, we have:

- **Hard and Soft Deadline Periodic Tasks.** A periodic task has a regular interarrival time equal to its period and a deadline that coincides with the end of its current period. Periodic tasks usually have hard deadlines, but in some applications the deadlines can be soft.
- **Soft Deadline Aperiodic Tasks.** An aperiodic task is a stream of jobs 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. Note that without a minimum interarrival time restriction, it is impossible to guarantee that a sporadic task's deadline would always be met.

To meet the timing constraints of the system, a scheduler must coordinate the use of all system resources using a set of well understood real-time scheduling algorithms that meet the following objectives:

- Guarantee that tasks with hard timing constraints will always meet their deadlines.
- Attain a high degree of schedulable utilization for hard deadline tasks (periodic and sporadic tasks). Schedulable utilization is the degree of resource utilization at or below which all hard deadlines can be guaranteed. The schedulable utilization attainable by an algorithm is a measure of the algorithm's utility: the higher the schedulable utilization, the more applicable the algorithm is for a range of real-time systems.
- Provide fast average response times for tasks with soft deadlines (aperiodic tasks).
- Ensure scheduling stability under transient overload. In some applications, such as radar tracking, an overload situation can develop in which the computation requirements of the system exceed the schedulable resource utilization. A scheduler is said to be stable if under

overload it can guarantee the deadlines of critical tasks even though it is impossible to meet all task deadlines.

The quality of a scheduling algorithm for real-time systems is judged by how well the algorithm meets these objectives.

1.2.1 Scheduling Approaches

The two main approaches for scheduling real-time systems fall into two categories: the use of heuristic scheduling algorithms for real-time systems in which the timing constraints of the system can change frequently or are unpredictable and the use of rigorous schedulability analysis coupled with predictable scheduling algorithms to guarantee performance for real-time systems in which the timing constraints do not change dynamically.

1.2.1.1 Heuristic Scheduling

Real-time scheduling algorithms for very dynamic systems, in which little can be said about task arrival patterns or expected computation load, typically use heuristics to schedule tasks. Obtaining optimal solutions for these systems is generally an NP-hard problem, so heuristics are used to create a schedule at runtime that attempts to maximize a particular scheduling metric. Examples of proposed heuristic approaches show that online scheduling can provide acceptable scheduling performance for dynamic real-time systems. One example of a scheduling heuristic for a distributed system is the guarantee routine [Ramamritham 84, Zhao 87a] that attempts to maximize resource utilization while meeting task deadlines most of the time. The guarantee routine accepts a task only if its deadline can be guaranteed locally without endangering the guarantees already granted for other tasks. If a task cannot be guaranteed, it is either terminated or sent to another processor as determined by a task distribution algorithm. The use of heuristics to preemptively schedule arbitrary task sets that share resources is addressed in [Zhao 87b]. This study shows that the application of simple heuristics with some backtracking performs very well. A different approach for scheduling under similar conditions is time-driven scheduling (TDS) [Tokuda 87]. The TDS model incorporates the timing criticality and semantic importance of each task into a value function. The goal of the TDS scheduler is to maximize the total value of completed tasks. While heuristic approaches are necessary for dynamic real-time systems, such as those that employ artificial intelligence techniques [Stankovic 88], these approaches do not provide a comprehensive characterization or guaranteed level of system performance.

1.2.1.2 Predictable Scheduling

An alternative to using heuristics is to employ algorithms that attempt to capture the timing behavior of all tasks in the system. The goal of these algorithms is to provide a high degree of schedulable resource utilization and an *a priori* verification of timing correctness for all tasks in the system. These algorithms are more restrictive than the heuristic methods in that they require more knowledge about task utilization, arrival patterns, and resource requirements. The well developed algorithms in this class are for the priority-driven, preemptive scheduling of periodic tasks with hard deadlines. These algorithms assign priorities based upon task deadlines (the deadline driven algorithm), available slack time (the least slack algorithm), or upon task frequency (the rate monotonic algorithm).

The deadline driven algorithm dynamically assigns priorities based upon the deadlines of the current tasks: the task with the closest deadline is given the highest priority and the task with the furthest deadline is given the lowest priority. Liu and Layland [Liu 73] have shown the deadline driven algorithm to be optimal in the sense that if a task set can be scheduled by any algorithm then it can be scheduled using the deadline driven algorithm. It was also shown that the deadline driven algorithm can achieve 100% processor utilization. Lawler and Martel [Lawler 81] have investigated the use of the deadline driven algorithm for multiprocessors. Despite the favorable qualities of the deadline driven algorithm, Leung and Merril [Leung 80] have shown that schedulability analysis for a task set scheduled on a single processor or multiprocessor systems using the deadline driven algorithm is NP-hard.

The least slack algorithm assigns priorities based upon the available slack time for each task. Slack time is defined to be the maximum amount of time a task can be delayed before it is certain to miss its deadline. The least slack algorithm gives highest priority to the task with the smallest slack. Mok [Mok 78] has shown that the least slack algorithm is also an optimal dynamic priority assignment algorithm. However, this algorithm can lead to an unbounded scheduling overhead and thrashing in the case when two or more tasks have similar slack times. Liu, Liu, and Liestman [Liu 82] have also investigated the use of slack time by developing lower bounds for the slack times of periodic jobs as a means of allowing an intelligent scheduler to allocate more execution time to certain tasks. One use of this extra time is to allow a task to provide higher quality results and still meet their deadlines. These ideas and goals have been extended to the general problem of scheduling periodic jobs using imprecise results by Liu, Lin, and Natarajan in [Liu 87].

A well understood scheduling algorithm for guaranteeing the hard deadlines of periodic tasks in a multiprogrammed real-time system is Liu and Layland's rate monotonic scheduling algorithm [Liu 73]. Under this algorithm, fixed priorities are assigned to tasks based upon the rate of their requests (i.e., a task with a relatively short period is given a relatively high priority). Liu and Layland proved that this algorithm is the optimum static preemptive scheduling algorithm for periodic tasks with hard deadlines. The algorithm guarantees that **n** periodic tasks can always be guaranteed to meet their deadlines if the resource utilization of the tasks is less than $\mathbf{n}(2^{1/\mathbf{n}}-1)$, which converges to ln 2 (= 69%) for large **n**. Although the rate-monotonic algorithm was originally developed for periodic tasks running on a single processor, it has been investigated as a tool for scheduling periodic tasks on multiprocessors [Dhall 78, Davari 86].

1.2.2 Scheduling Periodic Tasks

Although the predictable scheduling approach for real-time systems is less general than the heuristic scheduling approach, we have chosen to investigate predictable scheduling algorithms because hard real-time systems require *a priori* guarantees that all timing constraints can be met. We have chosen to use the rate monotonic algorithm as the basis for our scheduling algorithm research, because it has several favorable qualities for real-time systems:

- **High Schedulable Utilization.** Although Liu and Layland show a low scheduling bound for the rate monotonic algorithm, this bound is pessimistic and represents the absolute worst case conditions. Lehoczky, Sha, and Ding [Lehoczky 89] performed an exact characterization and stochastic analysis for a randomly generated set of periodic tasks scheduled by the rate monotonic algorithm and found that the average scheduling bound is usually much better than the worst case. They concluded that a good approximation to the threshold of schedulability for the rate monotonic algorithm is 88%. In fact, with the period transformation method, the utilization threshold can, in principle, be arbitrarily close to 100% [Sha 89a]. As an example of the high degree of schedulable utilization attainable with the rate monotonic algorithm, a schedulable utilization level of 99% was achieved for the Navy's Inertial Navigation System [Borger 87].

- **Stability Under Transient Overload.** Another concern for scheduling algorithms is transient overload, the case where stochastic execution times can lead to a desired utilization greater than the schedulable utilization bound of the task set. To handle transient overloads, Sha, Lehoczky, and Rajkumar describe a period transformation method for the rate monotonic algorithm that can guarantee that the deadlines of critical tasks can be met [Sha 86].
- **Aperiodic Tasks.** A real-time system often has both periodic and aperiodic tasks. Strosnider developed the *Deferrable Server* algorithm [Strosnider 88], which is compatible with the rate monotonic scheduling algorithm and provides a greatly improved average response time for soft deadline aperiodic tasks over polling or background service algorithms while still guaranteeing the deadlines of periodic tasks.
- **Resource Sharing.** Although determining the schedulability of a set of periodic tasks that use semaphores to enforce mutual exclusion has been shown to be NP-hard [Mok 83], Sha, Rajkumar, and Lehoczky [Rajkumar 89, Sha 87] have developed a priority inheritance protocol and derived a set of sufficient conditions under which a set of periodic tasks that share resources using this protocol can be scheduled using the rate monotonic algorithm.
- **Low Scheduling Overhead.** Since the rate monotonic algorithm assigns a static priority to each periodic task, the selection of which task to run is a simple function. Scheduling algorithms that dynamically assign priorities, may incur a larger overhead because task priorities have to be adjusted in addition to selecting the highest priority task to execute.

1.2.3 Scheduling Aperiodic Tasks

The scheduling problem for aperiodic tasks is very different from the scheduling problem for periodic tasks. Scheduling algorithms for aperiodic tasks must be able to guarantee the deadlines for hard deadline aperiodic tasks and provide good average response times for soft deadline aperiodic tasks even though the occurrences of the aperiodic requests are nondeterministic. The aperiodic scheduling algorithm must also accomplish these goals without compromising the hard deadlines of the periodic tasks.

Two common approaches for servicing soft deadline aperiodic requests are background processing and polling tasks. Background servicing of aperiodic requests occurs whenever the processor is idle (i.e. not executing any periodic tasks and no periodic tasks are pending). If the load of the periodic task set is high, then utilization left for background service is low, and background service opportunities are relatively infrequent. Polling consists of creating a periodic task for servicing aperiodic requests. At regular intervals, the polling task is started and services any pending aperiodic requests. However, if no aperiodic requests are pending, the polling task suspends itself until its next period and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is instead used by periodic tasks. Note that if an aperiodic request occurs just after the polling task has suspended, then the aperiodic request must wait until the beginning of the next polling task period or until background processing resumes before being serviced. Even though polling tasks and background processing can provide time for servicing aperiodic requests, they have the drawback that the average wait and response times for these algorithms can be long, especially for background processing.

Figures 1-2 and 1-3 illustrate the operation of background and polling aperiodic service using the periodic task set presented in Figure 1-1. The rate monotonic algorithm is used to assign priorities to the periodic tasks yielding a higher priority for task **A**. In each of these examples, periodic tasks **A** and **B** both become ready to execute at time = 0. Figures 1-2 and 1-3 show the task execution from time = 0 until time = 20. In each of these examples, two aperiodic requests occur: the first at time = 5 and the second at time = 12.

Periodic Tasks:

		Execution Time	Period	Priority
▢	Task A	**4**	**10**	**High**
▨	Task B	**8**	**20**	**Low**

Figure 1-1: Periodic Task Set for Figures 1-2, 1-3, 1-4, and 1-5

The response time performance of background service for the aperiodic requests shown in Figure 1-2 is poor. Since background service only occurs when the resource is idle, aperiodic service cannot begin until time = 16. The response time of the two aperiodic requests are 12 and 6 time units respectively, even though both requests each need only 1 unit of time to complete.

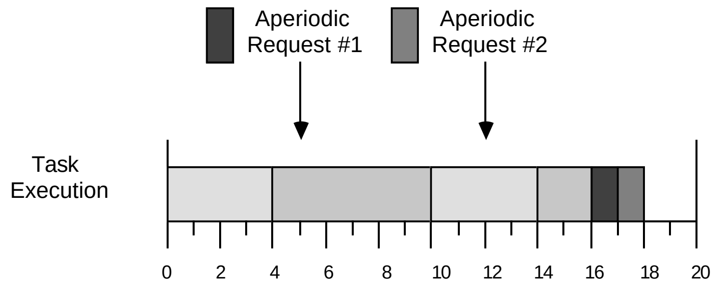


Figure 1-2: Background Aperiodic Service Example

The response time performance of polling service for the aperiodic requests shown in Figure 1-3 is better than background service for both requests. For this example, a polling server is created with an execution time of 1 time unit and a period of 5 time units which, using the rate monotonic algorithm, makes the polling server the highest priority task. The polling server's first period begins at time = 0. The lower part of Figure 1-3 shows the capacity (available execution time) of the polling server as a function of time. As can be seen from the upward arrow at time = 0 on the capacity graph, the execution time of the polling server is discarded during its first period because no aperiodic requests are pending. The beginning of the second polling period coincides with the first aperiodic request and, as such, the aperiodic request receives immediate service. However, the second aperiodic request misses the third polling period (time = 10) and must wait until the fourth polling period (time = 15) before being serviced. Also note, that since the second aperiodic request only needs half of the polling server's capacity, the remaining half is discarded because no other aperiodic tasks are pending. Thus, this example demonstrates how polling can provide an improvement in aperiodic response time performance over background service but is not always able to provide immediate service for aperiodic requests.

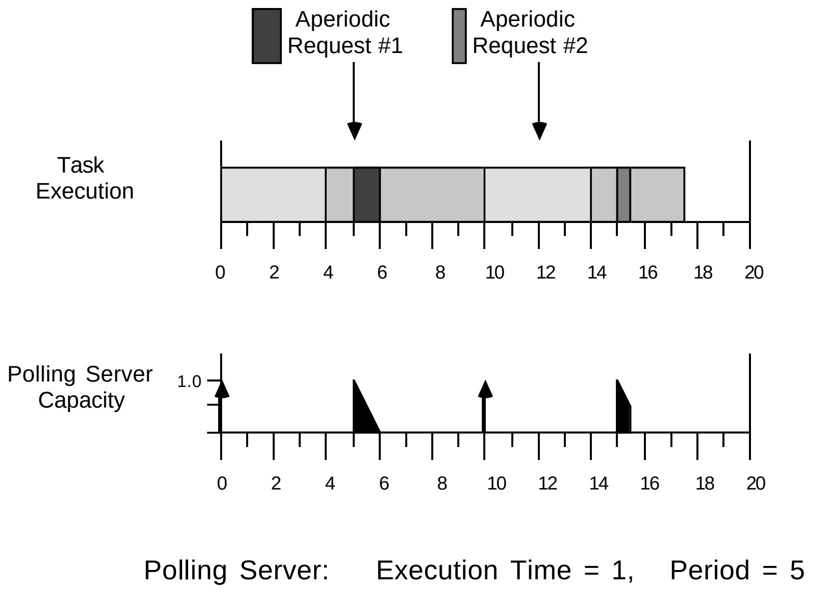


Figure 1-3: Polling Aperiodic Service Example

The *Priority Exchange* (PE) and *Deferrable Server* (DS) algorithms, introduced by Strosnider in [Strosnider 88], overcome the drawbacks associated with polling and background servicing of aperiodic requests. As with polling, the PE and DS algorithms create a periodic task (usually of a high priority) for servicing aperiodic requests. However, unlike polling, these algorithms will preserve the execution time allocated for aperiodic service if, upon the invocation of the server task, no aperiodic requests are pending. These algorithms can yield improved average response times for aperiodic requests because of their ability to provide immediate service for aperiodic tasks. In particular, the DS algorithm has been shown to be capable of providing an order of magnitude improvement in the responsiveness of asynchronous class messages for real-time token rings [Strosnider 88]. These algorithms are called *bandwidth preserving* algorithms, because they provide a mechanism for preserving the resource bandwidth allocated for aperiodic service if, upon becoming available, the bandwidth is not immediately needed. The PE and DS algorithms differ in the manner in which they preserve their high priority execution time.

The DS algorithm maintains its aperiodic execution time for the duration of the server's period. Thus, aperiodic requests can be serviced at the server's high priority at anytime as long as the server's execution time for the current period has not been exhausted. At the beginning of the DS's period, the server's high priority execution time is replenished to its full capacity.

The DS algorithm's method of bandwidth preservation is illustrated in Figure 1-4 using the periodic task set of Figure 1-1. For this example, a high priority server is created with an execution time of 0.8 time units and a period of 5 time units. At time = 0, the server's execution time is brought to its full capacity. This capacity is preserved until the first aperiodic request occurs at time = 5, at which point it is immediately serviced, exhausting the server's capacity by time = 6. At time = 10, the beginning of the third server period, the server's execution time is brought to its full capacity. At time = 12, the second aperiodic request occurs and is immediately serviced. Notice that although the second aperiodic request only consumes half the server's execution time, the remaining capacity is preserved, not discarded as in the polling example. Thus, the DS algorithm can provide better aperiodic responsiveness than polling because it preserves its execution time until it is needed by an aperiodic task.

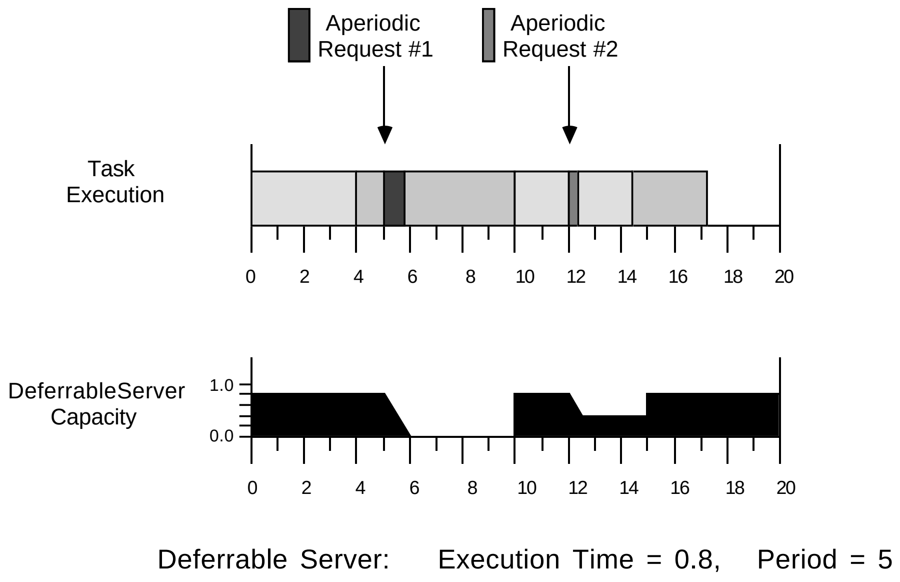


Figure 1-4: Deferrable Server Example

Unlike the DS algorithm, the PE algorithm preserves its high priority execution time by exchanging it for the execution time of a lower priority periodic task. At the beginning of the PE server's period, the server's high priority execution time is replenished to its full capacity. If the highest priority execution time available is aperiodic time (as is the case at the beginning of the PE server's period) and aperiodic tasks are pending, then the aperiodic tasks are serviced. Otherwise, the highest priority pending periodic task is chosen for execution and a priority exchange occurs. The priority exchange converts the high priority aperiodic time to aperiodic time at the assigned priority level of the periodic task. When a priority exchange occurs, the periodic task executes at the priority level of the higher priority aperiodic time, and aperiodic time is accumulated at the priority level of the periodic task. Thus, the periodic task advances its execution time, and the aperiodic time is not lost but preserved, albeit at a lower priority. Priority exchanges will continue until either the high priority aperiodic time is exhausted or an aperiodic request occurs in which case the aperiodic time is used to service the aperiodic request. Note that this exchanging of high priority aperiodic time for low priority periodic time continues until either the aperiodic time is used for aperiodic service or until the aperiodic time is degraded to the priority level of background processing (this complete degradation will occur only when no aperiodic requests arrive early enough to use the aperiodic time). Also, since the objective of the PE algorithm is to provide a low average response time for aperiodic requests, aperiodic requests win all priority ties. At all times, the PE algorithm uses the highest priority execution time available to service either periodic or aperiodic tasks.

The PE algorithm's method of bandwidth preservation is demonstrated in Figure 1-5 using the periodic task set of Figure 1-1. In this example, a high priority PE server is created with an execution time of 1 time unit and a period of 5 time units. Since the PE algorithm must manage aperiodic time across all priority levels, the capacity of the PE server as a function of time consists of three graphs: one for each priority level. The PE server's priority is priority level 1 which corresponds to the highest priority level followed by priority 2 for periodic task **A** and priority 3 for periodic task **B**. At time = 0, the PE server is brought to its full capacity, but no aperiodic tasks are pending and a priority exchange occurs between priorities 1 and 2. The PE server gains aperiodic time at priority 2 and periodic task **A** executes at priority 1. At time = 4, task **A** completes and task **B** begins. Since no aperiodic tasks are pending, another exchange takes place between priority 2 and priority 3. At time = 5, the server's execution time at priority 1 is brought to its full capacity and is used to provide immediate service for the first aperiodic request. At time = 10, the server's priority 1 execution time is brought to full capacity and is then exchanged down to priority 2. At time = 12, the server's execution time at priority 2 is used to provide immediate service for the second aperiodic request. At time = 14.5 the remaining priority 2 execution time is exchanged down to priority 3. At time = 15, the newly replenished server time at priority 1 is exchanged down to priority 3. Finally, at time = 17.5, the remaining PE server execution time at priority 3 is discarded because no tasks, periodic or aperiodic, are pending. Thus, the PE algorithm can also provide improved response times for aperiodic tasks compared to the polling algorithm. An enhancement of the PE algorithm, the Extended Priority Exchange Algorithm [Sprunt 88], has also been developed that takes advantage of the stochastic execution times of periodic tasks to provide more high priority execution time for aperiodic service.

The PE and DS algorithms differ in their complexity and in their effect upon the schedulability bound for periodic tasks. The DS algorithm is a much simpler algorithm to implement than the PE algorithm, because the DS algorithm always maintains its high priority execution time at its original priority level and never exchanges its execution time with lower priority levels as does the PE algorithm. However, the DS algorithm does pay a schedulability penalty (in terms of a lower schedulable utilization bound) for its simplicity. Both algorithms require that a certain resource utilization be reserved for high priority aperiodic service. We refer to this utilization as the server size, $\mathbf{U_s}$, which is the ratio of the server's execution time to the server's period. The server size and type (i.e. PE or DS) determine the scheduling bound for the periodic tasks, $\mathbf{U_p}$, which is the highest periodic utilization for which the rate monotonic algorithm can always schedule the periodic tasks. Below are the equations developed in [Lehoczky 87] for $\mathbf{U_p}$ in terms of $\mathbf{U_s}$ as the number of periodic tasks approaches infinity for the PE and DS algorithms:

(1)

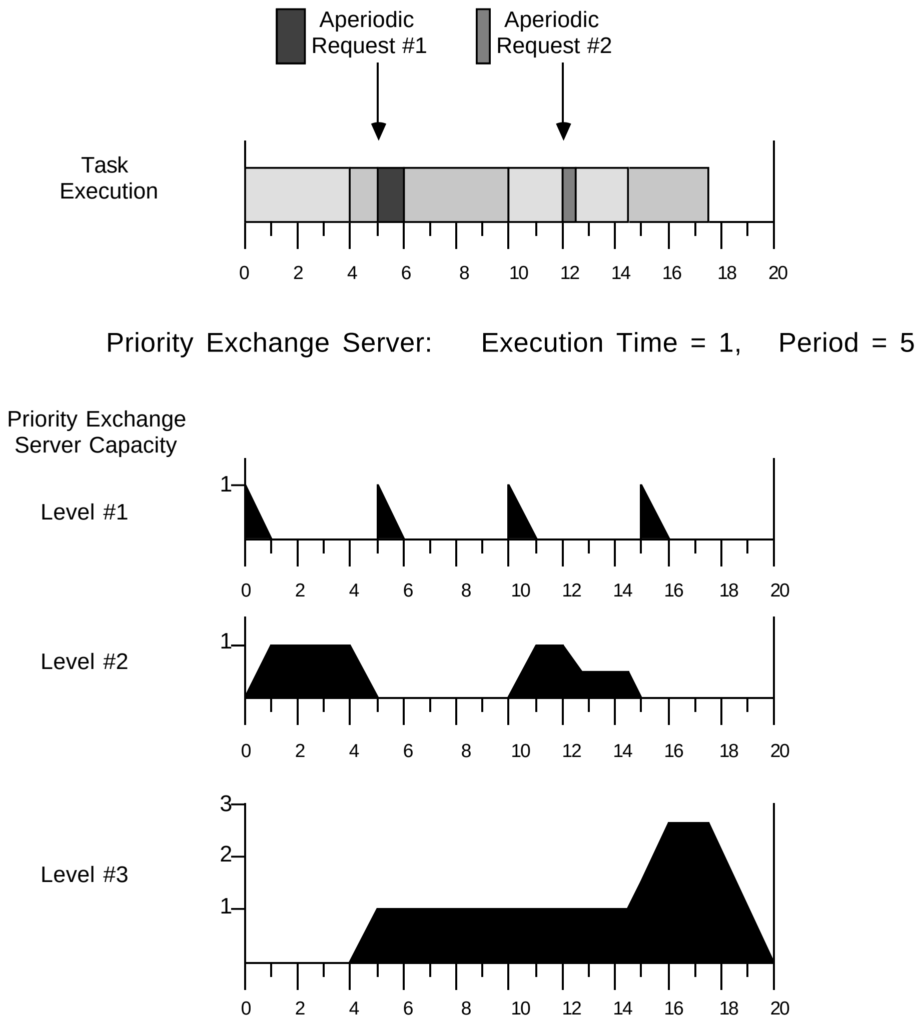


Figure 1-5: Priority Exchange Server Example

$$\textbf{PE:} \qquad \mathbf{U_p} = \ln \frac{2}{\mathbf{U_s}+1}$$

(2)

$$\textbf{DS:} \qquad \mathbf{U_p} = \ln \frac{\mathbf{U_s}+2}{2\mathbf{U_s}+1}$$

Equations 1 and 2 show that for a given server size, $\mathbf{U_s}$ ($0 < \mathbf{U_s} < 1$), the periodic schedulability bound, $\mathbf{U_p}$, for the DS algorithm is lower than it is for the PE algorithm. These equations also imply that for a given periodic load, the server size, $\mathbf{U_s}$, for the DS algorithm is smaller than that for the PE algorithm. For example, with $\mathbf{U_p}$ = 60%, Equation 1 indicates a server size for the PE algorithm of 10% compared to a server size for the DS algorithm of 7% given by Equation 2.

1.2.4 Research Scope and Goals

Although the Deferrable Server and Priority Exchange algorithms do significantly improve aperiodic responsiveness, they have limitations, and several important scheduling issues for aperiodic tasks are not addressed by these algorithms. The research presented in this thesis develops new algorithms that overcome the limitations of these previous algorithms and provides solutions for several aperiodic scheduling problems not addressed by these algorithms. This section discusses the research scope of this thesis.

By comparing the PE and DS algorithms, we can identify the relative merits of each. The advantage of the DS algorithm over the PE algorithm is that it is conceptually much simpler and, thus, easier to implement. The PE algorithm must manage aperiodic time across all priority levels in the system, whereas the DS algorithm maintains its execution time at its original priority level. The simple bandwidth preservation technique of the DS algorithm also implies that, on average, the priority of the server's execution time will be higher than it would be for an equivalently sized PE server. In contrast, the PE algorithm must either use its high priority execution time for aperiodic service or trade it for lower priority periodic time. However, the PE algorithm has a server size advantage over the DS algorithm. This advantage of the PE algorithm is even greater when multiple servers are executing at different priority levels. This thesis develops the Sporadic Server (SS) algorithm that retains the advantages of these previous algorithms while overcoming their limitations.

One important class of aperiodic tasks that is not generally supported by previous aperiodic service algorithms are sporadic tasks (hard deadline aperiodic tasks with a specified minimum interarrival time and worst-case execution time). To guarantee hard deadlines for sporadic tasks, a high-priority aperiodic server can be created with enough execution time to guarantee that a sporadic task can meet its hard deadline. However, the minimum interarrival time for the sporadic task must be equal to or greater than the period of its server. As long as the sporadic task does not request service more frequently than the minimum interarrival time or consume more than its worst-case execution time, its hard deadline can be guaranteed. However, this guarantee can only be made if the sporadic task's deadline is equal to or greater than its minimum interarrival time. This thesis develops a scheduling approach and schedulability analysis technique that can guarantee deadlines for short-deadline sporadic tasks.

The development of priority inheritance protocols have provided a good solution to the problem of scheduling periodic tasks that share data [Rajkumar 89, Sha 87]; however, the interaction of these protocols with aperiodic tasks and their servers has not been defined. When tasks share data, blocking occurs whenever a lower priority task prevents a higher priority task from executing. The priority inheritance protocols limit the amount of blocking a task can experience when attempting to access a shared resource and, therefore, allow a much higher degree of schedulable utilization and greatly simplify the schedulability analysis. The basic mechanism used by the priority inheritance protocols to limit blocking is to increase temporarily the priority of the task causing the blocking. Aperiodic service algorithms also temporarily increase the priority of low priority aperiodic tasks in order to provide responsive service. This thesis develops a coherent strategy for coordinating these manipulations of priority by the priority inheritance protocols and aperiodic service algorithms. This thesis also develops the associated schedulability analysis techniques for periodic and aperiodic tasks that share data.

The general goal of the Advanced Real-time Technology group at Carnegie Mellon University is to develop scheduling theories and analytical tools that allow real-time system designers to determine the schedulability of their application and examine various scheduling/performance tradeoffs [Tokuda 88]. The current technique for determining the expected response time for aperiodic requests is simulation. Although simulation can provide a good characterization of average aperiodic response time, it is a tedious and error prone process. This thesis develops analytical approaches for estimating the expected response time performance for aperiodic tasks serviced with or without aperiodic servers.

An important aspect of any scheduling algorithm for real-time systems is its implementability for real-time system platforms. This thesis develops implementations for the SS algorithm in Ada, a programming language designed for real-time applications [Ada 83]. Two SS implementations for Ada are developed: (1) a partial implementation using an Ada task which requires no modifications to the Ada runtime system and (2) a complete implementation within the Ada runtime system. An implementation approach to support the SS algorithm for high-priority aperiodic transfers on FutureBus+ [FutureBus 89] is also developed.

1.2.5 Thesis Organization

This thesis is organized as follows. In Chapter 2, the Sporadic Server algorithm is defined and examples are given of its operation. The expected performance improvement of the SS algorithm over the polling algorithm is explained. The schedulability analysis of aperiodic servers is discussed. It is proved that a sporadic server can be treated as an equivalently-sized periodic task for schedulability analysis. The operation of the SS algorithm for sporadic tasks and the necessity of the deadline monotonic priority assignment for short-deadline sporadic tasks is discussed and the associated schedulability analysis defined. This chapter closes with a discussion of sporadic servers and the priority inheritance protocols. The schedulability implications of using sporadic servers to service aperiodic tasks that share data with periodic tasks is also discussed and the necessary schedulability analysis is defined.

In chapter 3, the results of a simulation study of the Background, Polling, Deferrable Server, Priority Exchange, and Sporadic Server algorithms is presented. The response time performance and context swap overhead of each algorithm is compared for a set of randomly generated task sets over a significant range of periodic and aperiodic loads. The Complete-Service policy for aperiodic service is defined and shown to reduce context swap overhead and, in many cases, improve response time performance. An investigation of the performance of sporadic servers as a function of priority level is presented. A simplification of the SS algorithm that reduces the complexity and overhead of its implementation is defined and its effect upon performance is presented. The effect of priority and server period upon server size and the maximum server execution time is discussed and demonstrated for several task sets. The average response time performance of the SS algorithm as a function of the server period and priority is investigated. This chapter closes with a section describing guidelines for choosing a sporadic server's period, execution time, and priority for a given application.

Chapter 4 presents approaches for modeling the average aperiodic response time performance. An approach for modeling the average aperiodic queue length for background service is discussed. A method for determining the conditions under which the DS and SS algorithms can effectively eliminate the effects of the periodic task set in terms of average aperiodic response time is defined and demonstrated.

Chapter 5 investigates software and hardware implementations of sporadic servers. An application-level Ada task implementation and a full Ada runtime implementation are presented. An implementation for a shared sporadic server on the Futurebus+ is also presented.

Finally, Chapter 6 summarizes the research contributions of this work and suggests directions in which this work can be extended.

2. The Sporadic Server Algorithm

2.1 Sporadic Server Algorithm Definition

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 [Lehoczky 87, Sprunt 89a] 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, **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.

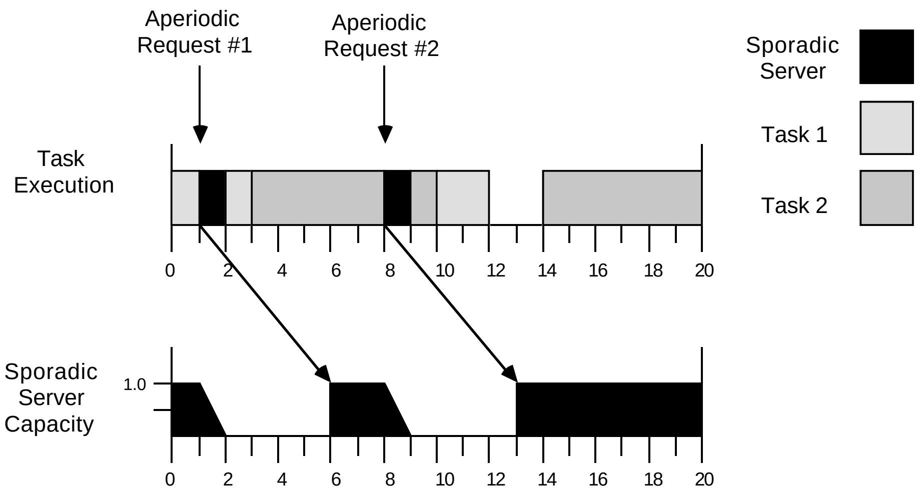
2.1.1 SS Algorithm Examples

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 2-1, 2-2, 2-3, and 2-4 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 2-1 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.

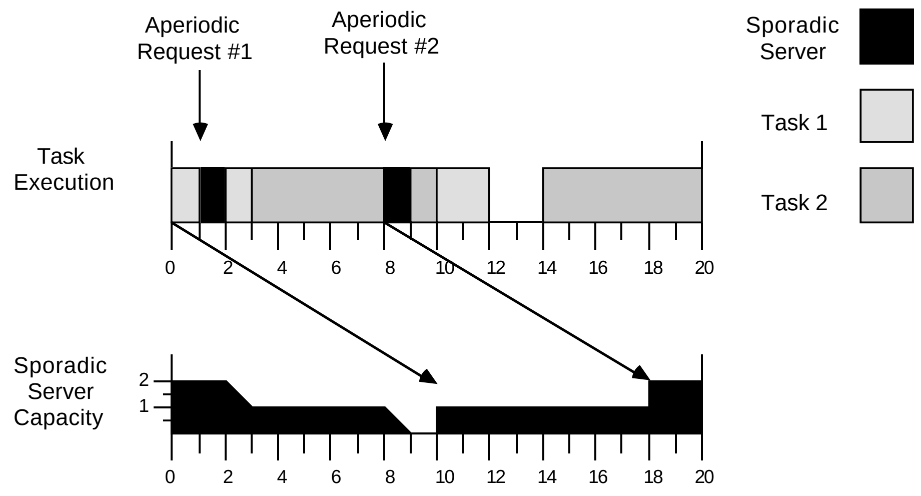
The task execution in Figure 2-1 proceeds as follows. For this example, the sporadic server begins with its full execution time capacity. At t = 0, τ_1 begins execution. At time = 1, the first aperiodic request occurs and is serviced by the sporadic server. Priority level $\mathbf{P}_1$ has become active 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 2-1 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 2-1 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 2-2 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,



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

Figure 2-1: High Priority Sporadic Server Example

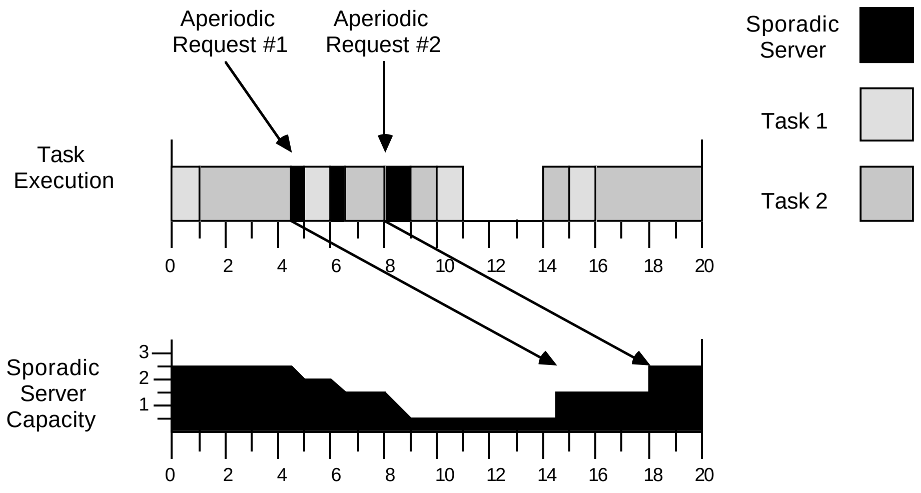


Task	Exec Time	Period	Utilization
SS	2	10	20.0%
τ_1	2	10	20.0%
τ_2	6	14	42.9%

Figure 2-2: Equal Priority Sporadic Server Example

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 2-2 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 2-2 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.



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%

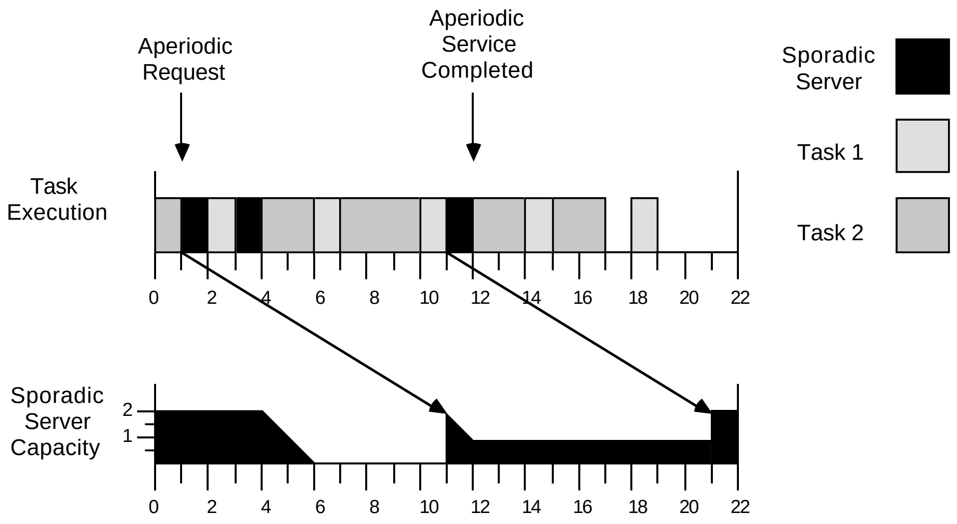
Figure 2-3: Medium Priority Sporadic Server Example

Figure 2-3 shows the task execution and the task set characteristics for the medium priority 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 occurs 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 preempts 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 2-3 illustrates another important property of the sporadic server algorithm. Even if the sporadic server is preempted and provides discontinuous service for an aperiodic request (as occurs with the first aperiodic request in Figure 2-3), only one replenishment is necessary. Preemption 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 replenished together. Note that one replenishment for the consumption of sporadic server execution time resulting from both aperiodic requests in Figure 2-3 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 2-4, illustrates the application of the replenishment rules stated in Section 2.1 for a case when the sporadic server's execution time is exhausted. Figure 2-4 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 2-4, 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	Exec Time	Period	Utilization
τ_1	1	4	25%
SS	2	10	20%
τ_2	10	40	25%

Figure 2-4: Exhausted Sporadic Server Replenishment

Task execution in Figure 2-4 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 preempts τ_2 and begins servicing the aperiodic request. While the aperiodic request is being serviced, τ_1 becomes ready at t = 2 and preempts the sporadic server. At t = 3, τ_1 completes execution and servicing of the aperiodic request continues. At t = 4, the sporadic 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 preempts τ_2 and executes until t = 7 when τ_2 resumes execution. At t = 10, τ_1 again preempts 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 t = 11, the point at which the sporadic server's capacity became greater than zero.

2.2 Sporadic Server and Polling Server Performance

In this section, we investigate the ability of a sporadic server to provide better aperiodic response time performance than an equivalent polling server. First, we explain and demonstrate the advantage of sporadic servers over polling servers. We then prove that a sporadic server always provides high-priority aperiodic service at times equal to or earlier than a polling server and argue that this will usually yield a better average aperiodic response time. However, we also present an example to demonstrate that a sporadic server cannot always provide a better response time than polling.

The advantage of a sporadic server over a polling server is that the sporadic server is often able to provide immediate service to an aperiodic request. A sporadic server accomplishes this by preserving its execution time until an aperiodic request occurs at which point aperiodic service is provided immediately. In contrast, a polling server does not preserve its execution time if no aperiodic tasks are pending. The only time a polling server provides service to aperiodic tasks occurs when aperiodic tasks are pending at polling instants (a polling instant corresponds to the beginning of the next polling period). If aperiodic tasks are pending at a polling instant, then the polling server provides service until either no aperiodic tasks are pending or it has exhausted its capacity. Otherwise, the polling server discards its capacity and suspends high-priority aperiodic service until the next polling instant. As such, aperiodic tasks that are serviced with a polling server often have to wait until the next polling instant to begin service.

Figure 2-5 presents two examples that show the advantage of a sporadic server over a polling server. In the examples, both periodic tasks become ready to execute at time = 0. Referring to the polling server example in Figure 2-5, we see that the arrival of the aperiodic requests have missed the polling instant at time = 0. This forces aperiodic requests #1 and #2 to wait until both periodic tasks complete execution (and the resource is idle) before aperiodic service begins. At time = 10, the service for the second aperiodic request would normally be preempted by the highest priority periodic task, but the polling server activates and completes aperiodic requests #2 and #3. The average response time for these three aperiodic requests is 5.67 units of time.

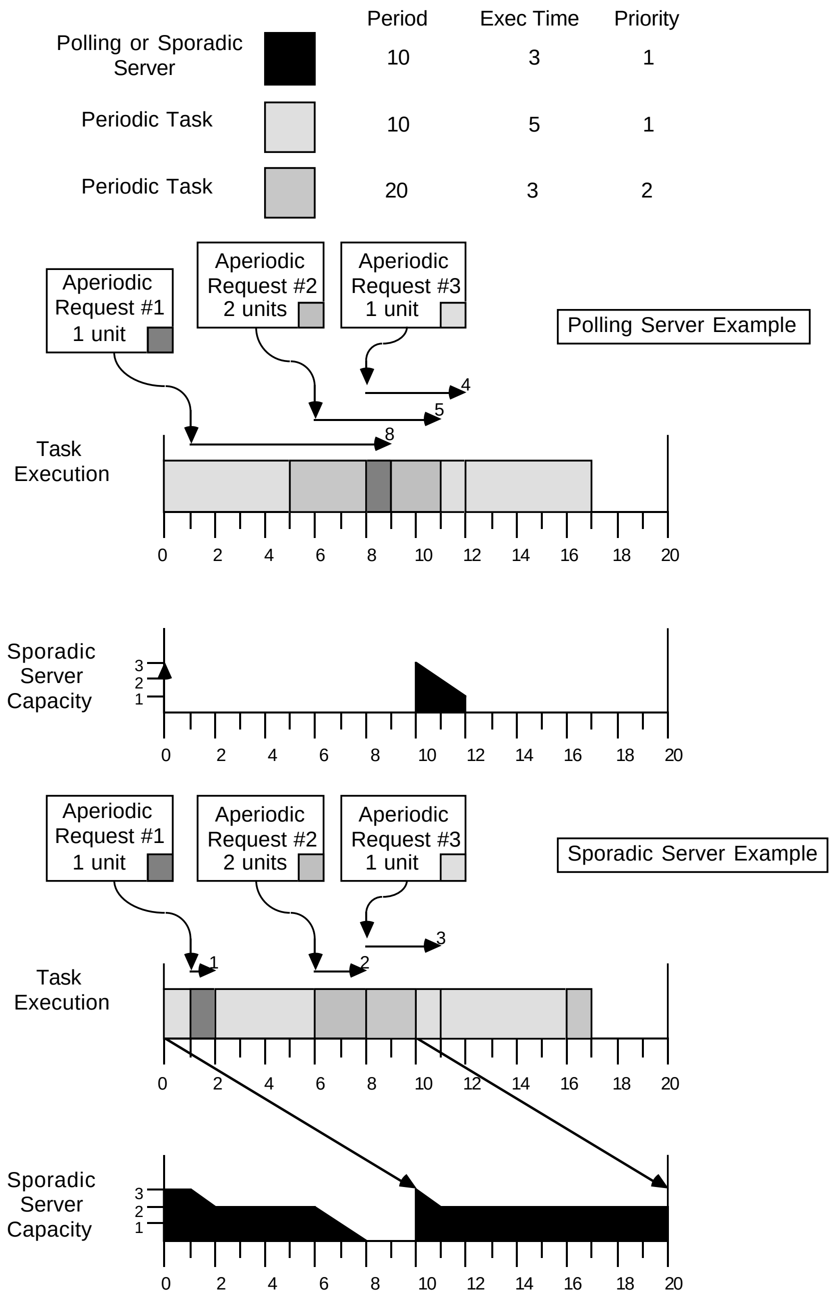


Figure 2-5: Example of a Sporadic Server Providing Better Aperiodic Response Times than a Polling Server

The sporadic server example in Figure 2-5 shows a much better aperiodic response time performance than the polling example. Since the sporadic server preserves its execution time until needed by an aperiodic request, aperiodic requests #1 and #2 are immediately given service when they arrive. However, because the sporadic server exhausts its execution time to finish aperiodic request #2, aperiodic request #3 must wait until the sporadic server's execution time is replenished at time = 10 before being serviced. The average response time for the three aperiodic requests is 2 units of time, a substantial improvement over polling.

We can also demonstrate the advantage of the sporadic server with a proof that a sporadic server always provides high-priority aperiodic service at times equal to or earlier than a polling server having the same period, execution time and priority. The theorem and proof are as follows.

Theorem 1: Given a real-time system composed of soft-deadline aperiodic tasks and hard-deadline periodic tasks, let the soft-deadline aperiodic tasks be serviced by a polling server that starts at full capacity and executes at the priority level of the highest priority periodic task. If the polling server is replaced with a sporadic server having the same period, execution time and priority, the sporadic server will provide high-priority aperiodic service at times earlier than or equal to the times the polling server would provide high-priority aperiodic service.

Proof: Since the only time a polling server can begin to provide aperiodic service occurs at polling instants, the behavior of a similarly equipped sporadic server only needs to be considered at the corresponding polling instants. With respect to polling instants, two cases must be considered:

1. The sporadic server is at full capacity at the polling instant and aperiodic work is pending.
2. The sporadic server has less than its full capacity at the polling instant and aperiodic work is pending.

Case 1: Since both the polling server and the sporadic server have the same capacity at the same instant in time, the same amount of pending aperiodic work will be serviced in the same amount of time by both servers. In this case, the sporadic server begins providing high priority aperiodic service at the same time and for the same duration as the polling server.

Case 2: Since the sporadic server is not at full capacity, it must have provided aperiodic service in the recent past which depleted some or all of its capacity. By the definition of the sporadic server algorithm, the longest interval of time between the consumption of sporadic server execution time and its replenishment is one sporadic server period. Therefore, the sporadic server must have provided service to aperiodic tasks after the last polling instant and before the current polling instant (otherwise, the sporadic server would be at full capacity). Therefore, in this case, the sporadic server has provided high priority aperiodic service at an earlier time than the polling server.

Since, in both cases, a sporadic server provides high priority aperiodic service at times earlier or equal to the times at which a polling server does, the Theorem follows.

Although, it is the ability of the sporadic server to provide high priority service earlier than the polling server that enables the sporadic server to provide better aperiodic responsiveness, it is not the case that a sporadic server will always provide better aperiodic responsiveness than a polling server. Figure 2-6 shows an example in which a polling server provides a better response time than a sporadic server. As in Figure 2-5, both periodic tasks become ready to execute at time = 0 for the examples in Figure 2-6.

The polling server example in Figure 2-6 shows the first aperiodic request arriving at time = 11.5, missing the polling instant at time = 10. However, between time = 12 and time= 20, no periodic tasks are executing and the first aperiodic request can be serviced with an initial delay of only 0.5 units of time. This execution is noted in the example as "Background Service" because the polling server is not providing high-priority aperiodic service over this interval. The first aperiodic task completes at time = 20. The second aperiodic request arrives at time = 18 and is completed by the polling server at time = 22. The average aperiodic response time for these two aperiodic tasks is 6.25 units of time.

The sporadic server example in Figure 2-6 shows the first aperiodic task receiving immediate service upon its arrival at time = 12. The immediate service preempts the execution of the low priority periodic task. The sporadic server has used its high-priority execution time to provide a better response time for the first aperiodic request (8 units compared to 8.5 units for the polling server). However, this service exhausts the sporadic server and its execution time is scheduled for replenishment at time = 21.5. This replenishment is too late to enable the second aperiodic request to receive high-priority service from the sporadic server. The second aperiodic request must wait for the completion of the low priority periodic task at time = 20 and the completion of the high priority periodic task at time = 22 before beginning service at time = 21. The average response time for these two aperiodic requests is 6.5 units of time which is less than the average response time of 6.25 units for the polling server.

From these examples, one can see that if an aperiodic task arrives in an interval of time during which the periodic tasks will be executing for a substantial amount of time, the ability of the sporadic server to preempt the periodic tasks and immediately service the aperiodic request can significantly improve the average response time. However, one can also see that providing immediate, high-priority service does not necessarily improve the average aperiodic response time.

2.3 The Schedulability of Sporadic Servers

In this section we prove that, from a scheduling point of view, a sporadic server can be treated as a standard periodic task with the same period and execution time as the sporadic server. It is necessary to prove this claim because the sporadic server violates one of the basic assumptions governing periodic task execution as described by Liu and Layland [Liu 73] in their analysis of the rate monotonic algorithm. Given a set of periodic tasks that is schedulable by the rate monotonic algorithm, this assumption requires that, once a periodic task is the highest priority task that is ready to execute, it must execute.* If a periodic task *defers* its execution when it otherwise could execute immediately, then it may be possible that a lower priority task will miss its deadline even if the set of tasks was schedulable. A sporadic server does not meet this Liu and Layland requirement because even though it may be the highest priority task that is ready to execute, it will preserve its execution time if it has no pending requests for aperiodic service. This preservation of the server's execution time is equivalent to deferring the execution of a periodic task. A deferrable server also fails to meet the above requirement because of its preservation capabilities. To prove that a sporadic server can still be treated as a normal periodic task, we will show that the sporadic server's replenishment method compensates for any deferred execution of the sporadic server. In contrast, we will also show how the replenishment method for a deferrable server fails in this respect.

To demonstrate how deferred execution can cause a lower priority task to miss its deadline, the execution of a periodic task will be compared to the execution of a deferrable server with the same period and execution time. Figure 2-7 presents the execution behavior of three periodic tasks. Let T represent the period of a periodic task and C represent its execution time. Task A, with $T_A = 4$ and $C_A = 1$, has the highest priority. Task B, with $T_B = 5$ and $C_B = 2$, has a medium priority. Task C, with $T_C = 10$ and $C_C =$

*For the case when two or more periodic tasks of equal priority are the highest priority tasks that are ready to execute, the requirement is that one of these tasks must execute.

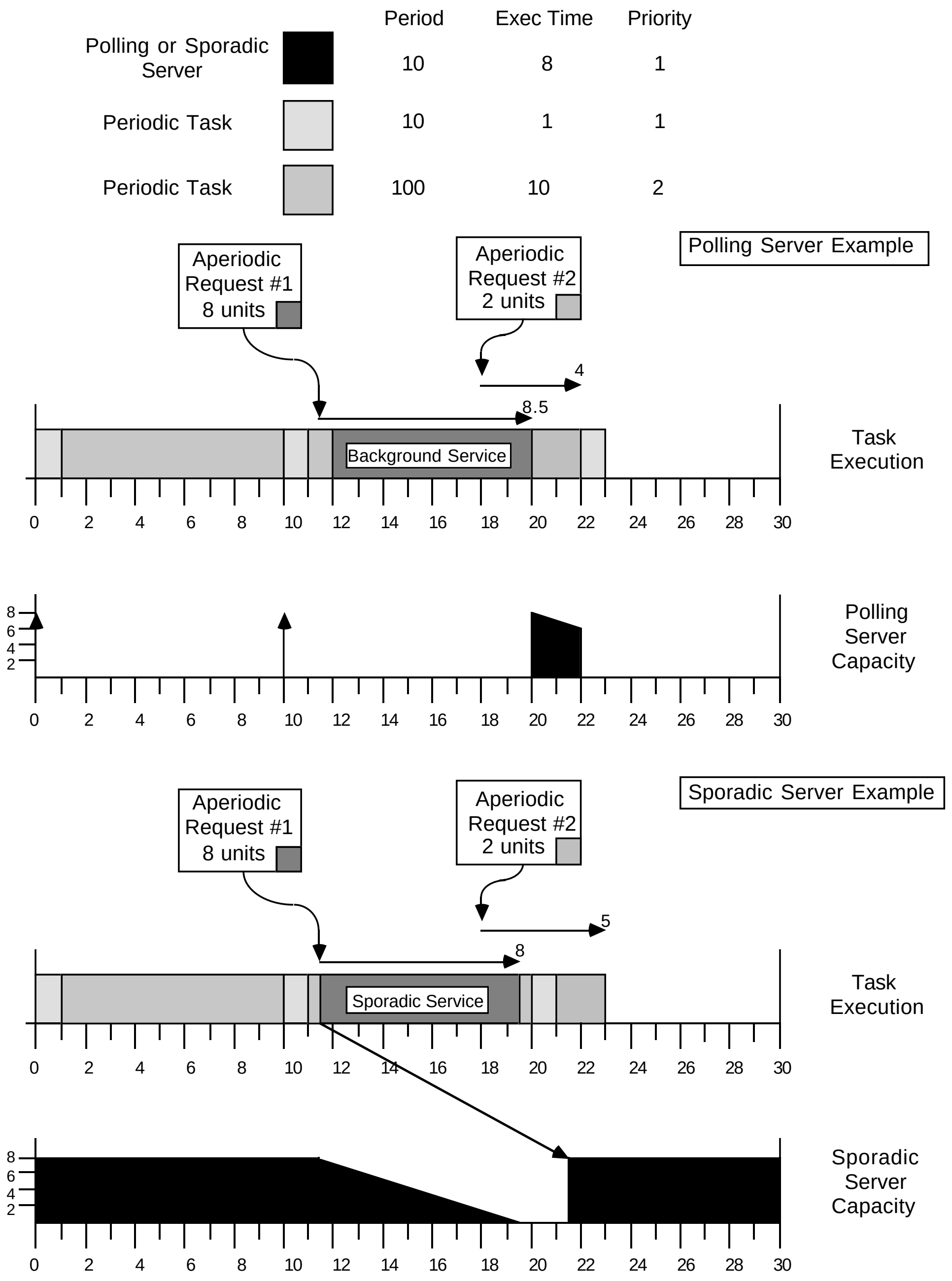


Figure 2-6: Example of a Sporadic Server Providing A Worse Aperiodic Response Time than a Polling Server

3, has the lowest priority. Task A and B begin their first periods at time = 0 and Task C begins its first period at time = 3. Note that no idle time exists during the first period of Task C. This constrains Task C to a maximum execution time of 3 units.

Figure 2-8 presents the execution of the task set of Figure 2-7 with Task B replaced with a deferrable server. The capacity of the deferrable server as a function of time is also presented in Figure 2-8. For this example, the following aperiodic requests are made to the deferrable server:

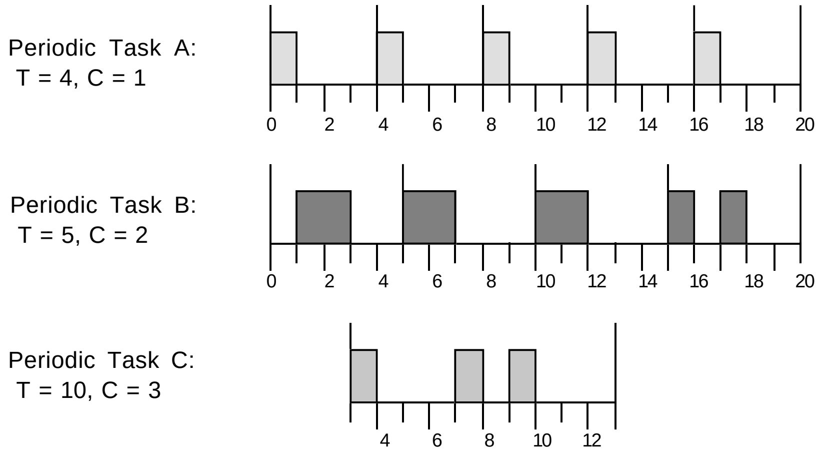


Figure 2-7: Periodic Task Example

Request	Request Instant	Exec Time
1	1	1
2	3	1
3	5	2
4	10	2

Referring to the deferrable server execution graph in Figure 2-8, at time = 2 the service for the first aperiodic request completes and no other aperiodic requests are pending. At this point, the deferrable server *defers* its execution by preserving its remaining execution time until the second aperiodic request occurs at time = 3. Note that this deferred execution followed by the servicing of the second aperiodic request from time = 3 to time = 4 has blocked Task C from executing during this interval, whereas during the same interval in Figure 2-7 Task C was not blocked. The third and fourth aperiodic requests are executed by the deferrable server during the same intervals as Task B executes in Figure 2-7. Thus, Task A and the deferrable server limit Task C to a maximum of 2 units of execution time per period, whereas the original periodic task was able to complete 3 units of computation during the same period in Figure 2-7. If Task C had needed 3 units of execution time during the period shown in Figure 2-8, it would have missed its deadline. It is this *invasive* quality of the deferrable server for lower priority periodic tasks that results in the lower scheduling bound for the DS algorithm described in Section 1.2.3.

Figure 2-9 presents the execution of the task set of Figure 2-7 with Task B replaced by a sporadic server. The third timeline in Figure 2-9 is the capacity of the sporadic server as a function of time. The arrows in Figure 2-9 indicate the replenishment of consumed sporadic server execution time. The requests for aperiodic service are the same as the requests of Figure 2-8. Note that the sporadic server, like the deferrable server in Figure 2-8, blocks the execution of Task C during time = 3 to time = 4. However, the sporadic server replenishment method prevents the execution time consumed during this interval from being replenished until time = 8. This allows Task C to execute from time = 6 to time = 7, whereas the deferrable server was executing during this interval in Figure 2-8. The sporadic server, unlike the

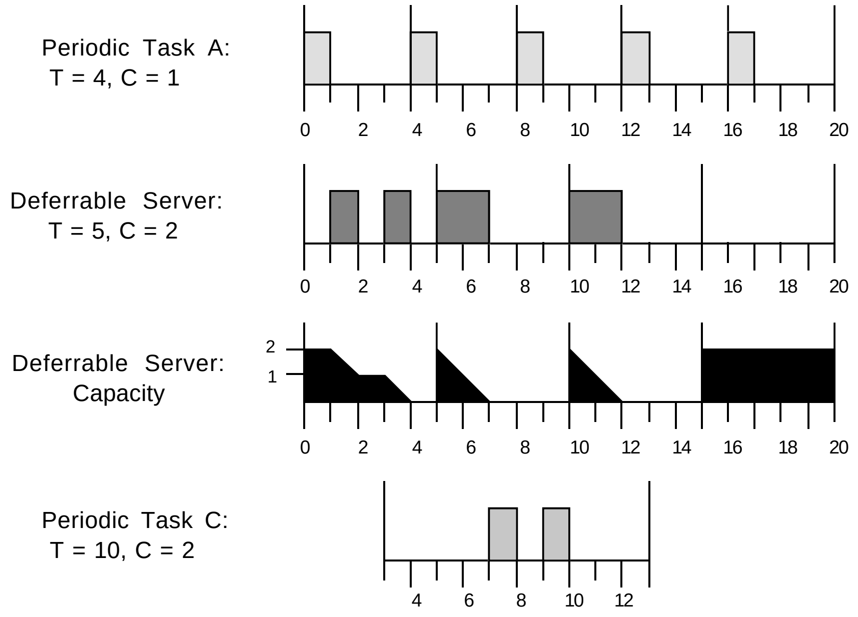


Figure 2-8: An Example of Deferred Execution with the DS Algorithm

deferrable server, blocks Task C from executing from time = 9 to time = 10. However, the replenishment of the server execution time consumed during this interval does not occur until time = 14. This allows Task C to complete its execution from time = 11 to time = 12. Thus, in this example, the sporadic server allows Task C to meet its deadline whereas a deferrable server having the same period and execution time cannot. However, one should note that the sporadic server completes the third and fourth aperiodic requests 3 time units later than the deferrable server. Thus, for the same series of aperiodic requests a sporadic server may provide a longer response time than an equivalently sized deferrable server. Chapter 3 presents a simulation study that investigates the relative performance of these algorithms.

Now that we have shown a specific example of how the replenishment method of the SS algorithm can compensate for deferred execution of the sporadic server, we need to prove that, in terms of schedulability, a sporadic server can always be treated as a periodic task having the same period and execution time. To do this we first show that a sporadic server can behave in a manner identical to a periodic task with the same period and execution time. Next we show that any execution of a sporadic server before the server's priority level becomes idle, falls into one of three cases: 1) none of the server's execution time is consumed, 2) the server's execution time is completely consumed, or 3) the server's execution time is only partially consumed. In the first case, the server can be shown to be equivalent to a periodic task with a delayed request. In the second case, the server's execution is shown to be identical to that of a normal periodic task. In the third case, the execution behavior of the server is shown to be equivalent to two periodic tasks: one that executes normally and one that is delayed. Thus, all types of sporadic server execution are shown to be equivalent to the execution of one or more periodic tasks.

In order to explore the schedulability effects of sporadic servers upon a periodic task set, several terms and concepts developed by Liu and Layland in [Liu 73] are needed. A periodic task set is composed of n

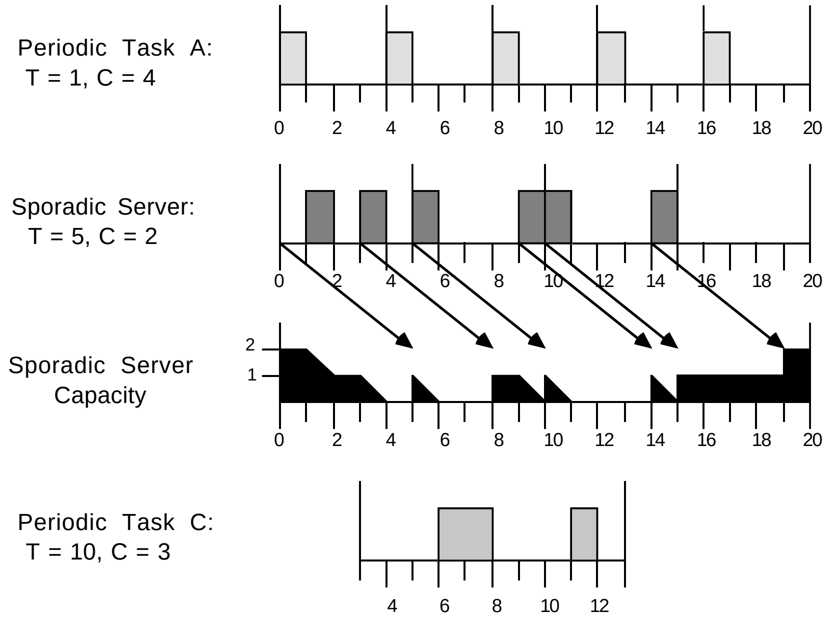


Figure 2-9: An Example of Deferred Execution with the SS Algorithm

independent tasks, τ_1 through τ_n, numbered in order of decreasing priority. A periodic task, τ_i is characterized by its computation time, C_i, and a period, T_i. At all times, the highest priority task that is ready to execute is selected for execution, preempting any lower priority tasks as necessary. The deadline of a periodic task occurs one period after it is requested. The *critical instant* for a periodic task is the instant at which a request for that task will have the longest response time. Liu and Layland [Liu 73] state and prove the following theorem concerning the critical instant of a periodic task:

Theorem 2: Given a set of periodic tasks, the critical instant for any task occurs whenever the task is requested simultaneously with all higher priority tasks.

The *critical zone* for a periodic task is the time interval between its critical instant and the completion of that request. A periodic task set is schedulable if the critical zone for each task is less than or equal to its deadline.

We now state and prove the following property of schedulable periodic task sets:

Lemma 3: Given a periodic task set that is schedulable with τ_i, the task set is also schedulable if τ_i is split into k separate periodic tasks, $\tau_{i,1}, \ldots, \tau_{i,k}$, with $\sum_{j=1}^{k} C_{i,j} = C_i$ and $T_{i,j} = T_i$ for $1 \le j \le k$.

Proof: By the rate monotonic algorithm, all tasks $\tau_{i,1}, \ldots, \tau_{i,k}$ will be assigned the same priority because they have the same period. The rate monotonic algorithm schedules a periodic task set independently of the phasing between any two tasks. Suppose the requests for tasks $\tau_{i,1}, \ldots, \tau_{i,k}$ are all in sync with each other. The execution pattern of $\tau_{i,1}, \ldots, \tau_{i,k}$ will now be identical to the execution pattern of the original task, τ_i, and therefore the task set is still schedulable for this phasing of $\tau_{i,1}, \ldots, \tau_{i,k}$. Since the task set is schedulable for one phasing of $\tau_{i,1}, \ldots, \tau_{i,k}$, it is schedulable for them all. The Lemma follows.

Next we establish that the execution behavior of a sporadic server can be identical to that of a periodic task with the same period and execution time.

Lemma 4: Given a schedulable periodic task set, replace a periodic task with a sporadic server having the same period and execution time. If the requests for the sporadic server are identical to that of the original periodic task, then the execution behavior of the sporadic server is identical to that of the original periodic task.

Proof: The periodic task and the sporadic server execute at the same priority level and each request for the sporadic server and the periodic task require the same execution time. Each request for the sporadic server completely consumes its execution time and the consumed execution time is replenished before the next request. Therefore, the sporadic server's execution is identical to that of the original periodic task.

We now show that sporadic servers are equivalent to periodic tasks in terms of schedulability.

Theorem 5: A periodic task set that is schedulable with a task, τ_i, is also schedulable if τ_i is replaced by a sporadic server with the same period and execution time.

Proof: To prove this theorem we will show that, for all types of sporadic server execution, the sporadic server exhibits an execution behavior that can also be represented by a combination of one or more periodic tasks.

Let the sporadic server's period be T_{SS} and execution time be C_{SS}. At time t_A, let the priority level of the server become active and let the sporadic server be at full capacity. Consider the interval of time beginning at t_A during which the priority level of the server remains active. The execution behavior of the server during this interval can be described by one of three cases:

1. None of the server's execution time is consumed.
2. All of the server's execution time is consumed.
3. The server's execution time is partially consumed.

Case 1: If the server is not requested during this interval, then it preserves its execution time. The server's behavior is identical to a periodic task for which the interarrival time between two successive requests for the periodic task is greater than its period. Referring to Theorem 2, if any request for a periodic task is delayed it cannot lengthen the critical zone of any lower priority periodic task because the lower priority tasks would be able to execute during the delay, yielding a *shorter* critical zone. Since a critical zone of a lower priority task cannot be made longer, the delay has no detrimental schedulability effect.

Case 2: If the execution time of the sporadic server is completely consumed, a replenishment will be scheduled to occur at $t_A + T_{SS}$ to bring the server back to full capacity. By Lemma 4, the behavior of the sporadic server between t_A and $t_A + T_{SS}$ is identical to that of a similar periodic task that is requested at t_A.

Case 3: If the server's execution time is not completely exhausted, then a replenishment will be scheduled to occur at $t_A + T_{SS}$ for the consumed execution time and the server will preserve the unconsumed execution time until it is requested. Let the amount of execution time to be replenished be C_R.

Now consider a periodic task with a period of T_{SS} and an execution time of C_{SS} that is split into two periodic tasks, τ_x and τ_y, both with a period of T_{SS}. Let τ_x have an execution time of C_R and let τ_y have an execution time of C_{SS} - C_R. By Lemma 3, the splitting of the original periodic task can be done without affecting schedulability. Let requests for both τ_x and τ_y be in sync until t_A. At t_A, let τ_x execute normally, but delay τ_y. As in Case 1, the delay for τ_y has no schedulability effect. The behavior of the two periodic tasks τ_x and τ_y from t_A to $t_A + T_{SS}$ is identical to that of the sporadic server over the same time interval.

Since a sporadic server's execution in each of these cases can be represented by a periodic task or a combination of periodic tasks with a period and total execution time that is identical to that of the sporadic server, the Theorem follows.

If a sporadic server's execution time is only partially consumed before a replenishment is scheduled, as described in Case 3 of the proof for Theorem 5, the server must then manage two quantities of execution time: the execution time that will be replenished and the unconsumed execution time. This could be a concern for sporadic server implementations if many successive executions of the sporadic server split the server's execution time into many small quantities. However, if the server is not requested for a while, these small quantities can be merged together. In fact, regardless of how many times a server's execution time has been split into smaller quantities, if the server is not requested for an amount of time equal to or greater than its period, then all of these quantities can be merged together. An example of this behavior is shown in Figure 2-10. The sporadic server is the highest priority task in the system and has a period of 5 and an execution time of 2. The first four requests for the server require 0.5 units of execution time. Since the priority level of the sporadic server becomes idle between each of these requests, the execution time of the sporadic server has been split four ways, requiring four separate replenishment times. However, no requests are made to the server from time = 4 to time = 9, one server period after the last request. By time = 9, the server's capacity has been completely replenished and merged together.

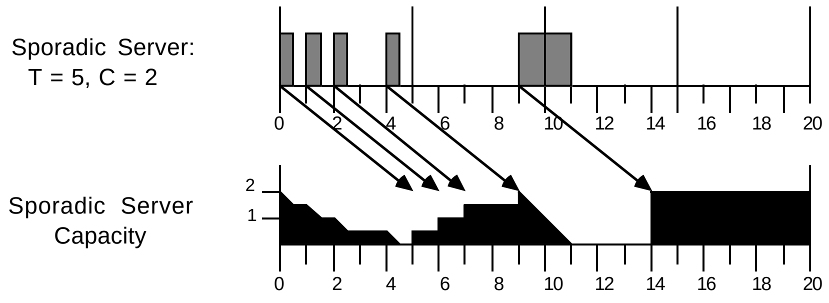


Figure 2-10: An Example of The Splitting and Merging of Sporadic Server Execution Time

In this section we have shown that, although a sporadic server can defer its execution time, it can still be treated as a periodic task. However, it was also shown that this is not true for a deferrable server. The key difference between these two types of servers is that when a sporadic server defers its execution, it also defers the replenishment of its execution time once it is consumed. In contrast, a deferrable server replenishes its execution time independently of when its execution time is consumed.

2.4 Sporadic Server Operation for Sporadic Tasks

One important type of aperiodic task that is not generally supported by previous aperiodic service algorithms is a sporadic task, which is an aperiodic task with a hard deadline with a minimum interarrival time. To guarantee a hard deadline for a sporadic task, a high priority server can be created to exclusively service the sporadic task. This server preserves its execution time at its original priority level until it is needed by the sporadic task. To guarantee that the server will always have sufficient execution time to meet the sporadic task's deadline, a minimum interarrival time restriction must be placed upon the sporadic task. This minimum interarrival time must be equal to or greater than the period of the server task. As long as the sporadic task does not request service more frequently than the minimum interarrival time, its hard deadline can be guaranteed. The SS algorithm can be used to provide a guarantee for a sporadic task as long as the aperiodic task's deadline is equal to or greater than its minimum interarrival time. However, for the cases when the deadline is shorter than the minimum interarrival time, a different priority assignment for the sporadic server and a different schedulability analysis are necessary. The priority of the sporadic server should be based upon the deadline of its associated sporadic task, not upon the maximum arrival rate of the sporadic task. This type of priority assignment, first considered by Leung and Whitehead in [Leung 82], is referred to as deadline monotonic and requires a schedulability analysis different from that necessary for a rate monotonic priority assignment. Apart from the assignment of server priority, the operation of a deadline monotonic sporadic server is identical to that of the SS algorithm as defined in Section 2.1. This section demonstrates the necessity of a deadline monotonic priority assignment for a short deadline sporadic task and describes the schedulability analysis for a task set where the priorities of the periodic tasks are assigned by the rate monotonic algorithm and the priorities of the sporadic servers are assigned with the deadline monotonic algorithm.

2.4.1 A Simple Example of the Deadline Monotonic Priority Assignment

The necessity for a deadline monotonic priority assignment for a short deadline sporadic task will be illustrated with a simple example that shows how a rate monotonic priority assignment cannot guarantee the sporadic task's hard deadline while a deadline monotonic priority assignment can be used guarantee that the hard deadline will always be met. The set of tasks presented in Figure 2-11 will be used for this example.

Periodic Task	Exec Time	Period	Utilization
□	4	12	33.3%
▨	4	20	20.0%

Aperiodic Task	Exec Time	Minimum Interarrival Time	Utilization	Deadline
■	8	32	25.0%	10

Figure 2-11: Periodic Task Set with Short Deadline Aperiodic Task

Figure 2-12 presents the execution of these tasks using the rate monotonic priority assignment for the sporadic task. In Figure 2-12, a high priority is represented by a low number. Note that the rate monotonic algorithm assigns the lowest priority to the sporadic task because of its long minimum interarrival time. The task execution graph presented in Figure 2-12 assumes that requests for both periodic tasks and the sporadic task occur at time = 0. Both periodic tasks meet their deadlines, but the sporadic task completes only 2 of the required 8 units of execution time before it misses its deadline. Clearly, the rate monotonic algorithm is inappropriate for this task set.

Rate Monotonic Priority Assignment:

	Rate	Exec Time	Deadline	Priority
□	12	4	12	1
▨	20	4	20	2
■	32	8	10	3

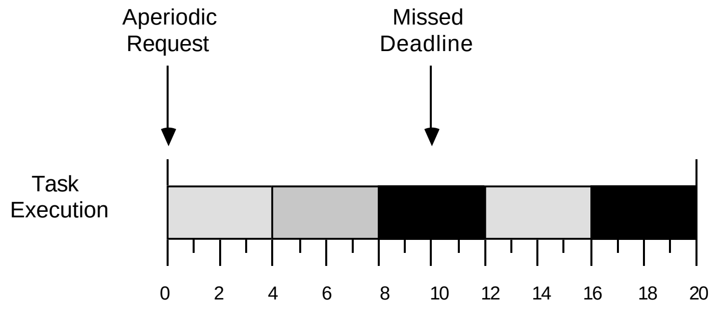
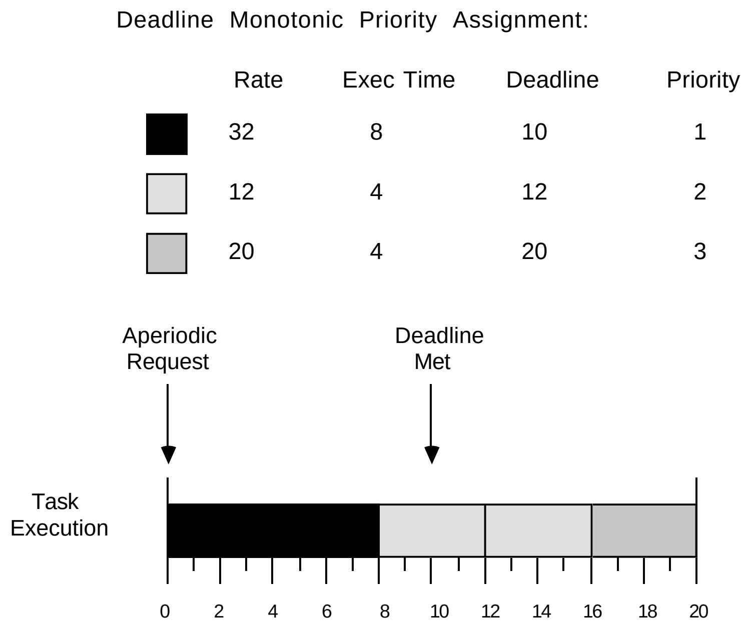


Figure 2-12: Example of Rate Monotonic Priority Assignment for a Short Deadline Sporadic Task

Since a higher priority is needed for the short deadline sporadic task, one might consider treating it as a periodic task with a period of 10. This would give the sporadic task the highest priority and guarantee that its deadline would be met. However, this would cause other problems. The sporadic task only needs a resource utilization of 25% to meet its deadline. Treating it as a periodic task with a period of 10 means that a resource utilization of 80% (an execution time of 8 units divided by a period of 10 units) would be dedicated to a task that needs only 25%. This is a very wasteful scheduling policy. Also, if the priority of the sporadic task is increased in this manner, the total utilization required to schedule all three tasks is now 133.3% (33% + 20% + 80%) making this an infeasible schedule. A different method is needed that will assign priorities such that all their deadlines will be met using an efficient level of scheduled resource utilization.

A deadline monotonic priority assignment can be used to efficiently schedule the task set presented in Figure 2-11. A sporadic server is created with an execution time of 8 units and a period of 32 units to service the short deadline sporadic task. The priority of the sporadic task is assigned based upon the deadline of the task it is servicing, not upon its period. In other words, the rate monotonic algorithm is used to assign priorities for the periodic tasks and the priority of the sporadic task is assigned as if its rate of occurrence were equal to its deadline. The task execution graph for this priority assignment is presented in Figure 2-13. The sporadic task now has the highest priority and meets its deadline. The execution of the periodic tasks is delayed but both still meet their deadlines.



	Rate	Exec Time	Deadline	Priority
■	32	8	10	1
□	12	4	12	2
▨	20	4	20	3

Figure 2-13: Example of Deadline Monotonic Priority Assignment for a Short Deadline Sporadic Task

Now that we have shown that a deadline monotonic priority assignment can be used to guarantee that a sporadic task will meet its deadline, we need to be able to perform a schedulability analysis that will indicate whether or not a deadline monotonic priority assignment for the sporadic server and a rate monotonic priority assignment for the periodic task is feasible for a given task set. The deadline monotonic priority assignment raises the priority of the sporadic server above the priority that would be assigned by the rate monotonic algorithm. As such, the sporadic server can be considered as a low priority task that is allowed to block a higher priority task from executing. A similar problem can occur when periodic tasks that share data using critical sections are scheduled using the rate monotonic algorithm. A low priority task that has already entered a critical section can block the execution of a high priority task that needs to enter the critical section. The blocking of the high priority task continues as long as the critical section is locked by the lower priority task. Sha, Rajkumar, and Lehoczky [Sha 87] have developed the priority ceiling protocol for a set of periodic tasks that share data using critical sections. This protocol establishes a set of sufficient conditions under which the periodic tasks can be scheduled using the rate monotonic algorithm. The schedulability analysis equations developed for the priority ceiling protocols can be used to test the schedulability of sporadic servers with deadline monotonic priority assignments. These equations are presented below:

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1) \tag{3}$$

$$\forall\, i,\ 1 \le i \le n, \qquad \min_{(k,\,l) \in R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1 \tag{4}$$

where C_i and T_i are respectively the execution time and period of task τ_i, $U_\mathrm{i} = C_\mathrm{i}/T_\mathrm{i}$ is the utilization of task τ_i, and $R_\mathrm{i} = \{\, (k, l) \mid 1 \le k \le i,\ l = 1, \cdots, \lfloor \mathrm{T_i}/\mathrm{T_k} \rfloor \,\}$. The term B_i is the worst-case blocking time for task τ_i.

Equations 3 and 4 were derived from equations developed for testing the schedulability of a set of periodic tasks whose priorities have been assigned using the rate monotonic algorithm. Equation 3 was derived using the worst case utilization bound equation for scheduling periodic tasks developed by Liu and Layland in [Liu 73] which, under the absolute worst case conditions, provides a sufficient condition for determining schedulability of a rate monotonic priority assignment. Equation 4 was derived from an equation developed by Lehoczky, Sha, and Ding [Lehoczky 89] that provides necessary and sufficient conditions for determining schedulability of a rate monotonic priority assignment. Both Equations 3 and 4 provide sufficient conditions under which a periodic task set that allows lower priority tasks to block higher priority tasks can be scheduled by the rate monotonic algorithm. However, Equation 3 represents the absolute worst case conditions and a much tighter characterization is provided by Equation 4.

The blocking term, B_i, in Equations 3 and 4 represents the amount of time that a lower priority task can block a higher priority task from executing. For a sporadic server with a deadline monotonic priority, B_i is used to represent the amount of time the sporadic server can block a periodic task that has a higher priority than the rate monotonic priority of the sporadic server. To check the schedulability of a periodic task set with a deadline monotonic sporadic server, the term B_i is set equal to the execution time of the sporadic server for all τ_i with a priority less than or equal to the priority of the sporadic server and greater than the sporadic server's original rate monotonic priority. For all τ_i with a priority less than the sporadic server's original rate monotonic priority, the sporadic server should be treated normally (i.e., treated as a normal sporadic server with a rate monotonic priority assignment). For all τ_i with a priority greater than the priority of the sporadic server, the corresponding value of B_i is set to zero.

The use of Equations 3 and 4 will be demonstrated with the task set presented in Figure 2-11. Let τ_1 and τ_2 be the periodic tasks and let τ_3 be the short deadline sporadic task. Below are the parameters for each task:

Task	C_i	T_i	B_i
τ_1	4	12	8
τ_2	4	20	8
τ_3	8	32	0

Evaluation of Equation 3 proceeds as follows:

$$i = 1, \quad \frac{C_1}{T_1} + \frac{B_1}{T_1} \leq 1$$

$$\frac{4}{12} + \frac{8}{12} \leq 1$$

$$\frac{12}{12} \leq 1$$

$$i = 2, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq 2(2^{1/2}-1).$$

$$\frac{4}{12} + \frac{4}{20} + \frac{8}{20} \leq 0.824$$

$$\frac{56}{60} \geq 0.824$$

The inequality for i = 1 holds but the inequality for i = 2 does not, therefore it is not necessary to check the i = 3 case and Equation 4 must now be used. The evaluation of Equation 4 proceeds as follows.

i = 1: Check if $C_1 + B_1 \leq T_1$. Since $4 + 8 \leq 12$, task τ_1 is schedulable.

i = 2: Check whether either of the following two inequalities hold:

<u>(k, l)</u>		
(1, 1)	$C_1 + C_2 + B_2 \leq T_1$	$4 + 4 + 8 \geq 12$
(2, 1)	$2C_1 + C_2 + B_2 \leq T_1$	$8 + 4 + 8 \leq 20$

The inequality for (k, l) = (2, 1) holds and therefore, τ_2 is schedulable.

i = 3: Check whether any of the following inequalities hold:

<u>(k, l)</u>		
(1, 1)	$C_1 + 2C_2 + C_3 + B_3 \leq T_1$	$4 + 8 + 8 + 0 \geq 12$
(1, 2)	$2C_1 + 2C_2 + C_3 + B_3 \leq 2T_1$	$8 + 8 + 8 + 0 \leq 24$
(2, 1)	$2C_1 + C_2 + C_3 + B_3 \leq 8T_2$	$8 + 4 + 8 + 0 \leq 20$
(3, 1)	$3C_1 + 2C_2 + C_3 + B_3 \leq 8T_2$	$12 + 8 + 8 + 0 \leq 32$

Since all the inequalities except (k, l) = (1, 1) hold, τ_3 is schedulable. Note that it would have been sufficient to stop checking after finding one inequality that holds for each value of i; all the inequalities are listed here for completeness only.

2.5 Sporadic Servers and Priority Inheritance Protocols

In this section, we discuss the schedulability impact of servicing aperiodic tasks that share data using priority inheritance protocols developed by Sha, Rajkumar, and Lehoczky in [Sha 87]. The interaction of sporadic servers and the priority inheritance protocols is also defined.

2.5.1 The Priority Inheritance Protocols

Mok [Mok 83] has shown that the problem of determining the schedulability of a set of periodic tasks 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 [Sha 87] 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.* 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.
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.

2.5.2 The Schedulability Impact of Sporadic Servers and the Priority Inheritance Protocols

We now investigate 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 example describes 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 [Sha 87]. 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 [Sha 87]:

*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 [Sha 87].

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1) \tag{5}$$

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 [Liu 73] which, under the absolute worst case conditions, provides a sufficient condition for determining schedulability of a rate monotonic priority assignment.

For these 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 "preemption time". Blocking occurs when a lower priority task blocks the execution of a higher priority task. Preemption 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 preemption 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 preemption 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 5 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 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 5 to determine the schedulability of the second example proceeds as follows:

$$\frac{C_{SS}}{T_{SS}} \leq 1(2^{1}-1)$$

$$\frac{3}{10} \leq 1$$

$$\frac{C_{SS}}{T_{SS}} + \frac{C_P}{T_P} + \frac{B_P}{T_P} \leq 2(2^{\frac{1}{2}}-1).$$

$$\frac{3}{10} + \frac{8}{20} + \frac{2}{20} \leq 0.83$$

$$\frac{8}{10} \leq 0.83$$

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 preemption 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 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 preempt *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 preemption 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 preemption 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 later in Section 5.3.5.

2.5.3 Interaction of Sporadic Servers and the Priority Inheritance Protocols

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 now consume the execution time of the aperiodic task's sporadic server. This expands the potential users of the sporadic server execution time 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.

3. Sporadic Server Performance for Soft-Deadline Aperiodic Tasks

3.1 Performance Issues

In this chapter, we investigate the response time performance of the SS algorithm for soft-deadline aperiodic tasks executing with hard-deadline periodic tasks. The goal is to provide better response time performance for the soft-deadline aperiodic tasks by reducing the interference of the hard-deadline periodic tasks. The minimum requirement for SS performance is to provide better response times than a polling server. Also, since the sporadic server algorithm was designed as a replacement for the DS and PE algorithms, the sporadic server should perform as well, if not better, than these previous algorithms. Finally, the ideal performance goal is to provide a level of service to the aperiodic tasks that would be equivalent to the complete elimination of the periodic tasks. The results presented in this chapter show that the SS algorithm:

- performs better than polling
- provides comparable, and in some cases better, performance than the DS and PE algorithms
- for many cases, provides a level of service equivalent to the service that could be provided if no periodic tasks were present.

Many factors affect aperiodic response time performance. In this chapter we examine the following factors:

- **Periodic Tasks.** The total periodic task utilization determines the resource utilization available for the aperiodic tasks. The secondary effect of the periodic task set is that the collection of periods and execution times of the periodic tasks directly determines the possible maximum aperiodic server sizes (size = execution time / period).
- **Aperiodic Server.** The aperiodic server algorithm and the server's execution time, period, and priority directly effect aperiodic response time performance. The execution time of the server determines how much high-priority, continuous service can be provided to an aperiodic task. Once the aperiodic server is exhausted, the aperiodic server period determines how long pending aperiodic tasks must wait to receive service. The priority of the aperiodic server determines how quickly available server execution time can be provided to an aperiodic task.
- **Aperiodic Tasks.** The aperiodic load has a direct affect upon aperiodic response time performance. The smaller the aperiodic load is compared to the aperiodic server size, then the more likely it is that the aperiodic server will always be able to provide immediate, uninterrupted service to the aperiodic tasks. However, the mean service time also has a significant affect upon aperiodic response time performance. The smaller mean service time is compared to the aperiodic server's execution time, the more often the aperiodic server can service aperiodic requests without exhausting its execution time capacity and suspending high-priority aperiodic service.

This chapter is organized as follows. Section 3.2 presents the results of a simulation study comparing the aperiodic response time performance of the background, polling, DS, PE, and SS algorithms for wide ranges of periodic and aperiodic loads. Section 3.3 investigates the context swap overhead of each aperiodic service algorithm. Section 3.4 presents an alternate service order policy for the SS algorithm which improves the average aperiodic response time and lowers the context swap overhead. Section 3.5 compares the response time performance effects of the full SS replenishment policy to a simpler replenishment policy which has a more efficient implementation. Section 3.6 investigates the effect of server priority on the performance of the SS algorithm. Section 3.7 investigates the maximum sporadic server execution time as a function of server period and priority. Section 3.8 investigates the average aperiodic response time as a function of the sporadic server period. Finally, Section 3.9 closes this chapter with a discussion of how to select priority, period, and execution time for the SS algorithm to meet the demands of an application.

3.2 Average Response Time Comparison: Background, Polling, DS, PE, SS

In this section, we present the results of simulation study to compare the aperiodic response time performance of the Background, Polling, DS, PE, and SS aperiodic service algorithms.

3.2.1 Experiment Description

For these experiments, ten different periodic task sets, each containing ten periodic tasks, were chosen at random. A period of 55 units was selected to be the minimum period for each periodic task set. The maximum periods for the task sets ranged from 210 to 2,310 units. The relative phasing of task periods within each task set was chosen at random. For each periodic task set, three periodic loads were chosen for simulation: 40%, 60%, and 80%.

The aperiodic tasks for these experiments were chosen as follows. The aperiodic task arrival times were modeled using a Poisson arrival process and the aperiodic task service times were modeled using an exponential service time distribution. Four different mean aperiodic service times were chosen to be 1%, 2%, 5%, and 10% of the minimum periodic task period (i.e. 0.55, 1.10, 2.75, and 5.50 units respectively). These aperiodic service times span the range of small aperiodic requests to more substantial aperiodic jobs. For each combination of periodic task load and mean aperiodic service time, five aperiodic loads were chosen. These aperiodic loads span the range of resource utilization unused by the periodic tasks from a total load just above the periodic task load up to a total load of 90%.

The background, polling, DS, PE, and SS algorithms were simulated[*] for each of the combinations described above. The period of each aperiodic server was chosen to be the longest period such that the rate monotonic priority of the server would be the highest priority in the system. Thus, the aperiodic server period was set equal to the minimum periodic task period (55). The execution time of each aperiodic server was chosen to be the maximum value at which the periodic task set remains schedulable. This strategy gives the largest server execution time at the highest priority level for a rate monotonic priority assignment. The average, maximum, and minimum server sizes for the polling, DS, PE, and SS algorithms for each periodic load are presented in Figure 3-1. The range of the server sizes shows the strong dependency of the maximum server execution time upon the periodic task set.

The simulation length for each of these experiments was chosen to be long enough to get a large number of aperiodic arrivals and long enough to go through the entire range of periodic task interactions many

[*]A description of the simulator used for these experiments is presented Appendix II

	Deferrable Server						Polling, Priority Exchange, or Sporadic Server					
Periodic Load	40%		60%		80%		40%		60%		80%	
	Exec Time	Size	Exec Time	Size	Exec Time	Size	Exec Time	Size	Exec Time	Size	Exec Time	Size
Average	21.8	40%	15.2	28%	5.7	10%	30.3	55%	18.3	33%	6.2	11%
Maximum	25.3	46%	19.6	35%	10.7	19%	32.9	59%	21.9	39%	10.9	19%
Minimum	18.4	33%	11.2	20%	1.4	2%	27.4	49%	15.1	27%	1.8	3%

Figure 3-1: Average, Maximum, and Minimum Execution Times and Sizes for the Aperiodic Servers

times. The length of each simulation was selected to insure first that approximately 10,000 aperiodic arrivals would occur before terminating the simulation. However, in order to insure that the entire range of periodic task interactions would be examined, the length of the simulation was increased, if necessary, to be at least 20 hyperperiods long. The hyperperiod of a task set is the least common multiple of all the task periods. Every hyperperiod, the sequence of periodic arrivals repeats itself.

To summarize, the parameters of the experiments were as follows:

- 10 periodic task sets
- 3 periodic loads
- 4 mean aperiodic service times
- 5 aperiodic loads
- 5 aperiodic service algorithms

Appendix I contains a summary of the experiment parameters which includes all the periodic task set and server size information.

3.2.2 Experiment Results for Average Response Time

The average response time performance data for a mean aperiodic service time of 0.55 units are presented in Figure 3-2. Figure 3-2 presents the average response time performance for the background, polling, DS, PE, and SS algorithms. Three graphs are shown, one for each periodic load (Experiment 1: 40%, Experiment 2: 60%, and Experiment 3: 80%). For each of the graphs in Figure 3-2, the y-axis represents average response time; the bottom scale represents the total periodic and aperiodic load; and the top scale shows the mean aperiodic arrival time. The data points presented in Figure 3-2 were obtained by averaging the data from each periodic task set simulation. In other words, each data point plotted in Figure 3-2 is an average of the ten average response times for the ten different periodic task sets. An additional curve, the M/M/1 curve, is also plotted on these graphs. The M/M/1 curve represents the ideal response time for the aperiodic tasks that would be obtained if the periodic tasks were not present. Since a low response time is desired, the lower the curve for a particular algorithm is on the graph, the better its response time performance.

Referring to the graph for Experiment 1 in Figure 3-2, one can see that the SS algorithm performs very well. The SS algorithm provides a substantial improvement in response time performance over the background, polling, and priority exchange algorithms for the entire range of aperiodic loads tested. The SS algorithm performance is as good as the DS algorithm performance for aperiodic loads up to 30% and the SS algorithm provides better response time performance than the DS algorithm for aperiodic loads greater than 30%. Also, note that the SS algorithm effectively eliminates the interference of the periodic tasks for aperiodic loads loads up to 40% (i.e. the SS curves lie on top of the M/M/1 curve).

The aperiodic response time performance of background and polling service displayed in the graph of Experiment 1 in Figure 3-2 can be explained as follows. As the aperiodic load increases, the response time performance for all algorithms should increase (this is true for the DS and SS algorithms but the range of the graph is too large to see the increase). Recalling the discussion in Section 1.2.3, the response time for background service is poor because the intervals between aperiodic service opportunities are typically large. The response time performance for polling is an improvement over background service, but aperiodic tasks typically have to wait for the start of the poller’s period before receiving service.

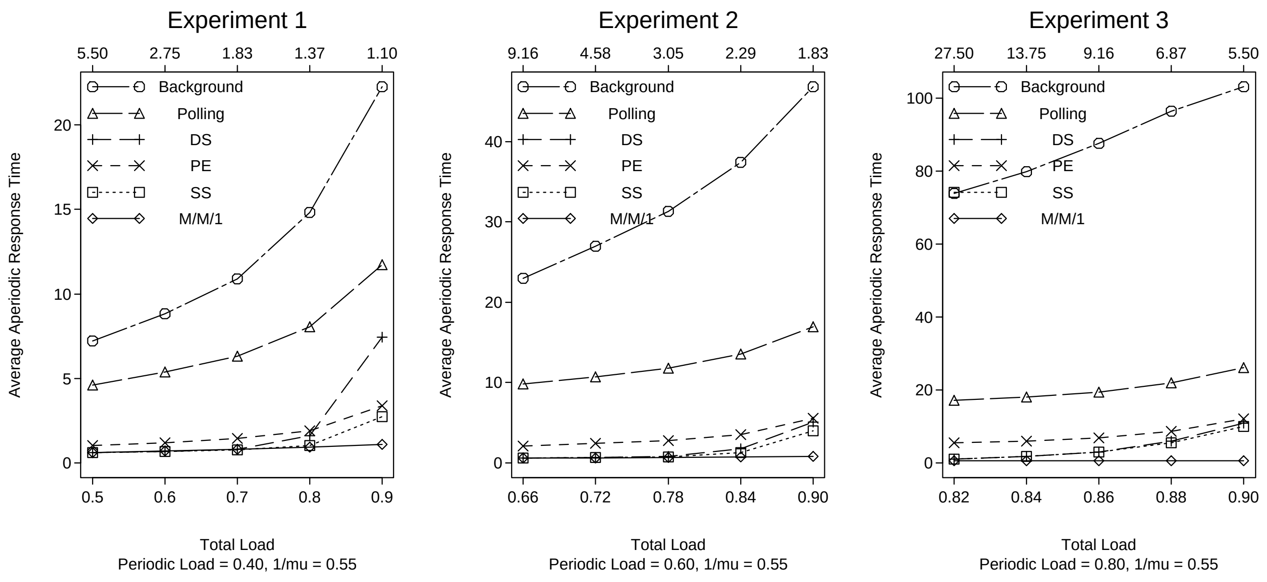


Figure 3-2: Average Aperiodic Response Time for Experiments: 1, 2, and 3

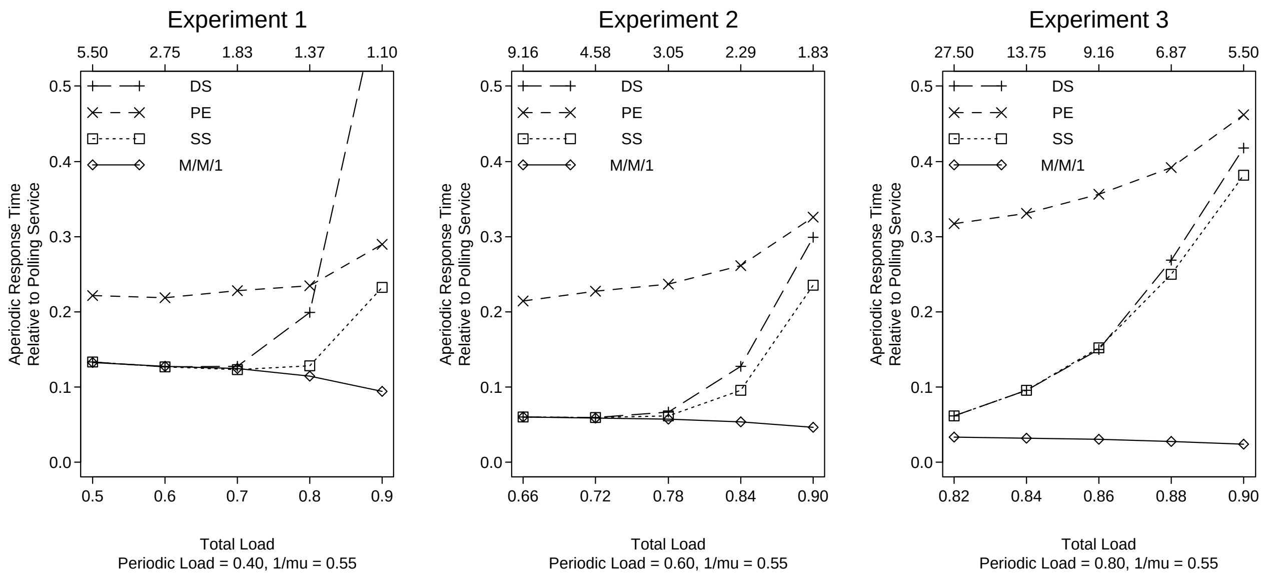


Figure 3-3: Average Aperiodic Response Time Relative to Polling for Experiments: 1, 2, and 3

The substantial improvement in aperiodic response time performance of the DS, PE, and SS algorithms over background and polling service for Experiment 1 in Figure 3-2 is explained as follows. The response time improvement is due to the fact that each of these algorithms preserves its execution time until needed by the aperiodic tasks. This allows the algorithms to provide immediate service to most aperiodic requests. For a range of aperiodic load up to 30% (i.e., a total load of 80%), the PE algorithm performs slightly worse than the DS and SS algorithms because the PE algorithm does not preserve its execution time at the highest priority level. Since the PE algorithm is often forced to trade its high priority execution time with lower priority, periodic tasks, its execution time is typically at a lower priority than the DS and SS algorithms. The DS and SS algorithms perform as well as the ideal M/M/1 response time over this same range of aperiodic load because their execution time is always at the highest priority. Starting at an aperiodic load of about 30%, the response time performance of the DS algorithm becomes worse than the response time performance of the PE and SS algorithms. This increase in DS response time is due to the substantially smaller average DS size (see Figure 3-1) for a 40% periodic load. Since both the PE and SS algorithms have a larger server size than the DS algorithm, these algorithms are able to maintain near ideal response time performance for a higher aperiodic load than the DS algorithm. The departure from the M/M/1 curve for the DS and SS algorithms occurs because, as the aperiodic load increases relative to their respective server sizes, the more likely it is that their server's will be come exhausted, forcing the aperiodic tasks to wait until their execution time is replenished.

The aperiodic response time behavior for these algorithms needs to be examined as the periodic load increases. The data for a mean aperiodic service time of 0.55 units and periodic loads of 60% and 80% are shown in the graphs of Experiments 2 and 3 in Figure 3-2. Note that the response time improvement of the DS, PE, and SS algorithms relative to the performance of background and polling service increases as the periodic load is increased from Experiment 1 through Experiment 3 (the range of the average aperiodic response time of the graphs in Figure 3-2 increases as the periodic load increases). This behavior is expected for background because, as the periodic load increases, background service opportunities typically become further apart and shorter. The response time performance of polling becomes worse as the periodic load increases because, although the frequency of polling service opportunities does not change as the periodic load increases, the execution time and size of the polling server decreases. The effect of the smaller server execution time is also seen in the response time curves for the DS and SS algorithms as they move away from the M/M/1 curve at lower aperiodic loads as the periodic load increases. Also, note that the response time performance difference between the DS and SS algorithms decreases as the periodic load increases. The reason the DS and SS performance become more similar as the periodic load increases is that the difference in their execution times and sizes becomes less (see Figure 3-1).

To better illustrate the performance benefits of the DS, PE, and SS algorithms, the data for these algorithms from Figure 3-2 is presented relative to the performance of polling service in Figure 3-3. In the graphs of Figure 3-3, the response time performance of polling service would have a value of 1.0 at all aperiodic loads on each of the graphs. Noting the relative response time performance for low aperiodic load in Experiments 2 and 3, we see that DS, PE, and SS algorithms can provide an order of magnitude response time improvement over polling service (i.e. the response time curves indicate the average response time of the DS, PE, and SS algorithms is less than one tenth the average response time performance of polling service). Also note, as mentioned above, the performance for low aperiodic loads of the DS, PE, and SS algorithms relative to polling improves as the periodic load increases.

The effect of increasing the mean aperiodic service time upon response time performance can be seen with the graphs for all the experiments presented in Figure 3-4. The top three graphs in Figure 3-4 are the same as the graphs in Figure 3-2. Each different row of graphs in Figure 3-4 represents a different mean aperiodic service time, ranging from a mean aperiodic response time of 0.55 units on the top row to a mean aperiodic service time of 5.50 units on the bottom row. Figure 3-3 presents the response time performance of the DS, PE, and SS algorithms relative to the performance of polling service for all the experiments. The top three graphs in Figure 3-5 plot the same data presented in Figure 3-3.

Referring to Figure 3-4, as the mean aperiodic service time increases, the aperiodic load at which the response time performance of the DS and SS algorithms moves away from the M/M/1 response time curves becomes lower. A departure from the M/M/1 curve indicates that some aperiodic jobs have a longer response time than they would experience if no periodic tasks were present. The only time this can occur with either the DS or the SS algorithm is when the aperiodic server's execution time is exhausted, causing all pending aperiodic tasks to wait longer for service than they would in an M/M/1 queue. Two factors change as the mean aperiodic service time is increased while the aperiodic load is maintained constant. The first factor is the probability that the aperiodic server is overrun (i.e. the probability that the server's execution time becomes exhausted and high-priority aperiodic service is suspended). The probability of the server being overrun increases as the mean service time is increased (this will be demonstrated later in Section 4.2). The second factor is the percentage of aperiodic jobs whose response times are lengthened by server overruns. As the mean service time increases, fewer aperiodic jobs are necessary to exhaust the aperiodic server. Thus, as the mean service time increases, the response time for a greater percentage of aperiodic jobs is lengthened because of server exhaustion. Since the effect of both of these factors increases with mean service time, the departure of the average aperiodic response time from the M/M/1 response time will occur at lower aperiodic loads as the mean service time increases.

It was also noted earlier that as the periodic load increases while the mean aperiodic service time is unchanged, the departure of the average aperiodic response time curve for the DS and SS algorithms from the M/M/1 response curve occurs at lower aperiodic loads. The important factors that cause this behavior are the decrease in server execution time and the increase in the percentage of aperiodic jobs whose response times are lengthened by server overruns. Since the server has a smaller execution time as the periodic load increases but the server's period has not changed, the aperiodic job must wait longer for the server to be replenished once the server has been exhausted. Also note that, the decrease in server execution time as the periodic load increases lowers the number of aperiodic jobs required to exhaust the server's capacity. The effect of both of these factors upon average aperiodic response time increases as the periodic load increases, causing the average response time of the DS and SS algorithms to depart from the M/M/1 response time at lower aperiodic loads.

The key factor which determines when the average aperiodic response time for the DS and SS algorithms departs from the ideal M/M/1 response time is the the ratio of the mean service time to the server's execution time. This ratio determines the average number of jobs required to exhaust the server. The higher this ratio, the lower the aperiodic load will be at which the average aperiodic response time moves away from the ideal M/M/1 response time.

Extreme cases of a server being overrun are found for the DS algorithm in Experiments 7 and 10 in Figure 3-4. For a high aperiodic load in both of these figures, the response time performance of the DS algorithm actually becomes worse than the response time performance of polling. This behavior is due to several factors: the DS has a smaller server execution time than the poller (see Figure 3-1) and both the aperiodic load and mean aperiodic service times are large.

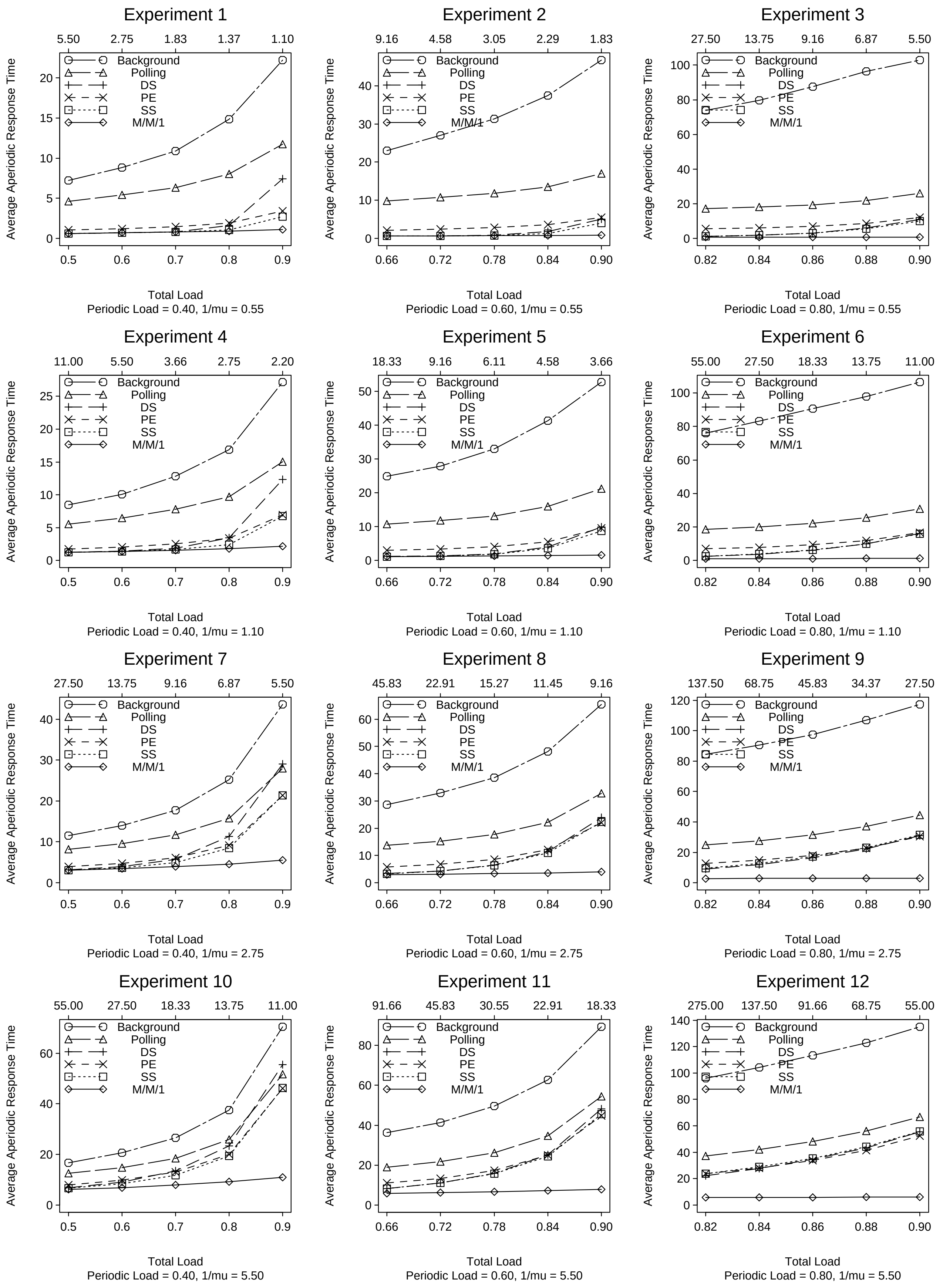


Figure 3-4: Average Aperiodic Response Time for the Background, Polling, DS, PE, and SS Aperiodic Service Algorithms

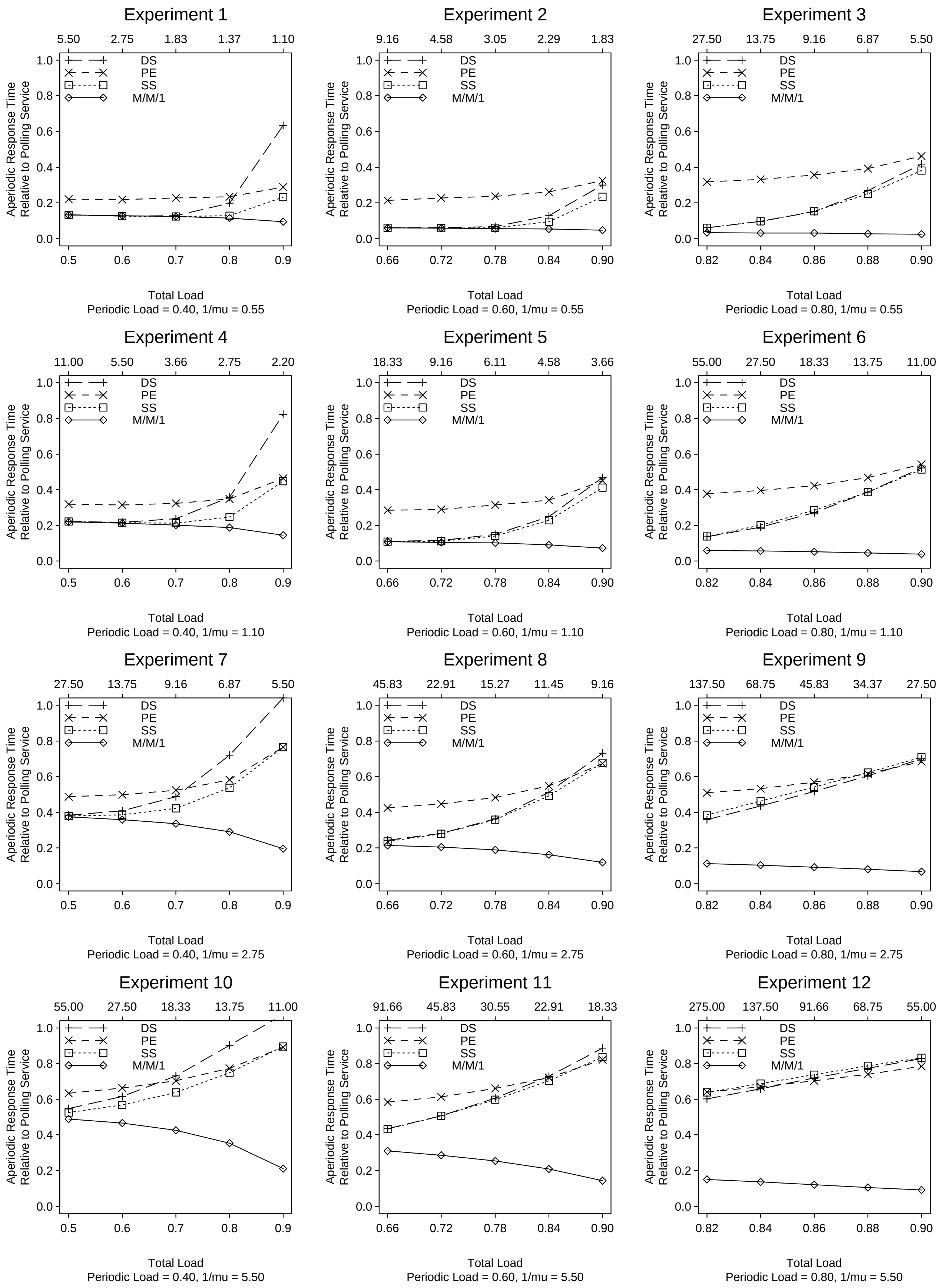


Figure 3-5: Average Aperiodic Response Time Relative to Polling Service for the DS, PE, and SS Aperiodic Service Algorithms

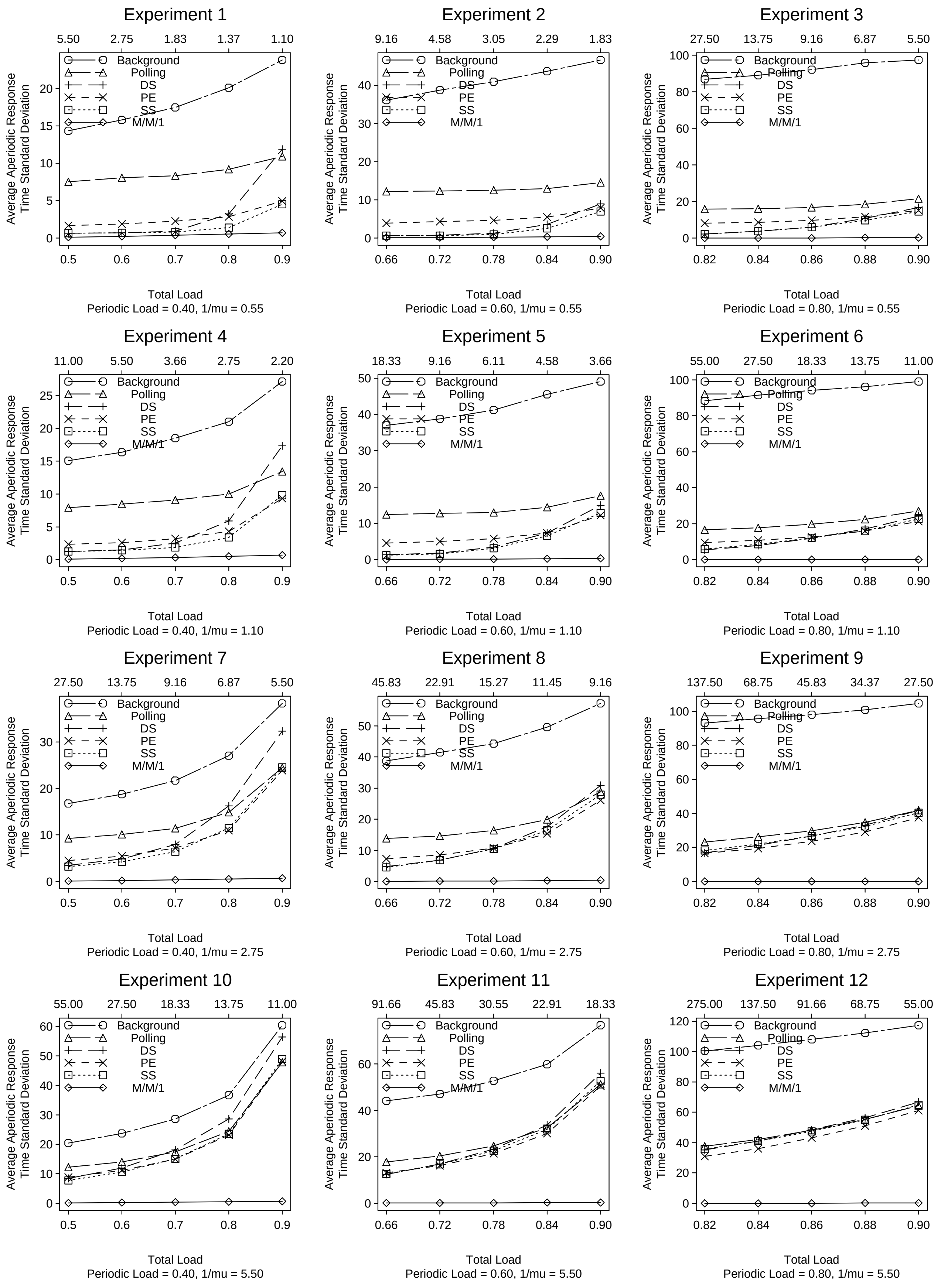


Figure 3-6: Average Response Time Standard Deviation for the Background DS, PE, and SS Aperiodic Service Algorithms

From the graphs in Figure 3-5, we see that the preformance improvement of the DS, PE and SS algorithms relative to polling decreases as the mean aperiodic service time increases. However, even for the cases of high aperiodic load and large mean aperiodic service time, these algorithms still provide approximately a 20% improvement in response time performance over polling.

Another important performance factor for aperiodic response time is the variance of the response time. For many soft-deadline aperiodic tasks, it is important to have a low response time variance. A good example of this would be an operator request. A high response time variance for an operator request can be frustrating to the operator and may result in duplicate requests for the same operation. Figure 3-6 presents the average of the standard deviation for each of the ten periodic tasks sets. As can be seen from the graphs in Figure 3-6, the relative performance of the aperiodic service algorithms with respect to response time variance is very similar to their performance for average response time. The notable difference is that as the mean aperiodic service time increases, the difference between the SS and polling algorithms with respect to response time variance is less than the corresponding difference for average response time, especially for the cases of high periodic and aperiodic load. However, for many cases, the response time variance of the SS algorithm is close to the ideal variance of an M/M/1 system and substantially less than the response time variance of polling.

3.2.3 Confidence Intervals for Aperiodic Response Time Experiments

The simulation experiments described in Section 3.2 were used to compare the response time performance of various aperiodic service algorithms. To provide a fair comparison of the various aperiodic service algorithms, the same random number seeds were used to generate the aperiodic arrival times and aperiodic service times for all simulations. In this section we discuss the confidence intervals for these simulations.

To investigate the confidence intervals for these experiments, we repeated a number of experiments for the SS algorithm using different seeds for the random number generators. For each repeated experiment, ten different sets of seeds were used. First, we repeated Experiment 8 for all task sets. Second, we repeated all experiments using Task Set 5. The data from the original simulations and the ten new simulations were then used to compute the 95% confidence intervals using the following equation from [Allen 78]:

$$\overline{x} \pm t_{5/2} \frac{s}{\sqrt{n}} \tag{6}$$

where $t_{5/2} = 2.228$ is the 97.5 percentile point for the Student t-distribution with 10 degrees of freedom, s is the sample standard deviation, and n = 11 is the number of samples.

Figure 3-7 presents the results for Experiment 8 using all of the task sets and Figure 3-8 presents the results for all of the experiments using Task Set 5. The solid line in these graphs presents the average aperiodic response time and the dashed lines present the upper and lower bounds for the 95% confidence intervals. For all the reported averages for total loads less than 90% in Figures 3-7 and 3-8, the percentage error in the estimate of the mean is less than 5% with 95% confidence. For the reported averages for total loads of 90% in Figures 3-7 and 3-8, the percentage error in the estimate of the mean is less than 7.5% with 95% confidence. In none of these cases does the estimated error in the mean change the conclusions reported in Section 3.2.

3.3 Context Swap Overhead Comparison: Background, Polling, DS, PE, SS

Section 3.2 demonstrated that the SS algorithm has an average response time performance for soft-deadline aperiodic tasks that is substantially better than polling and is comparable to or better than the DS algorithm. However, the response time performance advantage of the DS and SS algorithms is attained by preempting periodic tasks to provide immediate service for aperiodic tasks. In other words, each use of an aperiodic server requires a context swap to save the state of the periodic task and load the state of the aperiodic task. Therefore, the performance benefit of the DS and SS algorithms is not attained without an increase in context swap overhead. This section presents and discusses the context swap overhead data for the experiments described in Section 3.2.

3.3.1 Characterization of Context Swap Overhead

Context swap overhead occurs because the tasks share a resource (e.g. the processor) in a priority-driven, preemptive manner. From a per-task point of view, the highest priority task will have the lowest context swap overhead: 1 context swap to execute the task for each time it is activated.[*] Lower priority tasks incur more overhead because they can be preempted by higher priority tasks. A lower priority task incurs 1 context swap each time it is activated and 1 context swap for each time it is resumed after being preempted. Note that context swap overhead only refers to the action of saving the current processor state and loading the next state upon the scheduling of a different job to execute. Thus, context swap overhead only occurs when another job is chosen to execute. No context swap occurs when a job finishes and no other job is ready to execute.

To characterize the overhead of each of the aperiodic service algorithms, the number of context swaps for each periodic and aperiodic task was recorded for the experiments described in Section 3.2. To present the context swap overhead data, we use the ratio of the number of context swaps for a task divided by the number of times a task was activated. For example, a system with only one task (i.e. a system in which no preemptions occur) would have a context swap overhead ratio of 1.0, representing the lowest possible context swap overhead. A context swap overhead ratio of 3/2 for a medium priority task would imply that, on the average, the task is preempted once for every two activations (i.e. an additional context swap occurs once every other activation). Thus, the larger the difference between the context swap overhead ratio and 1.0, the greater the context swap overhead. To present the context swap overhead data for a complete task set, the context swap overhead ratio is the total number of periodic and aperiodic context swaps divided by the total number of periodic and aperiodic task activations.

[*]An activation refers to one complete execution of a task. For example, a periodic task with a period of 10 units is activated 10 times over an interval of 100 units.

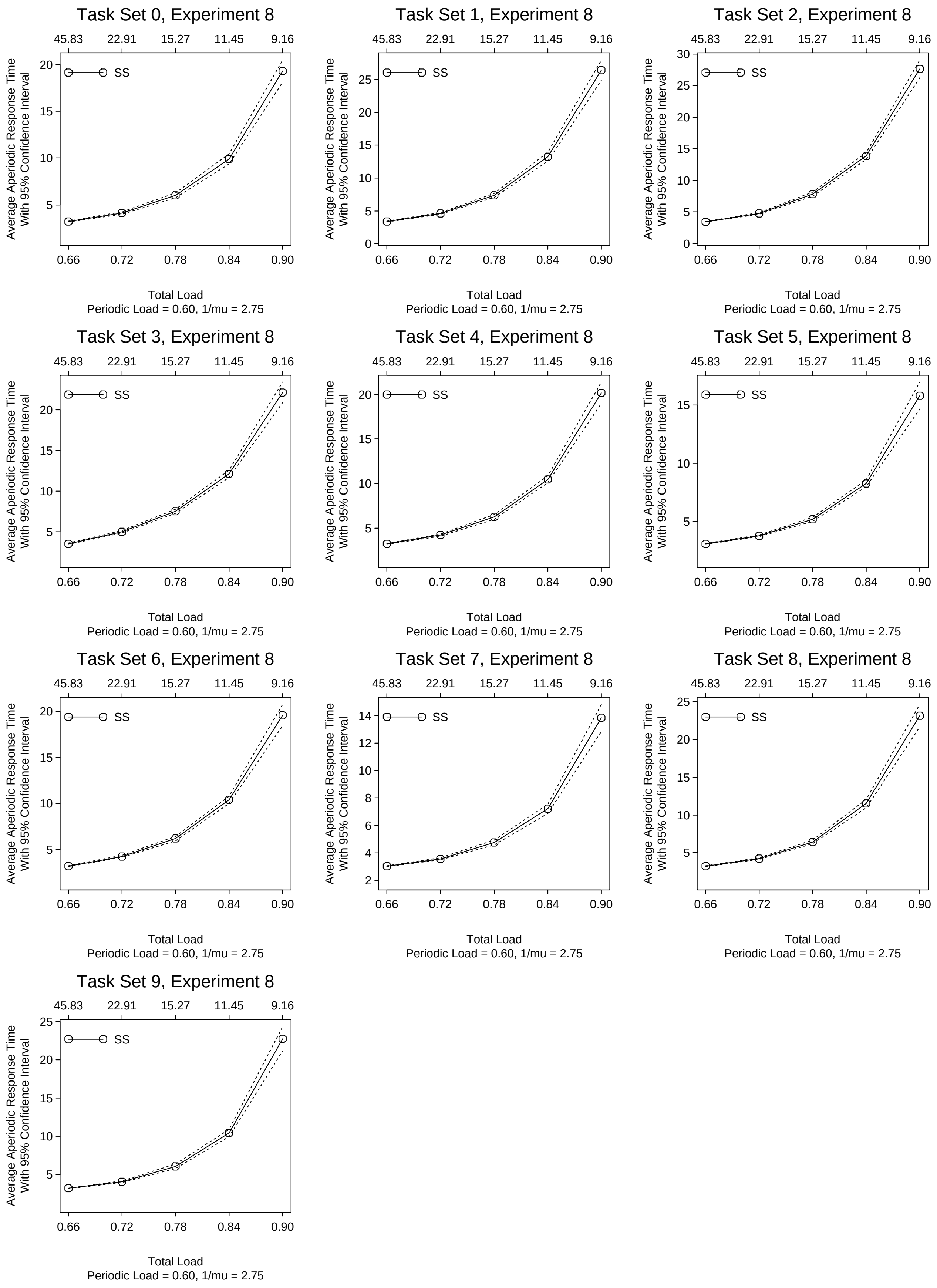


Figure 3-7: Average Aperiodic Response Time With 95% Confidence Interval For Experiment 8 For Each Task Set Using The SS Algorithm

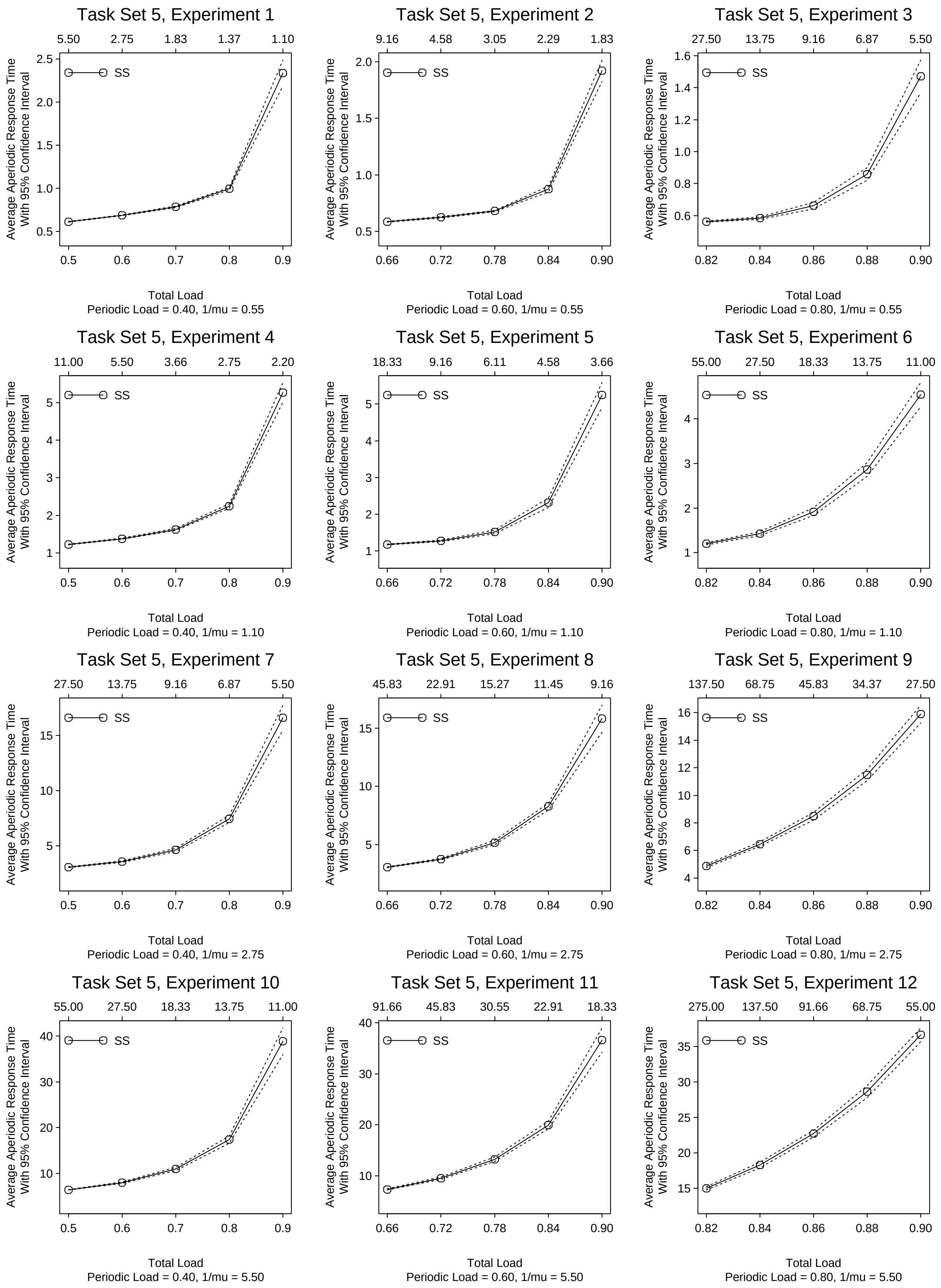


Figure 3-8: Average Aperiodic Response Time With 95% Confidence Interval For All Experiments Using Task Set 5 For The SS Algorithm

3.3.2 Experiment Results for Context Swap Overhead

Figure 3-9 presents the context swap overhead data for the experiments described in Section 3.2. These graphs are similar to the graphs presented in Figure 3-4: the bottom scale represents the total periodic and aperiodic load and the top scale shows the mean aperiodic arrival time. However, in Figure 3-9, the y-axis represents the context swap overhead rather than average aperiodic response time. The data points presented in Figure 3-9 were obtained by averaging the context swap ratio data from each periodic task set simulation. In other words, each data point plotted in these graphs is an average of the ten context swap overhead ratios for the ten different periodic task sets.

As we did with the average response time experiments, we will first discuss the background and polling data for Experiment 1. In this experiment, the overhead for the background and polling aperiodic service algorithms is lower than the overhead for the DS, PE, and SS algorithms. The aperiodic service opportunities provided by the background and polling algorithms are independent of the arrival of aperiodic requests. As such, most aperiodic tasks arrive and wait to be serviced. Therefore, since the mean service time for the aperiodic tasks is small, many short aperiodic jobs are typically waiting to be serviced when a polling or background service opportunity arrives. The duration of background service opportunities is usually large, since the periodic load is only 40% (leaving 60% utilization for background service) and the duration of polling service opportunities is also large (approximately 30 units in size, see Figure 3-1). Therefore, when the aperiodic tasks are serviced by either by the polling or background algorithms for the task set of Experiment 1, many aperiodic jobs are serviced to completion without being preempted. Also, note that background service causes no preemptions and polling service can cause only one preemption per activation. It is the low preemption rate for polling and background service in Experiment 1 that explains the low context swap overhead.

Now that the low overhead for background and polling service for Experiment 1 has been explained, it is simple to explain the greater overhead for the DS, PE, and SS algorithms. The DS, PE, and SS algorithms are *bandwidth preservation* algorithms that preserve their high-priority execution time until an aperiodic request arrives, at which point the aperiodic task is immediately provided service. Also, none of these algorithms uses its execution capacity for aperiodic service if no periodic tasks are executing. Therefore, when an aperiodic server algorithm provides aperiodic service, it *always* preempts a periodic task. Also, since these algorithms typically provide immediate service to aperiodic tasks, usually only one or a few aperiodic tasks are serviced in one, continuous interval of time. Therefore, for the DS, PE, and SS algorithms, almost every arrival of an aperiodic task causes a periodic task to be preempted. The DS, PE, and SS algorithms provide better aperiodic responsiveness than background and polling service, but in doing so, they also incur more context swap overhead.

Although the context swap overhead for the DS, PE, and SS algorithms in Experiment 1 is greater than the overhead for background and polling service, it is not the same for each algorithm. Of these three algorithms, PE algorithm has the lowest overhead. The PE algorithm preserves its high priority execution time by trading with lower priority periodic tasks. Therefore, the priority of the PE algorithm’s execution time is typically lower than the priority of the DS and SS algorithms. This implies that PE algorithm is not always able to preempt a periodic task to provide immediate aperiodic service and this explains its lower context swap overhead.

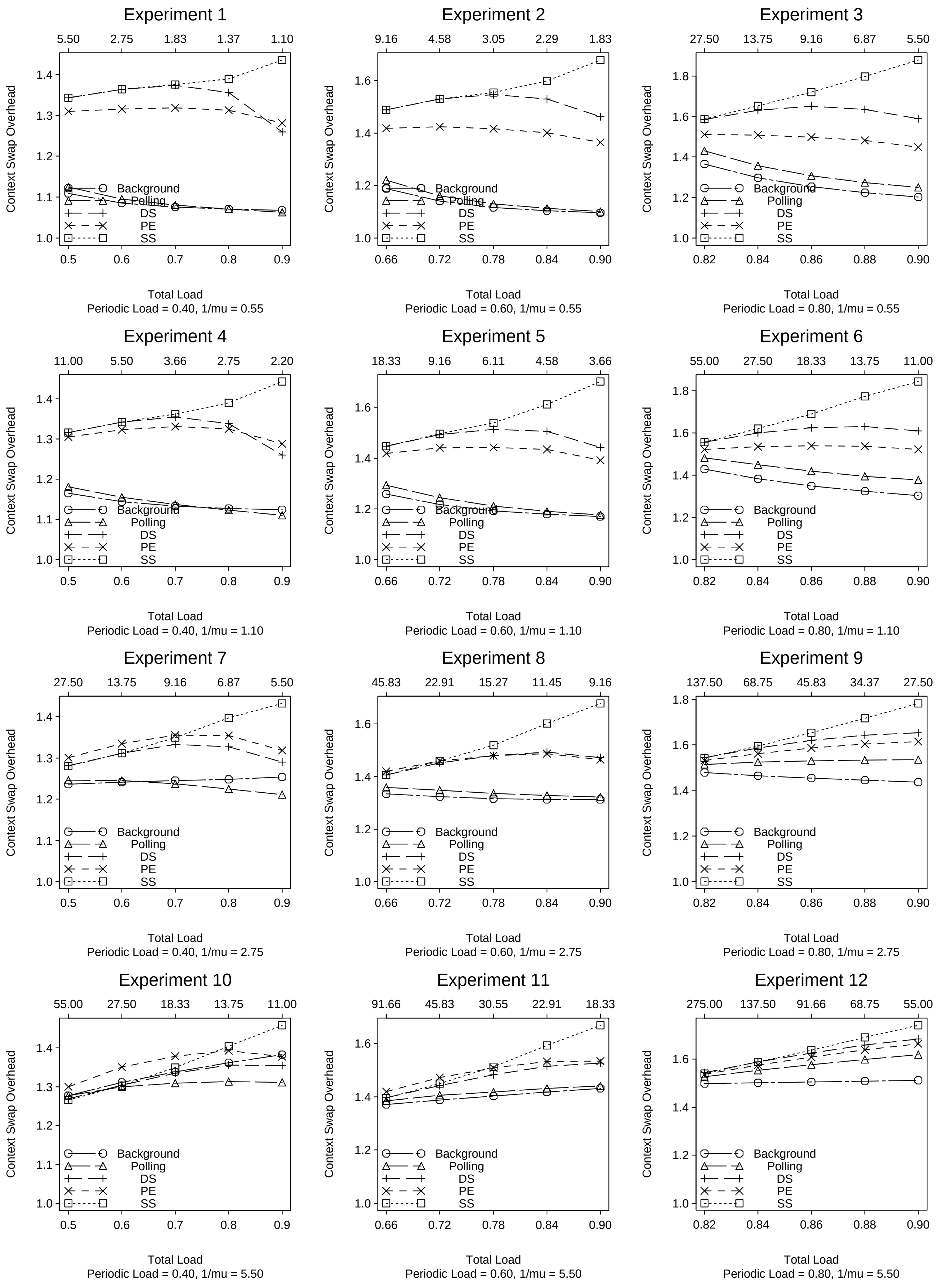


Figure 3-9: Context Swap Overhead for the Background, Polling, DS, PE, and SS Aperiodic Service Algorithms

In Experiment 1, the SS and DS algorithms have the same context swap overhead for aperiodic loads up to 30%, after which the overhead for the SS algorithm becomes higher and the overhead for the DS algorithm becomes lower. The overhead for DS algorithm decreases for these high aperiodic loads because the DS algorithm is overrun (its execution time is exhausted) more often for these cases. Once the server is overrun, many aperiodic jobs must wait to receive service and this reduces the preemption overhead associated with aperiodic service as was explained above for the background and polling algorithms. On the other hand, since the SS algorithm has a larger server capacity than the DS algorithm, its overhead continues to increase with aperiodic load because it is overrun less frequently than the DS algorithm. Also note that for Experiment 1, the SS algorithm is providing better service than the DS algorithm for high aperiodic load, therefore, one would expect a larger context swap overhead. However, as we move to experiments with larger periodic loads where the server capacity of the SS algorithm is not much larger than the DS capacity, the sporadic server still incurs a larger overhead than the DS algorithm for high aperiodic loads. This behavior is due to the different replenishment methods of the DS and SS algorithms and is explained in the next paragraphs.

For Experiments 1, 2, and 3, the overhead for each algorithm becomes higher as the periodic load increases while the mean aperiodic execution time remains the same (note that the overhead range is larger for the graphs with larger periodic loads). The Polling, DS, PE, and SS algorithms incur more overhead because they are more likely to preempt a periodic task since the periodic load has increased. Also, as the periodic load increases, the service opportunities for background service become shorter which means fewer aperiodic tasks can be serviced in each "window" of background service. This increases the probability of periodic task preempting an aperiodic task at the end of a background service window.

As noted earlier, for the periodic loads of 40%, 60%, and 80% in Experiments 1, 2, and 3, the overhead of the SS algorithm becomes larger than the overhead of the DS algorithm at high aperiodic loads. The increase in overhead for the SS algorithm at high aperiodic loads is due in part to the "spreading out" of the SS’s execution time into a collection of smaller execution times. This spreading effect was first described in Section 2.3 and demonstrated in Figure 2-10. If many aperiodic tasks use small amounts of the SS’s execution time (i.e. small compared to the SS’s maximum execution time), then the SS’s execution time is split into many separate and small quantities. Once split, these separate quantities of execution time can be merged together if no request is made for aperiodic service for one SS period. However, this is typically not the case for a high aperiodic load and, as such, the SS’s execution time will remain "spread out". The many separate, small quantities of SS execution time has a detrimental effect upon context swap overhead. Since SS execution time will only be available in separate, small quantities rather than one large quantity, the the server is less likely to be able to service an aperiodic task to completion before exhausting its available execution time. When an aperiodic task is not completed by the SS, then it must be swapped back in later to complete service which increases the context swap overhead. The DS algorithm does not suffer from this problem because, unlike the SS algorithm, its execution time is brought to full capacity at the beginning of every DS period.

For each of the 40% and 60% periodic load experiments, whenever the SS algorithm incurs more context swap overhead, it is also providing a better average aperiodic response time than the DS algorithm. However, for the 80% periodic load experiments, the SS algorithm is providing an equivalent aperiodic response time performance to that of the DS algorithm, but the SS algorithm has a higher context swap overhead than the DS algorithm. Thus, equivalent performance is being provided, but with greater overhead. A system designer may want to consider the cost of this extra overhead for the SS algorithm when selecting the DS or SS algorithm for systems with high periodic and aperiodic loads. However, in the next section, we define and investigate a different service policy that eliminates the partial servicing of aperiodic tasks and lowers the context swap overhead. This different service policy also improves the average aperiodic response time performance of the SS algorithm in most cases.

The context swap overhead as the mean service time is increased is shown as one moves down any one column of graphs in Figure 3-9. For each of the periodic loads, the SS overhead does not change substantially as the mean aperiodic service time increases. In contrast, the overhead for the other algorithms either remains the same or shows an increase as the mean aperiodic service time increases. The reason the SS algorithm’s overhead shows little change with mean aperiodic service time is that the "spreading out" effect for the SS algorithm is the dominant factor which determines its context swap overhead. Although the mean job size has increased, small jobs still occur and split up the SS’s execution time and, a high aperiodic load gives the SS few opportunities to merge together the separate, small quantities of SS execution time. The overhead increases for the other aperiodic service algorithms because the larger mean aperiodic service time lowers the probability that an aperiodic task will be serviced to completion before aperiodic service is suspended which requires additional context swaps for that aperiodic task activation to finish the task.

3.4 The Complete Service Policy for the DS and SS Algorithms

The previous section explained that the exhaustion of server capacity during service of an aperiodic task causes more context swap overhead because the uncompleted aperiodic task must be swapped back at a later time to complete its service. This problem was shown to cause the context swap overhead for the SS algorithm to be larger, in cases of high periodic and aperiodic load, than the overhead for the DS algorithm while, in these same cases, the SS algorithm does not provide a significant performance advantage over the DS algorithm. This suggests that a different service policy that prevents the DS and SS algorithms from providing partial service to an aperiodic task may reduce the context swap overhead for both algorithms, especially the SS algorithm. This section defines and investigates the *Complete Service* (CS) policy for the DS and SS algorithms which requires these algorithms to have enough capacity to complete an aperiodic task before providing any service to the task.

3.4.1 CS Policy: Advantages, Limitations, and Definition

The advantage of the CS policy, as pointed out above, is a lower context swap overhead. However, a reduction in context swap overhead would be no advantage if the average response time performance of either the DS or SS algorithms were increased. Consider what happens with the CS policy when an aperiodic task arrives with a large service time requirement. If the aperiodic server does not have enough capacity to provide service to the large aperiodic task, no aperiodic service is provided. This can cause an increase in the response time for the aperiodic task. But more importantly, note that all high-priority aperiodic service has been suspended until the large aperiodic task's remaining execution time has been lowered enough by background service to allow the high-priority aperiodic server to complete its execution. This could drastically lower the utility of the DS or SS algorithms whenever a large aperiodic job arrives. However, this also provides an opportunity to improve the average response time performance of both these algorithms.

If the aperiodic server provides service to the first aperiodic job in the aperiodic queue that has an execution time requirement that is less than or equal to the server's available execution time, then an improvement in average aperiodic service time is possible. The potential for improvement is demonstrated in Figure 3-10. Figure 3-10 shows the execution of the SS algorithm with the original service policy and with the CS policy for 3 aperiodic requests. Two periodic tasks are present in Figure 3-10: a high-priority periodic task with an execution time of 5 units and a period of 10 units and a low-priority periodic task with an execution time of 3 units and a period of 20 units. The sporadic servers for both the original service policy and the CS policy have an execution time of 3 units and a period of 10 units. The sporadic server executes at the priority level of the high-priority periodic task.

The task execution for the original service policy is shown in the top half of Figure 3-10. Service for the first two aperiodic requests is begun immediately upon their arrival. The first aperiodic request is completed leaving 1 unit of server execution time. The second aperiodic request is also provided immediate service, but it is not completed before the server exhausts its available execution time. The service for the second aperiodic request is continued once the server's execution time is replenished at time = 10 and its service completes at time = 11. The third aperiodic request must wait for the second aperiodic request to be completely serviced before it can be serviced. The average response time for the three aperiodic requests is 11/3 units.

The task execution for using the CS policy is shown in the bottom half of Figure 3-10. The CS policy allows immediate and complete service for the first aperiodic request because it requires only 2 units which is less than the available server execution time of 3 units. The second aperiodic request needs more execution time than the server has available and, as such, the CS policy does not immediately service this request. The third request, however, requires an execution time equal to the amount the server has available and receives immediate and complete service. The CS policy allows the completion of the third request 3 units earlier than the original service policy. At time = 10, the server's execution time is fully replenished and complete service can be provided for the second aperiodic request. The response time for the second aperiodic request is only 1 unit longer than its response time using the original SS policy. At time = 12, the CS sporadic server has serviced all three aperiodic requests with an average response time of 9/3 units which is less than the 11/3 units average response time for the original service policy.

Although it has been demonstrated that the CS policy can improve the average aperiodic service time, it can also degrade average response time performance as is demonstrated in Figure 3-11 using the same periodic task set and sporadic server size from Figure 3-10. The difference for this example is that a third aperiodic request does not occur to take advantage of the available SS capacity between time= 6 and time

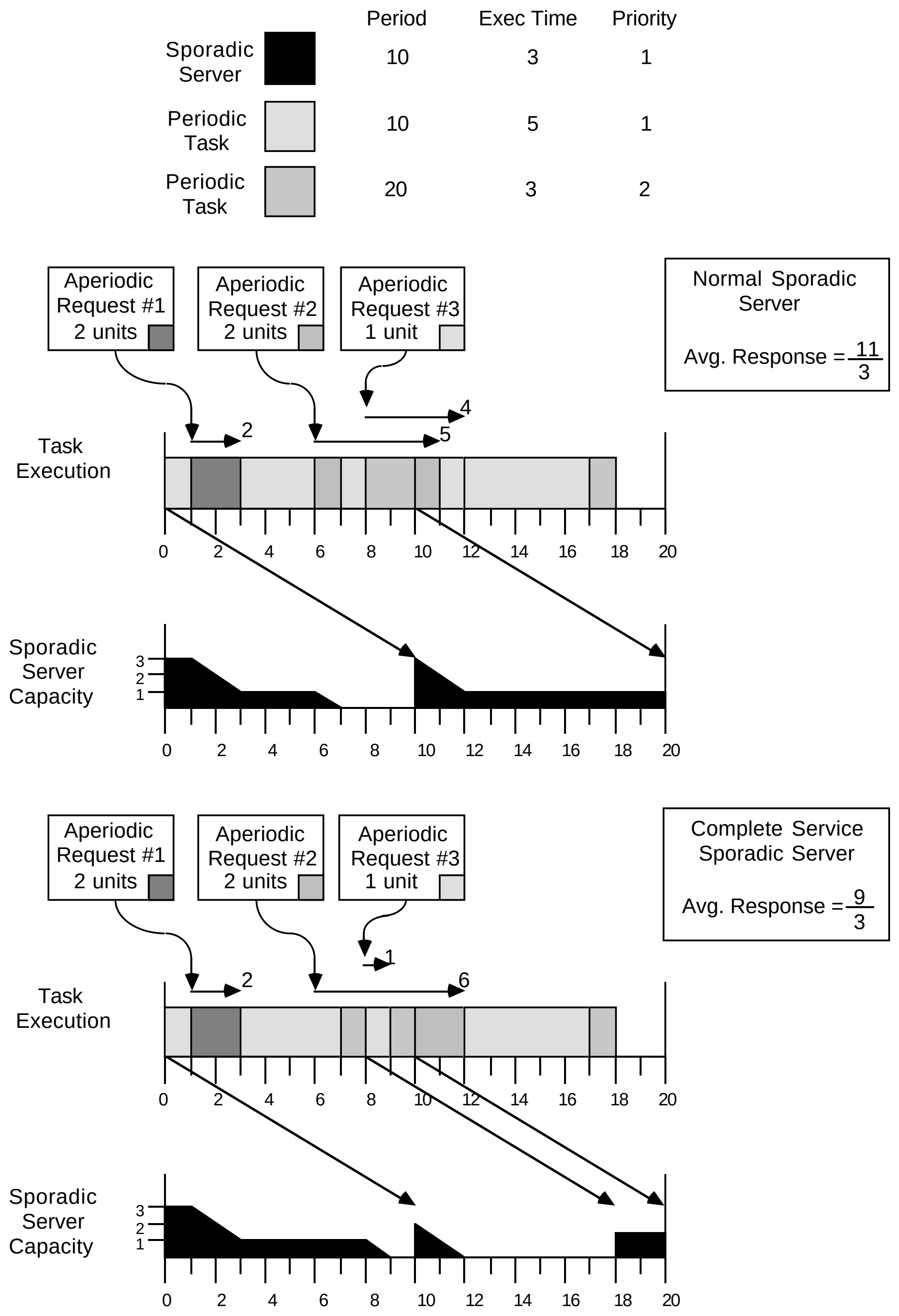


Figure 3-10: Example of Improved Average Response Time with the CS policy for the SS Algorithm

= 10. The response time for the second aperiodic request is 6 units which is greater than the 5 unit response time with the original service policy. Therefore, the CS policy may only improve average aperiodic response time when it is likely that another, shorter aperiodic task will arrive soon enough to use the "saved" SS execution time.

The CS policy definition is as follows. When an aperiodic task is activated, it is placed at the tail of the aperiodic queue. High-priority aperiodic execution time (i.e. DS or SS execution time) is only used to

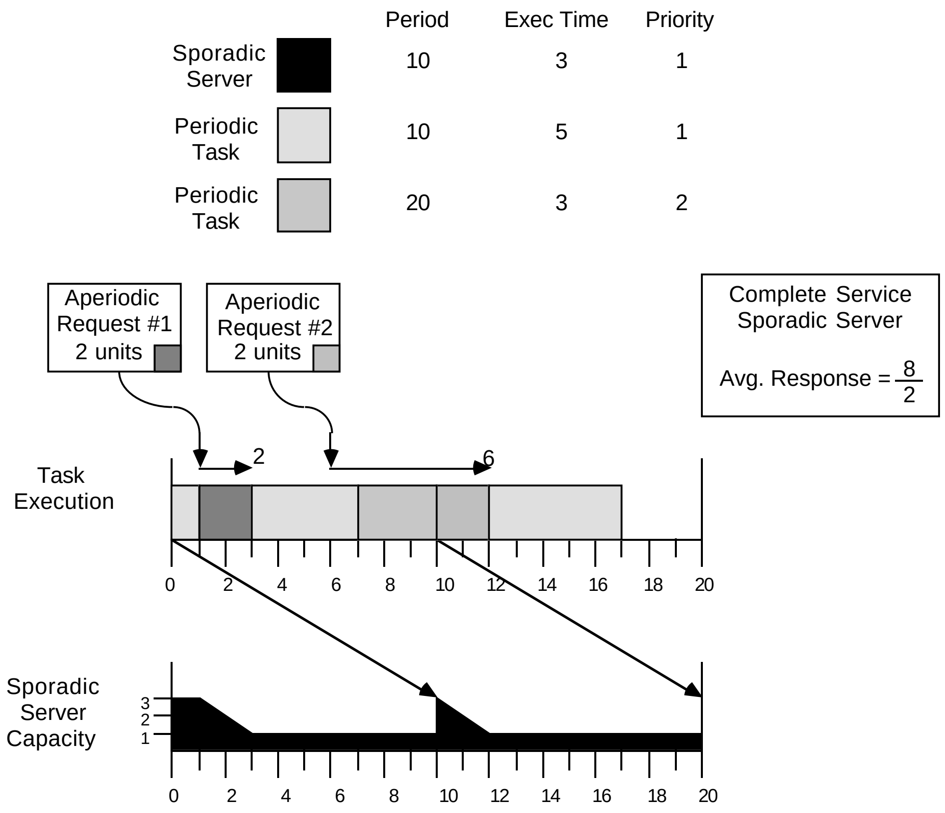


Figure 3-11: Example of Degraded Average Response Time with the CS policy for the SS Algorithm

service aperiodic tasks if the resource is being used by a periodic task. If the resource is idle, any pending aperiodic work at the head of the aperiodic queue is serviced. Any time the server has capacity, aperiodic tasks are pending, and periodic tasks are pending, the aperiodic server will find the first task on the aperiodic queue that has a remaining execution time equal to or less than the available execution time of the server. If such a task is found, it is moved to the head of the aperiodic queue and serviced. If no such task is found, then the pending periodic task with highest priority will be serviced. Thus, with the CS policy, the DS or SS algorithm will never initiate service for an aperiodic task unless it can completely service the request before exhausting its available execution capacity.

3.4.2 CS Policy Experiment Results

The Figures 3-12 and 3-13 present the results for the DS and SS algorithms using the CS policy for the same experiment setup described in Section 3.2. Note that, for most of the graphs in Figure 3-12, both the DS and SS algorithms have a better average response time performance using the CS policy compared to the original service policy (the only exceptions occur in Experiments 9 and 12). Also note in Figure 3-13 that the overhead for the DS and SS algorithms using the CS policy is equal to or lower in every case than the overhead for these algorithms using the original service policy. The CS policy eliminates the context swap overhead that results when an aperiodic server exhausts its available execution time before completing service for an aperiodic task. The CS service policy improves the average aperiodic response time for most cases and provides an equal or lower context swap overhead in all cases.

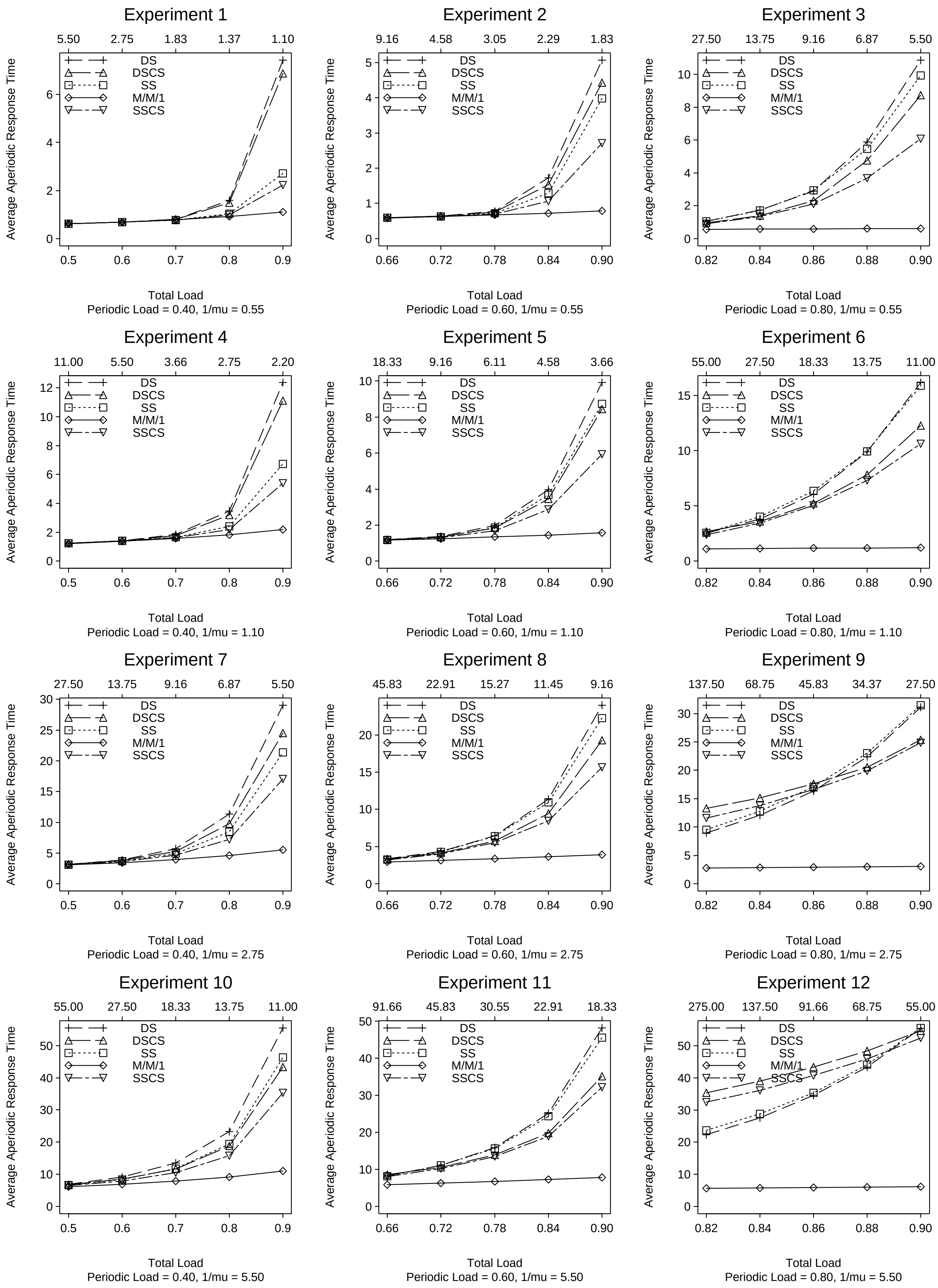


Figure 3-12: Average Response Time for the DS and SS Algorithms with the Complete Service Policy

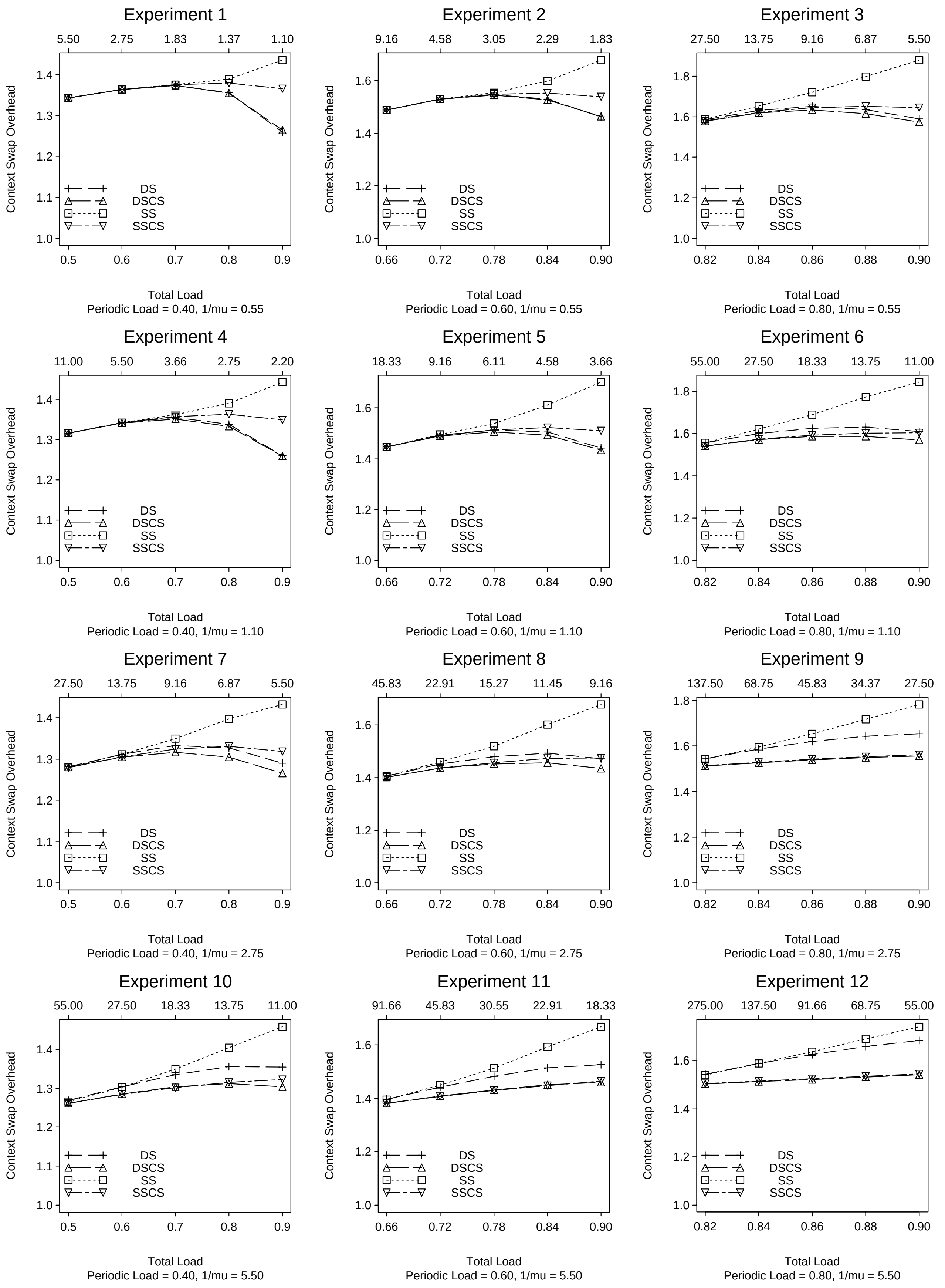


Figure 3-13: Context Swap Overhead for the DS and SS Algorithms with the Complete Service Policy

In Section 3.3, it was shown for several cases that the SS algorithm was offering little or no improvement in average aperiodic response time over the DS algorithm, but was incurring a larger context swap overhead than the DS algorithm. By comparing the response time performance graphs in Figure 3-12 and the context swap overhead graphs in Figure 3-13, it can be seen that this is not true for the CS policy. With the CS policy, for every case where the overhead for the SS algorithm is larger than the overhead for the DS algorithm, the SS algorithm is also providing a significantly better average aperiodic response time than the DS algorithm. Therefore, the additional overhead of the SS algorithm is justified in these cases by the improvement in average aperiodic response time.

The CS service policy results in a lower average aperiodic response time for some cases indicated in Experiments 9 and 12 in Figure 3-12. In these experiments the mean aperiodic service time is a significant percentage of the average DS and SS execution time: 44% (SS) and 48% (DS) for a mean service time of 2.75 and 90% (SS) and 97% (DS) for a mean service time of 5.5. Therefore, for these experiments, the typical aperiodic task consumes a significant portion of the server's execution time making it less likely that the any aperiodic task arriving before the aperiodic server is replenished will find enough high-priority aperiodic service time in the aperiodic server to receive immediate service. Also note that, for each aperiodic load where the average response time for the CS policy is worse than the average response time for the original service policy, the mean interarrival time of the aperiodic tasks (noted on the top scale) is longer than the server's period of 55 units. This implies that when the server is at partial capacity, the next aperiodic request is not likely to occur before the next replenishment and, therefore, cannot take advantage of the CS policy to improve the average response time.

3.5 SS Performance for the Full and Simple Replenishment Policies

The definition of the replenishment policy for sporadic server given Section 2.1 requires the tracking of the active/idle status for all the priority levels in the system in order to provide the earliest replenishment schedule for consumed sporadic server execution time. We refer to this definition as the full SS replenishment policy. Depending upon the application, the SS implementation, and the system platform, this tracking of active/idle status can increase the context swap overhead by increasing the computation time for each context swap (as opposed to the frequency of context swaps studied in Section 3.3). In this section, we describe results of experiments comparing the average aperiodic response time performance of the full SS replenishment policy to a simpler replenishment policy that does not require the tracking of active/idle status on every task switch (i.e. a policy that lowers the context swap overhead necessary to support sporadic servers).

3.5.1 Full and Simple Replenishment Policies: Advantages, Limitations, and Examples

The simple replenishment policy uses the time at which sporadic service is initially consumed to determine the time at which the consumed execution time can be replenished. In contrast, the full replenishment policy uses the time at which the sporadic server's priority level becomes active to determine the replenishment time. We refer to the point from which the replenishment time is determined as the replenishment origin. With either policy, the replenishment time is set to occur one sporadic server period after the replenishment origin. By the definition of the active/idle status of a priority level, the priority level of a sporadic server must be active if it is providing aperiodic service. Therefore, the simple replenishment policy can never provide an earlier replenishment than the full policy and no schedulability problems arise due to the simple replenishment policy.

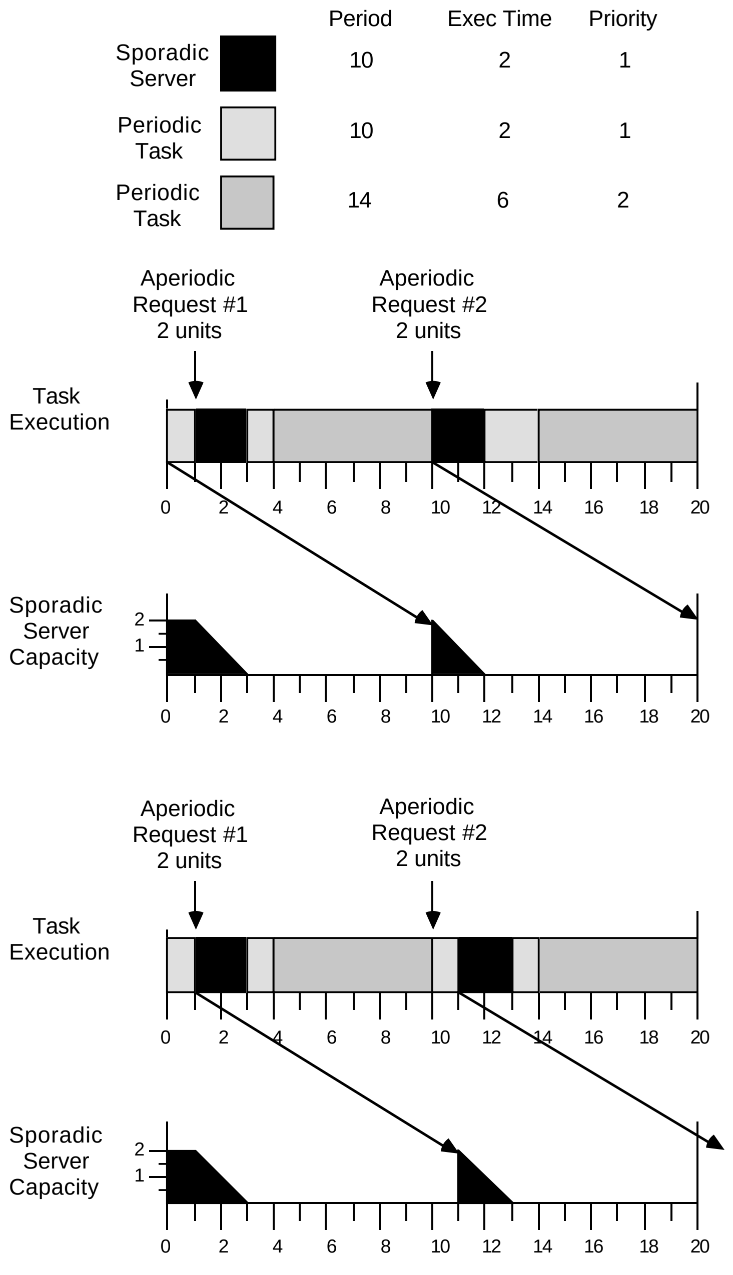


Figure 3-14: Examples of the Full and Simple SS Replenishment Policies

The advantage of the full replenishment policy is that, in some cases, it will allow an earlier replenishment of consumed sporadic server execution time than the simple replenishment policy. It accomplishes this by assuming that the aperiodic task being serviced was activated at the same time that the priority level of the sporadic server became active and lets this time be the replenishment origin. This is demonstrated in the top half of Figure 3-14. In the top half of Figure 3-14, the replenishment time for the sporadic server execution time consumed by the first aperiodic request is determined from time = 0, the point at which the high-priority periodic task began to execute. Since both the sporadic server and the high-priority periodic task are executing at the same priority, the sporadic server's priority level becomes active at time = 0. This full replenishment policy allows a replenishment of 2 units of execution time at time = 10 which enables the sporadic server to provide immediate service for the second aperiodic request.

The simple replenishment policy is demonstrated in the bottom half of Figure 3-14. The replenishment time for the sporadic server execution time consumed by the first aperiodic request is determined from time = 1, the point at which the sporadic server began servicing the first aperiodic request. Thus, the simple replenishment policy replenishes the consumed execution time later than the full replenishment policy which delays the servicing of the second aperiodic request by one unit of time.

3.5.2 Experiment Results: Full vs. Simple Replenishment

The following set of experiments were run to examine the relative performance of these two replenishment policies. The same task sets described in Section 3.2 were used for these experiments. Both the full and simple replenishement policies for the SS algorithm were simulated for these task sets. Since the lower the priority level of the sporadic server, the earlier the replenishment can be set using the full replenishment policy (because more higher-priority work can be underway before sporadic service is granted), the sporadic servers were executed at the top five priority levels, as in Section 3.6.

The results of these experiments are presented in Figure 3-15. The graphs in Figure 3-15 plot the average aperiodic response time of the simple replenishment policy relative to the average response time of the full replenishment policy. A plotted point on these graphs that has a value of 1 for its y-coordinate indicates that the average aperiodic response time of the simple replenishment policy is as good as the performance of the full replenishment policy. A point that has a y-coordinate greater than 1 indicates that the simple replenishment policy has a larger average response time than the full replenishment policy. Note that these graphs are very different from the graphs in Figure 3-16. The graphs in Figure 3-16 were designed to indicate the relative performance of the SS algorithm at different priority levels. This type of comparison is inappropriate using the graphs in Figure 3-15. The important quality to note for the curves in the graphs of Figure 3-15 is the distance of the curve from the horizontal line with a value of 1.0.

The graph for Experiment 1 in Figure 3-15 shows that for an aperiodic load up to 30%, the response time performance of the full and simple replenishment policies are equal. Both replenishment policies have the same performance because over this range of aperiodic load, the sporadic server's execution time is rarely ever exhausted (recall that the SS algorithm performs as well as an M/M/1 system for this range of aperiodic load, see Figure 3-4). Since the server always has capacity to service aperiodic tasks, a replenishment delay of the simple replenishment policy does not decrease its response time performance relative to the full replenishment policy.

At high aperiodic loads in Experiment 1, the average response time of the simple replenishment policy

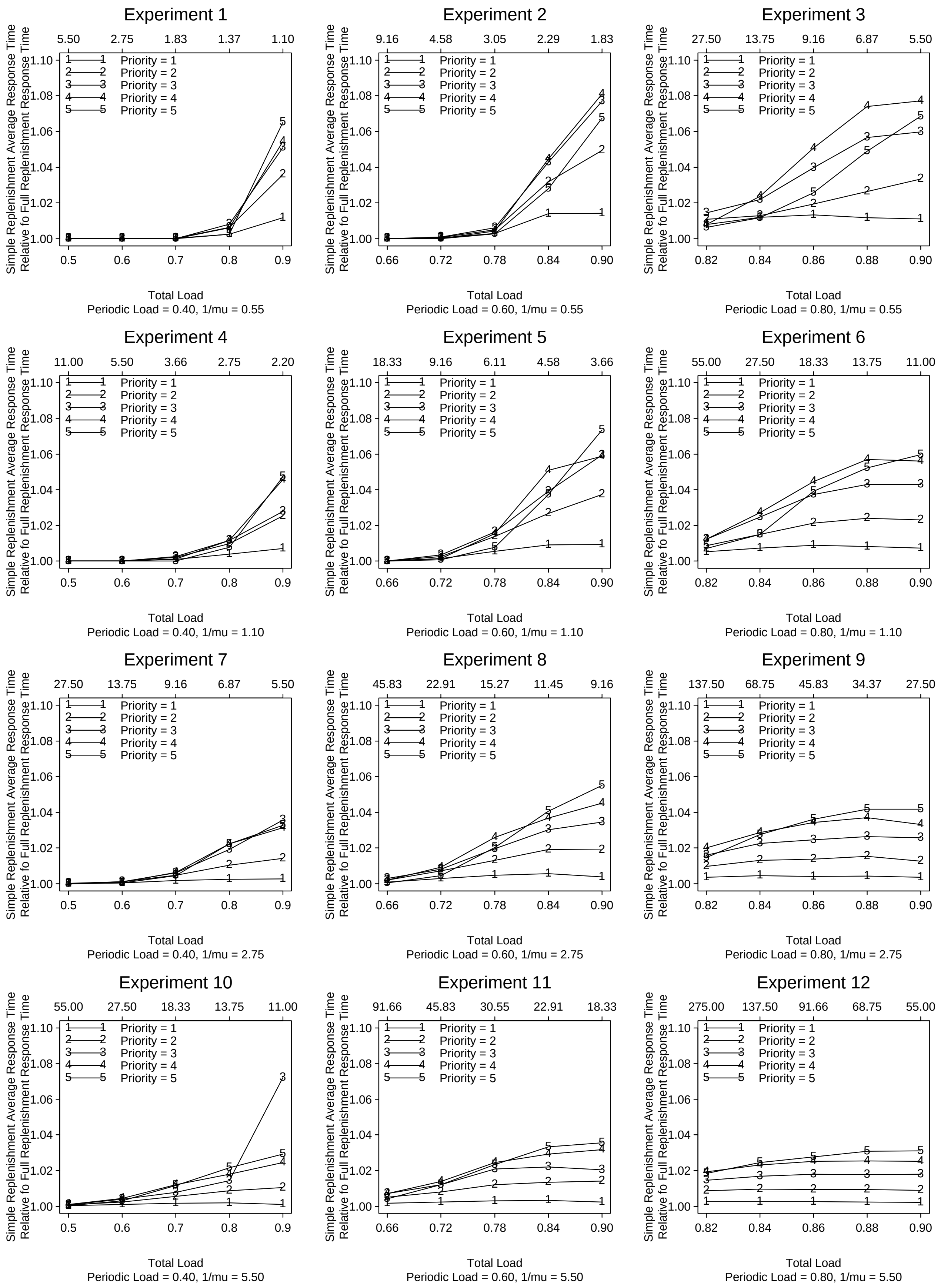


Figure 3-15: Performance Comparison of the Full and Simple SS Replenishment Policies

becomes larger than the the average response time for the full replenishment policy. Also, the lower the server's priority, the larger the increase in average response time over the full replenishment policy. At these high aperiodic loads, the server is frequently exhausted and the advantage of the full replenishment policy becomes apparent. The advantage of the full replenishment policy is greatest for the cases where the replenishment origin is the farthest away from the time at which sporadic service is actually provided. Generally, the length of this "distance" increases as the priority level of the service is lowered because more high priority work can occur before aperiodic service is begun. This is the reason the advantage of the full replenishment over the simple policy is greater at lower priorities. However, the period of the server also is an important factor and depending upon the ratio of the periodic tasks periods and the server's period, average response time can increase or decrease (this effect is discussed later in Section 3.7).

The explanation for the behavior of the full and simple replenishment policies for Experiment 1 can be generalized to explain the results for all of the graphs in Figure 3-15. The full replenishment policy has a response time advantage over the simple policy whenever the combination of aperiodic load and mean service time causes the sporadic server to be exhausted. However, it should be noted that, for high priority sporadic servers (i.e. executing at priority level 1 or 2), response time difference for the two policies is typically less than 2% with a maximum of 5%. Therefore, for high priority sporadic servers it is probably not worth the overhead of tracking the active/idle status of all the priority levels in the system in order to support the full SS replenishment policy. The same statement applies for ranges of aperiodic load and mean service times that do not frequently exhaust the server's capacity. The greatest difference in response time performance for full and simple replenishment policies is 8% which occurs for a low priority sporadic server at a very high aperiodic load. Therefore, we conclude that the overhead to implement the full replenishment policy is only rarely justified by the benefit it returns in improving average aperiodic response time.

3.6 The Effect of Sporadic Server Priority upon Average Aperiodic Response Time

In this section, we investigate the effect of SS priority upon average aperiodic response time performance and context swap overhead. The same task sets described in the previous sections are used again here. To investigate the performance effects of aperiodic server priority, the SS algorithm was simulated for these task sets at five different priority levels, beginning with the highest priority level and moving to lower priority levels. For each task set, five simulations were conducted for each priority level. For each simulation, the period of the sporadic server was set equal to the period of the periodic task with the same priority of the sporadic server and the execution time of the sporadic server was chosen to be the largest value for which the task set remained schedulable. The original service order policy was used for these simulations.

3.6.1 SS Priority Performance Tradeoffs

Generally, the longer the SS period and the lower its priority, the larger the server's execution time (this will be discussed in greater depth in Section 3.7). As such, a tradeoff exists between the larger server size and the priority of the server. The average response time performance graphs presented earlier in Figure

3-4 show that as the mean aperiodic service time increases while maintaining the same execution time (i.e. as one moves down a column of graphs in Figure 3-4), the average aperiodic response time moves away from the ideal M/M/1 curve response for lower values of aperiodic load. Thus, it is an advantage to have a large server execution time compared to the mean aperiodic service time. Also, the context swap overhead data presented in Figure 3-9 shows that, as the mean aperiodic service time increases, the context swap overhead increases because it is more likely that an aperiodic task does not complete before exhausting the server's execution time. Therefore, it may be possible to attain a performance advantage (i.e. a decrease in average aperiodic response time and/or a decrease in context swap overhead) by creating a lower priority server with a larger execution time, especially for task sets where the mean aperiodic service time is a significant portion of the execution time of a high priority sporadic server.

3.6.2 Experiment Results

Figures 3-16 and 3-17 present the results for the experiments described above. To better represent these data, they are plotted relative to the performance of the sporadic server with a priority of 1 (the highest priority level). As such, in each graph the performance of the sporadic server with a priority of 1 appears as a straight line across the graph with a y-axis value of 1.0. Therefore, an improvement in average response time for Figure 3-16 or in context swap overhead for Figure 3-17 is indicated by data points that lie below this priority 1 line. Conversely, a degradation in response time performance for Figure 3-16 or in context swap overhead for Figure 3-17 is indicated by data points that lie above the priority 1 line.

The graphs for Experiment 1 in Figures 3-16 and 3-17 show the results for a small mean aperiodic service time. As expected, the average aperiodic response time increases as the priority level of the sporadic server is lowered. For this experiment, a lower priority server implies that the aperiodic request will be serviced at a later time than with a server running at the highest priority. Also as expected, the context swap overhead is highest for the priority 1 sporadic server because the higher the server's priority, the more often it will preempt lower priority periodic tasks causing more context swap overhead.

The interesting experiments to note in Figure 3-16 are the ones in which the average response time is better for a lower priority sporadic server. From these graphs, it can be seen that for high periodic loads and/or large mean aperiodic service times, the performance of lower priority sporadic servers becomes better relative to the performance of the priority 1 sporadic server. This behavior is particularly noticeable for Experiments 9 and 12. In both of these experiments, the priority 1 sporadic server is always providing a worse average aperiodic response time than the lower priority sporadic servers (except for one case in Experiment 9). Also note that in experiment 9, the response time performance improves as the priority is lowered to priority level 3 and then becomes worse as the priority is lowered further to priority levels 4 and 5. Similar behavior is found in Experiment 12 where the response time performance peaks with priority level 4. Thus, it is possible to attain an improvement in average aperiodic response time by creating a lower priority server with a larger execution time for task sets where the periodic load is high and the mean aperiodic service time is a significant portion of the execution time of a high priority sporadic server. One should also note from Experiments 9 and 12 in Figure 3-17 that the context swap overhead decreases as the priority level is lowered. Thus, for these cases, it is a performance advantage in terms of both improved average aperiodic response time and reduced context swap overhead to create a lower priority sporadic server that has a larger service time.

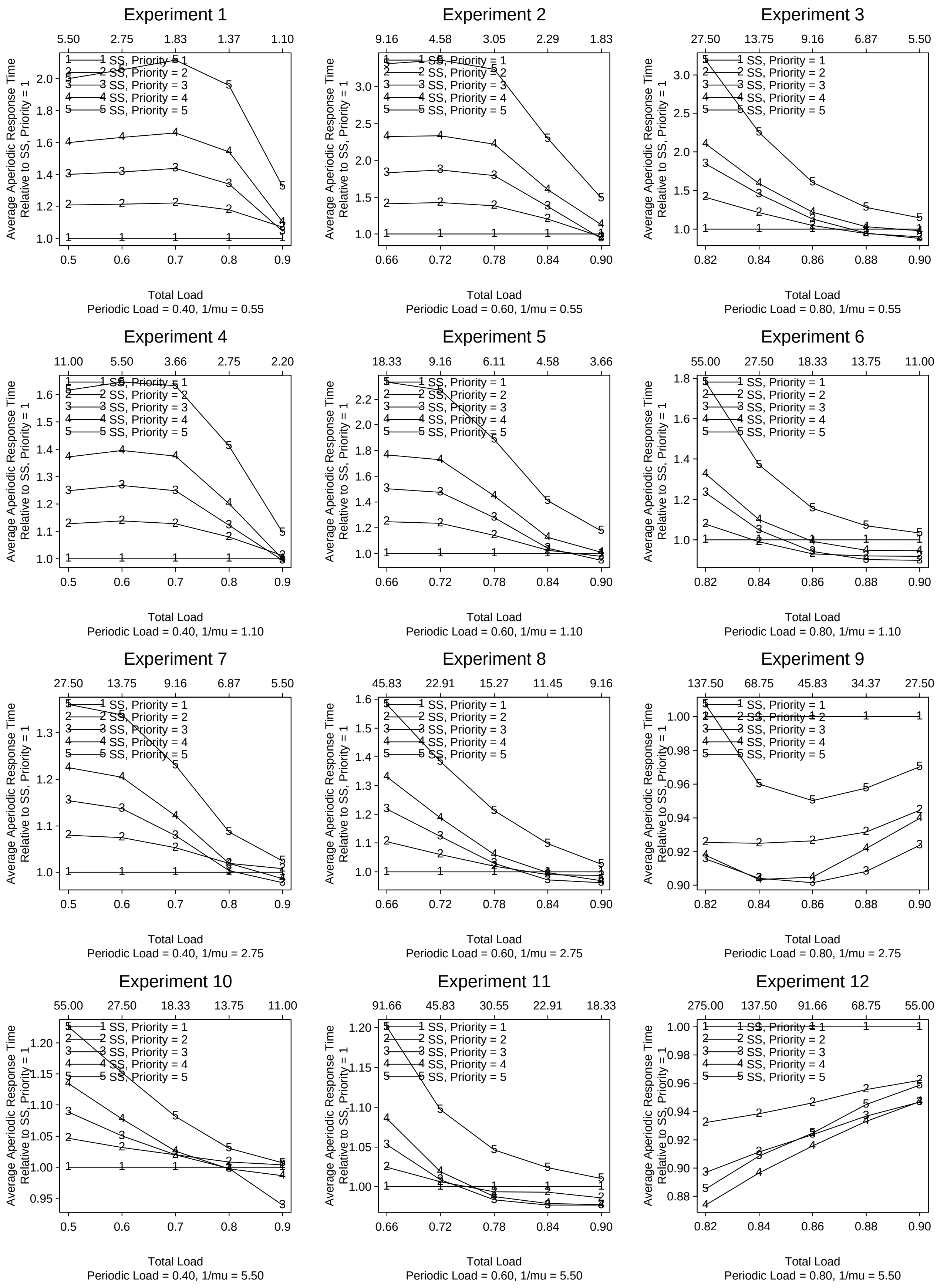


Figure 3-16: Average Response Time Performance for the SS Algorithm Executing at Priority Levels 1, 2, 3, 4, and 5

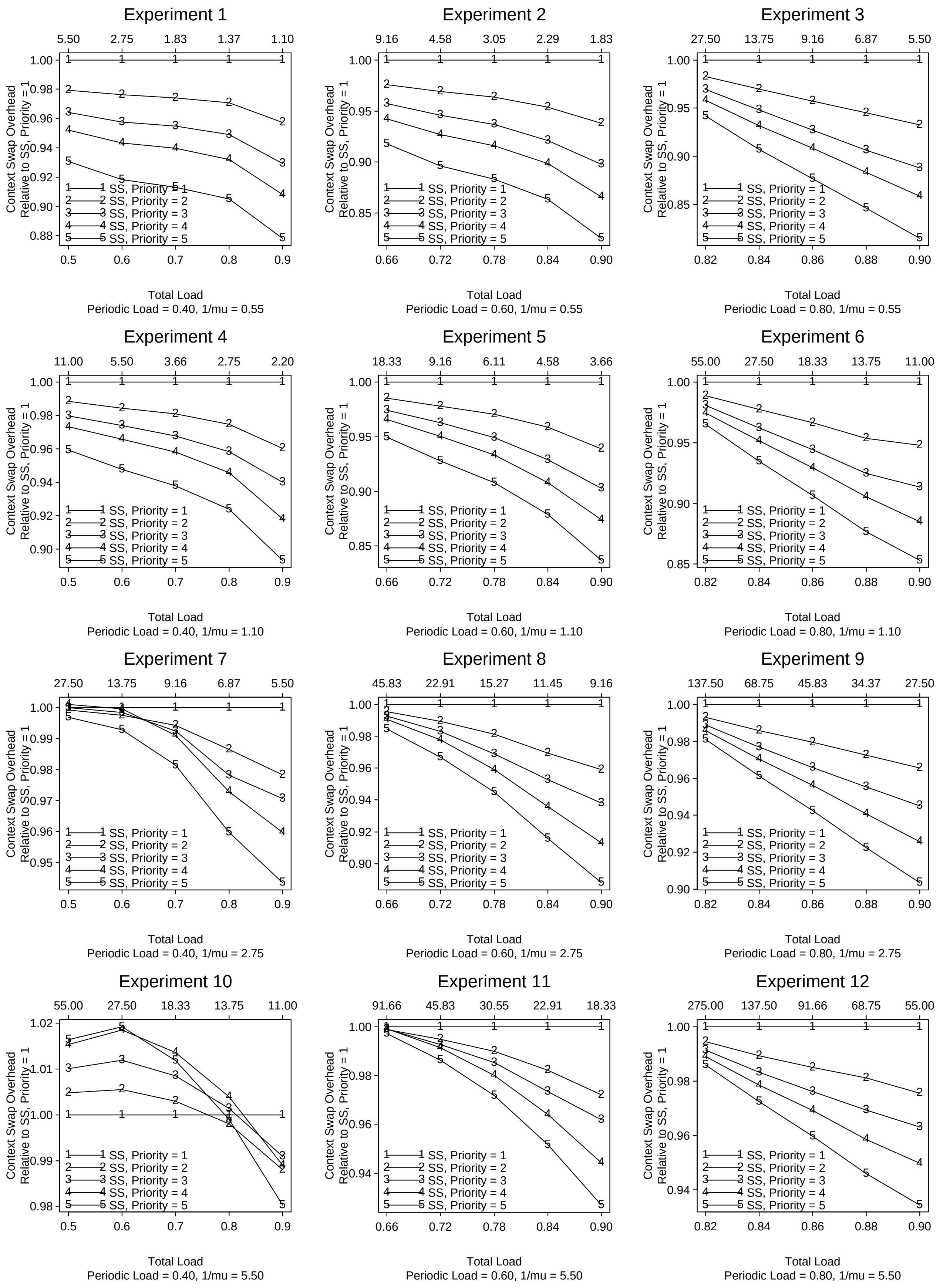


Figure 3-17: Context Swap Overhead for the SS Algorithm Executing at Priority Levels 1, 2, 3, 4, and 5

The interesting results to note in Figure 3-17 are the data for Experiment 10. A general observation can be made from all the graphs in Figure 3-17 that for low to medium aperiodic loads, as the mean aperiodic service time increases, the context swap overhead of the lower priority sporadic servers increases relative to the context swap overhead of the priority 1 sporadic server. This behavior is at its peak in Experiment 10 where the overhead of the lower priority servers actually becomes larger than the overhead of the priority 1 sporadic server. However, one should note that the increase is very slight (a maximum of 2% worse than the priority 1 server). The reason for this behavior is the combination of low priority sporadic servers, a large mean aperiodic service time, and a low periodic load. First, the lower the priority of the sporadic server the more likely it is to be preempted by higher priority periodic tasks. Second, the large mean aperiodic service time implies that the number of preemptions per aperiodic task activation increases. This serves to increase the context swap overhead ratio (described earlier in Section 3.3) because more preemptions occur per aperiodic task activation, not because the total number of preemptions increases. Third, the low periodic load allows larger server execution times than possible at higher periodic loads. All three of these factors imply that the context swap overhead on a per job basis is worse for a lower priority sporadic server as is demonstrated by the graph for Experiment 10.

3.7 Maximum Sporadic Server Execution Time and Size as a Function of Server Period and Priority

Three factors determine the maximum sporadic server execution time: the periodic tasks, the server's period, and the server's priority. Given the sporadic server period and priority, the maximum execution time of the sporadic server is defined to be the largest execution time for which no deadlines are missed.

The two attributes of the periodic tasks that affect the maximum sporadic server execution time are the total utilization of the periodic tasks and the maximum amount of preemption time each periodic task can withstand without missing a deadline. The resource utilization not used by the periodic tasks is the upper bound for the size of the sporadic server. The periods and execution times of all higher priority tasks determine the worst case preemption time a periodic task must be able to tolerate and still meet its deadline. Given the sporadic server's period and priority, the maximum tolerable preemption time of for each task having an equal or lower priority determines the sporadic server's maximum execution time.

The period of a sporadic server has a direct effect upon the server's maximum execution time. Over some range of periods, the maximum sporadic server execution time will generally increase as the sporadic server period increases. However, for a given sporadic server priority that is equal to or greater than the lowest periodic task priority, a the maximum server execution time reaches a peak value as the sporadic server period increases. This peak value corresponds to the maximum amount of additional preemption time caused by the sporadic server that one of the equal and lower priority tasks can withstand and still meet its deadline. Once this maximum amount of preemption time is reached, no increase in sporadic server period will result in an increase in the maximum sporadic server execution time.

The priority of a sporadic server determines the set of periodic tasks which the sporadic server may preempt. To understand the effect of server priority on the maximum server execution time for a sporadic server, consider the case of the maximum sporadic server execution times for two sporadic servers having the same period but different priorities. Over some range of sporadic server periods, the maximum sporadic server execution time for these two sporadic servers can be the same. When the maximum execution times are equal for two sporadic servers with the same period and different priorities, it indicates that the same lower priority task will miss its deadline if the execution time of either sporadic server is increased. Given the same sporadic server period, the maximum sporadic server execution time for two different priority levels can be different for one of two reasons. First, for a short sporadic server period and a low sporadic server priority, the maximum sporadic server execution time may be zero because the worst case preemption by the higher priority tasks may leave no time available for the sporadic server to execute within its short period. Second, the higher-priority sporadic server has reached its peak value (as described in the previous paragraph) which is lower than the peak value of the lower priority sporadic server.

3.7.1 Determination of Maximum Server Execution Time

A brief description of the method used to determine the maximum server execution time is helpful to explain the data presented in the next section.

A series of critical zone analysis tests are performed to determine the maximum sporadic server execution time. For these tests, the sporadic server is treated as a periodic task with the same execution time and period (see Section 2.3). Recall that a critical zone analysis is used to test the schedulability of a periodic task set. This analysis consists of activating all periodic tasks at the same instant in time (referred to as the critical instant) and verifying that each periodic task will meet its first deadline. If all tasks meet their first deadlines, the task set is schedulable. Given the sporadic server’s period and priority, the maximum sporadic server execution time is found be assuming an initial server execution time and performing a binary search using the critical zone analysis test to indicate whether to increase or decrease the sporadic server’s execution time. The largest execution time found for which the task set remains schedulable is then chosen as the sporadic server’s execution time.

3.7.2 Maximum Server Execution Time Examples

The maximum server execution time as a function of the sporadic server’s periodic and priority will first be discussed for the simple periodic task set presented in Figure 3-18. The periods of the tasks in this task set were chosen so that the ratio of the period of one task to the next higher priority task is always 2. For each total periodic utilization shown (i.e., the 40%, 60%, and 80% total periodic loads), each periodic task has the same utilization (i.e., 4%, 6%, and 8% respectively). The motivation for structuring the task set in this manner will become evident when we calculate the peak sporadic server execution time and server period at which this peak occurs for the five highest periodic task priorities.

The peak sporadic server execution times and the server period at which this peak occurs for the task set of Figure 3-18 can be computed as follows. For each periodic task, determine the difference between the task’s period and the total execution time consumed by the task and all higher priority tasks during the task’s first period using a critical zone phasing for all periodic tasks. This difference is the maximum amount of additional preemption the task could withstand and still meet its deadline. This corresponds to

		Execution Time		
Task	Period	40%	60%	80%
1	5	0.2	0.3	0.4
2	10	0.4	0.6	0.8
3	20	0.8	1.2	1.6
4	40	1.6	2.4	3.2
5	80	3.2	4.8	6.4
6	160	6.4	9.6	12.8
7	320	12.8	19.2	25.6
8	640	25.6	38.4	51.2
9	1280	51.2	76.8	102.4
10	2560	102.4	153.6	204.8

Figure 3-18: Simple Task Set

the peak sporadic server execution time for a sporadic server with the same priority as the periodic task. Because of the harmonic relationship of the periods in this task set, we know that the maximum utilization of the periodic tasks is 100%. This allows us to calculate the sporadic server period at which the peak server execution time occurs by dividing the peak server execution time by 1 minus the utilization of the periodic tasks. For example, these calculations for the five highest periodic task priority levels for a periodic utilization of 40% are as follows:

Peak Server Execution Time:

Priority 1:	5 - 0.2 = 4.8
Priority 2:	10 - (10/5)0.2 - 0.4 = 9.2
Priority 3:	20 - (20/5)0.2 - (20/10)0.4 - 0.8 = 17.6
Priority 4:	40 - (40/5)0.2 - (40/10)0.4 - (40/20)0.8 - 1.6 = 33.6
Priority 5:	80 - (80/5)0.2 - (80/10)0.4 - (80/20)0.8 - (80/40)1.6 - 3.2 = 64.0

Server Period for Peak Execution Time:

Priority 1:	4.8/(1 - 0.40) = 8.0
Priority 2:	9.2/(1 - 0.40) = 15.3
Priority 3:	17.6/(1 - 0.40) = 29.3
Priority 4:	33.6/(1 - 0.40) = 56.0
Priority 5:	33.6/(1 - 0.40) = 106.7

The corresponding calculations for the 60% and 80% periodic loads are done in the same manner by substituting the appropriate periodic task execution times and the appropriate periodic utilization.

Figure 3-19 presents the maximum server execution time and size as a function the periodic load and the sporadic server's period and priority for the periodic task set of Figure 3-18. For each graph in Figure 3-19, the x-axis represents the sporadic server period. The left column of graphs presents the data for the maximum server execution time and the right column of graphs presents the corresponding data for maximum server size. Each row of graphs represents a different periodic load. Each downward pointing arrow in a graph indicates the period of a periodic task. The priority of the periodic task is indicated above the arrow with priority 1 being the highest priority.

Referring to the lop left graph of Figure 3-19, one can see that the above calculations for the peak sporadic server execution times and periods accurately predict the data in the graph. One can also see each of the characteristics of the maximum sporadic server execution time that were explained earlier in the graphs of Figure 3-19. From the maximum server size graph, we see that the peak size is equal to the resource utilization left unused by the periodic tasks (this represents the best case for the server size, and is not typically attainable with a random periodic task set). The general increase in the maximum server execution time as the server period increases until a peak value is reached can be seen. The cases where the maximum server execution time is different for sporadic servers with the same period but different priorities can also be seen. First, note that, for an 80% periodic load and a server period of 5 units, the priority 5 sporadic server has a zero maximum execution time, whereas the higher priority sporadic servers have a nonzero maximum execution time. This can be explained by noting that for a critical zone phasing, the higher priority periodic tasks execute for 5 units during the first 5 units, leaving no time available for the low priority sporadic server to execute. Second, note that the different peak values for sporadic servers with the same period but different priorities allows the lower priority sporadic server to have a higher maximum execution time.

The simple task set of Figure 3-18 demonstrates the types of behavior seen in the maximum server execution time as a function of server period and priority, but the behavior for a random task set is not as easily calculated and does not necessarily appear similar to the graphs shown 3-19. The period ratios and execution times for a random periodic task set typically do not allow the straightforward calculation of the peak maximum execution times and the server periods at which the peaks occur. For example, it is often the case that the maximum schedulable utilization for the periodic task set and one sporadic server is less than 100% and one must examine the critical zone to determine the maximum sporadic server execution time given the server's period and priority (as was described in Section 3.7.1).

Figure 3-20 presents the maximum server execution time and size as a function the periodic load and the sporadic server's period and priority for task set 0[*] from the experiments described in Section 3.2. Two differences between the shapes of the maximum execution time curves of Figures 3-19 and 3-20 are immediately obvious. The different priority levels reach different peak maximum execution times only for a 40% periodic load in Figure 3-20 whereas different peak maximum execution times for the different priority levels occurs for all the periodic loads presented in Figure 3-19. Also, note that even before the peak maximum server execution time is reached, the maximum server execution time does not always show an increase when the server period is increased.

[*]A summary of the task set parameters is included in Appendix I.

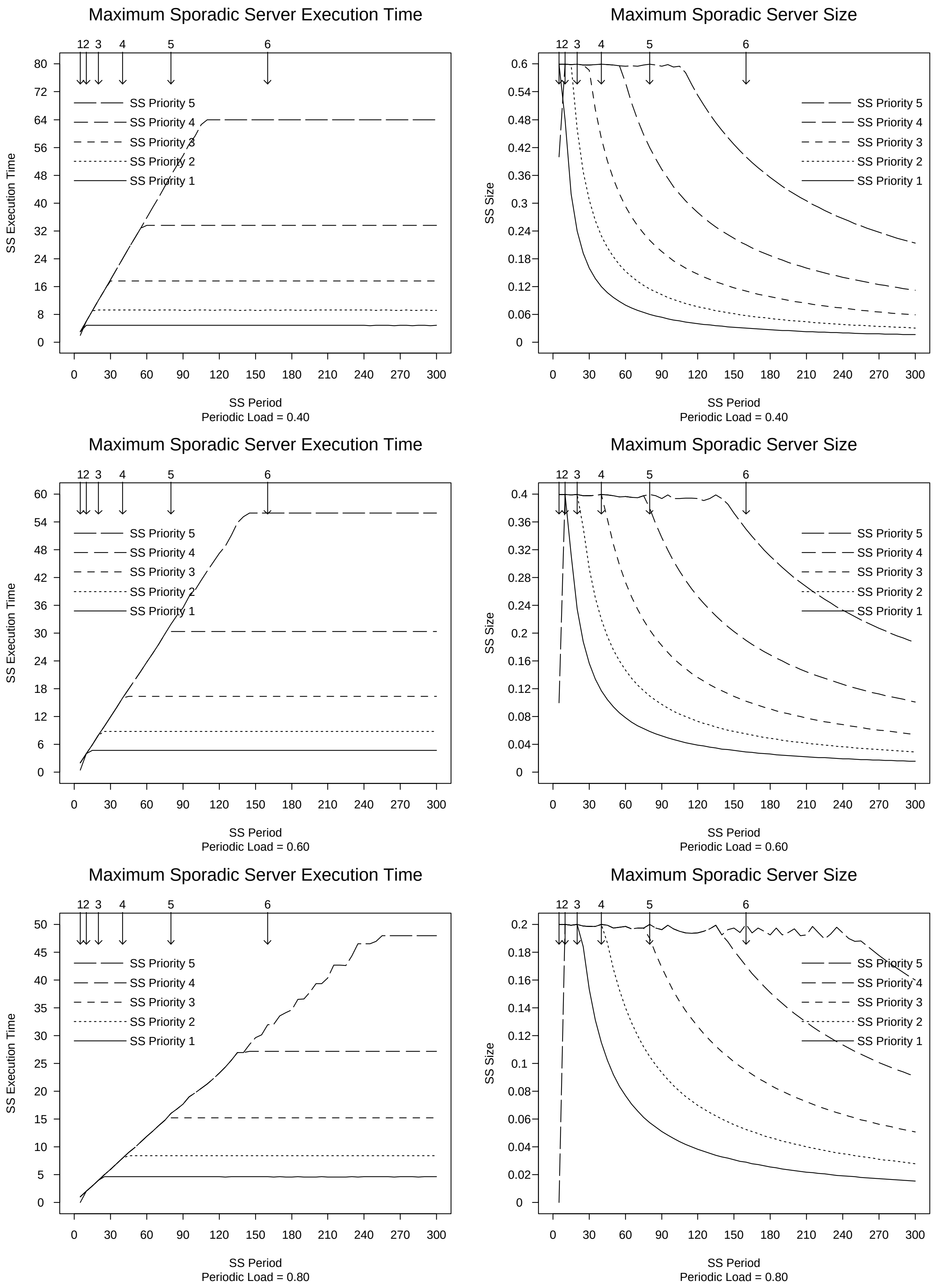


Figure 3-19: Maximum SS Execution Time and Size as a Function of SS Period and Priority for A Simple Task Set

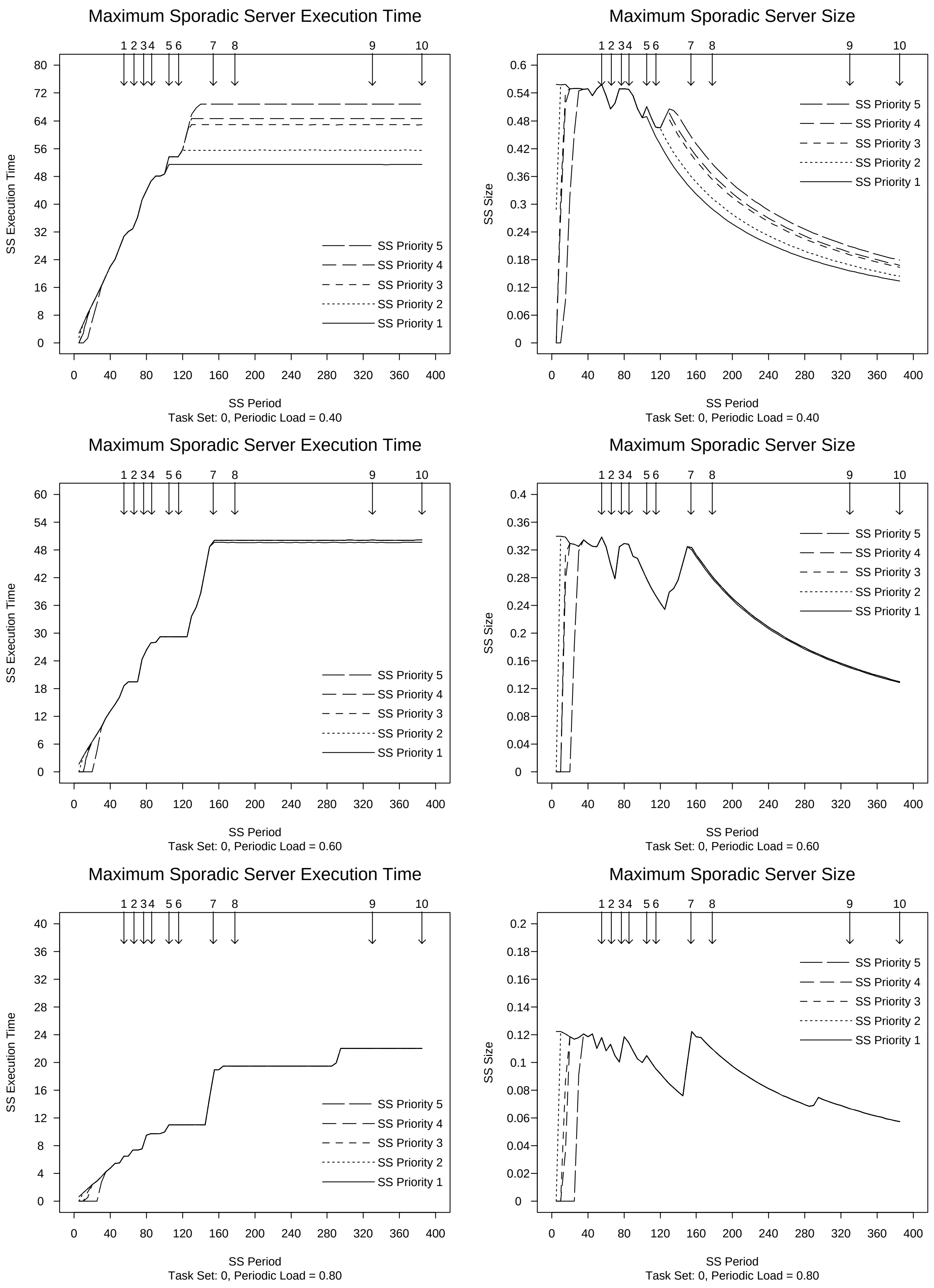


Figure 3-20: Maximum SS Execution Time and Size as a Function of SS Period and Priority for Task Set 0

The different priority sporadic servers for the 60% and 80% periodic load graphs in Figure 3-20 show little or no difference in peak maximum server execution times because the same lower priority periodic task will miss its deadline if any of these sporadic server's increases its execution time. Different peak maximum execution times for two sporadic servers having the same periodic but different priorities occurs because an increase in the execution time of the higher priority sporadic server would cause a periodic task with a priority higher than the low priority sporadic server to miss its deadline. For the 80% periodic load in Figure 3-20, an increase in the execution time for any of the sporadic servers shown would cause a periodic task with a priority lower than all the sporadic server priorities to miss its deadline. Therefore, no difference is seen in the peak maximum server execution times.

The maximum server execution time does not always show an increase with an increase in server period. This can occur, as described earlier, when the server has reached its peak maximum execution time. This can also occur before the peak maximum execution time is reached. Examples of this occur several times for the 80% periodic load in Figure 3-20 (e.g., the maximum server execution time is constant over the range of server periods from 115 to 150). This occurs when the length of the server's period causes its second or later activations during the critical zone of a lower priority task to occur at a point during which preemption by higher priority tasks cannot be withstood during the lower priority task's critical zone. Once the period of the server is increased so that it is activated after this point, the server's execution time may be increased.

The behavior of maximum sporadic server execution time as a function of server period will be used in the next section to investigate the average aperiodic response time performance as a function of server period and execution time.

3.8 SS Performance as a Function of Server Period

In this section, we investigate the effect of the sporadic server period upon average aperiodic response time. This is also an investigation of the effect of the server execution time upon average response time because the server's period determines the maximum server execution time.

Several factors must be considered when choosing the sporadic server period. A tradeoff exists between the server execution time and the server replenishment delay. For the same server size, a longer server period implies a larger server execution time and longer replenishment delay. The larger service time allows more continuous aperiodic service to be provided without any delays due to periodic task execution. However, the longer server period means a longer aperiodic service delay when the server is overrun. For a given server size, a shorter server period reduces the amount of continuous aperiodic service that can be provided without delays due to periodic task execution, but it also reduces the replenishment delay when the server is overrun. The experiments reported here investigate the effects of these tradeoffs upon average aperiodic response time.

For this study, we used task set 1 and the same aperiodic workload characteristics from the experiments described in Section 3.2. The maximum server execution time and size as a function of server period for task set 1 are presented in Figure 3-21. The layout of the graphs in Figure 3-21 is the same as found in Figures 3-19 and 3-20.

Referring to the graphs in Figure 3-21, we see that the manner in which the maximum server execution time and size varies as the server period increases is different for the different periodic loads. For a periodic load of 40% (the top row of graphs), the server execution time for the priority 1 sporadic server shows a fairly steady increase as the server period increases until the peak maximum sporadic server execution time of approximately 50 units is reached at a server period of 94 units. The corresponding server size graph shows that the server size over this range of periods varies slightly around a value of 54%. The graphs for a 60% periodic load are similar to the 40% graphs, but the rate of increase in maximum server execution time with server period is lower than that for the 40% periodic load. This implies that a longer server period is necessary with the 60% periodic load to attain a given maximum server execution time. The graphs for the 80% periodic load are very different from the 40% and 60% graphs. For several ranges of server periods, the maximum server execution time for the 80% periodic load shows no increase with the server period. Over these ranges the server size decreases as the server's period increases. However, once the maximum server execution time begins to increase, it rapidly reaches another plateau and the server size approaches its maximum value. By examining how the average aperiodic response time varies for each of these periodic loads as we select different sporadic server periods (and the corresponding maximum server execution times), we will see the effect of server execution time, server period, and server size upon average aperiodic response time.

The sporadic servers for these experiments were chosen in the following manner. Ten maximum sporadic server execution times were chosen ranging from 5 units to 50 units with an increment of 5 units. As can be seen from Figure 3-21, this range of maximum server execution times almost completely covers entire range of maximum server execution times for task set 1 at 40%, 60%, and 80% periodic loads. Each sporadic server period was chosen to be the shortest period at which the desired server execution time was attainable. This method of server period selection yields the largest server size for the given server execution time. For each of these experiments the sporadic server was given the highest priority.

Figure 3-22 presents the results of these experiments for a mean aperiodic service time of 0.55 units. The top row of graphs presents the average aperiodic response time for each of the sporadic servers described above for the periodic loads of 40%, 60%, and 80%. The bottom scale on these graphs indicates the sporadic server execution time and the top scale indicates the sporadic server period. Note that the bottom scale on these graphs is linear in server execution time but that the top scale is not linear in the server period. Each curve on these graphs represents the average aperiodic response time for a given aperiodic load (indicated at the left end of the curve). The bottom row of graphs presents the sizes of the sporadic servers used in these experiments for the periodic loads of 40%, 60%, and 80%. The height of the vertical lines in the bottom row of graphs indicates the sporadic server size. The top and bottom scales for the bottom row of graphs are the same as those for the top row of graphs.

Referring to the graph for Experiment 1 in Figure 3-22, we see that the average aperiodic response time improves for most cases as the server execution time is increased. The cases for which the average aperiodic response time increases as the server execution time increases are also cases in which the server size decreases as the server execution time increases (note the curve for a 50% aperiodic load as the server execution time changes from 10 to 15, from 35 to 40, and from 45 to 50). As the server execution

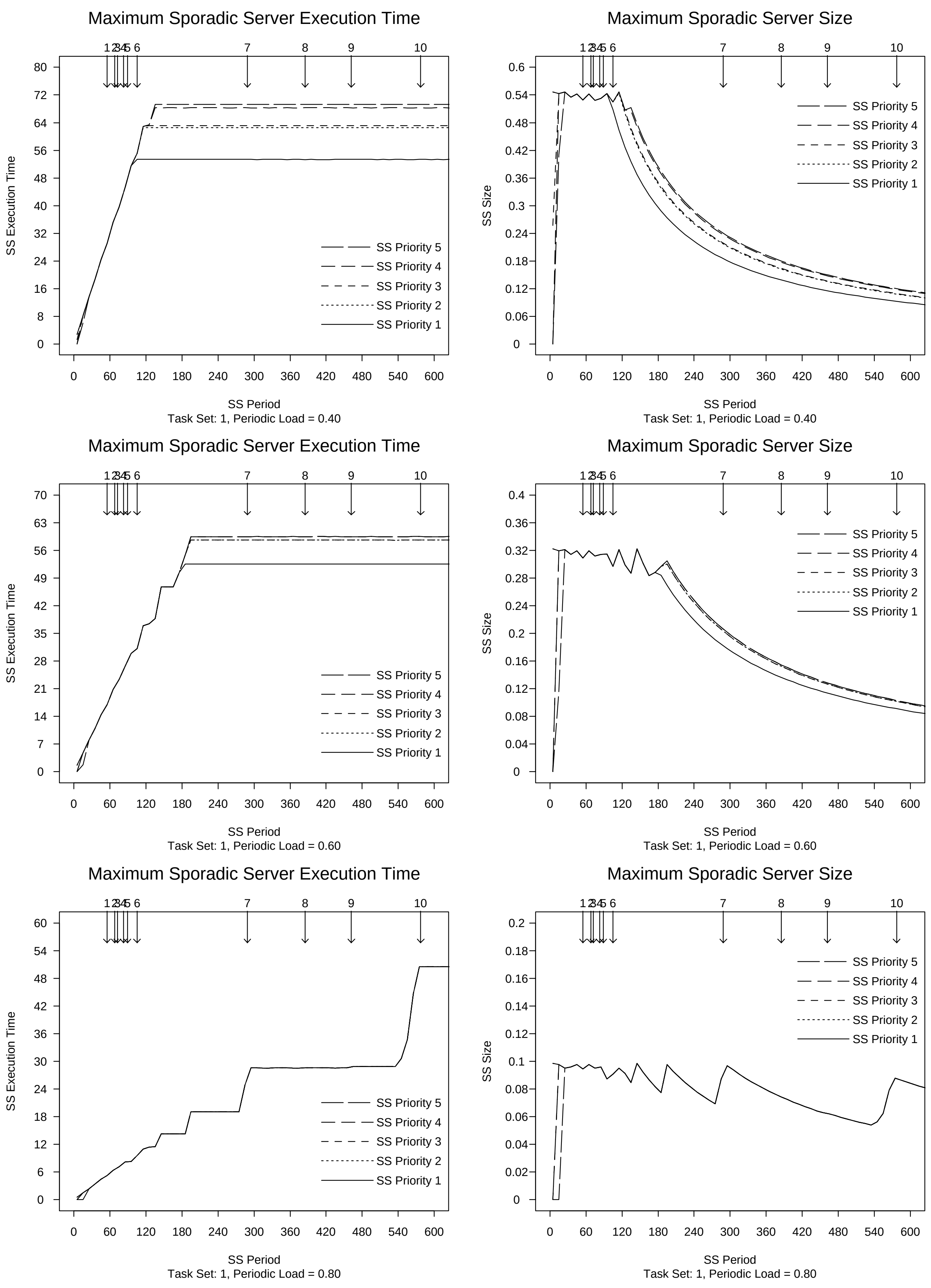


Figure 3-21: Maximum SS Execution Time and Size as a Function of SS Period and Priority for Task Set 1

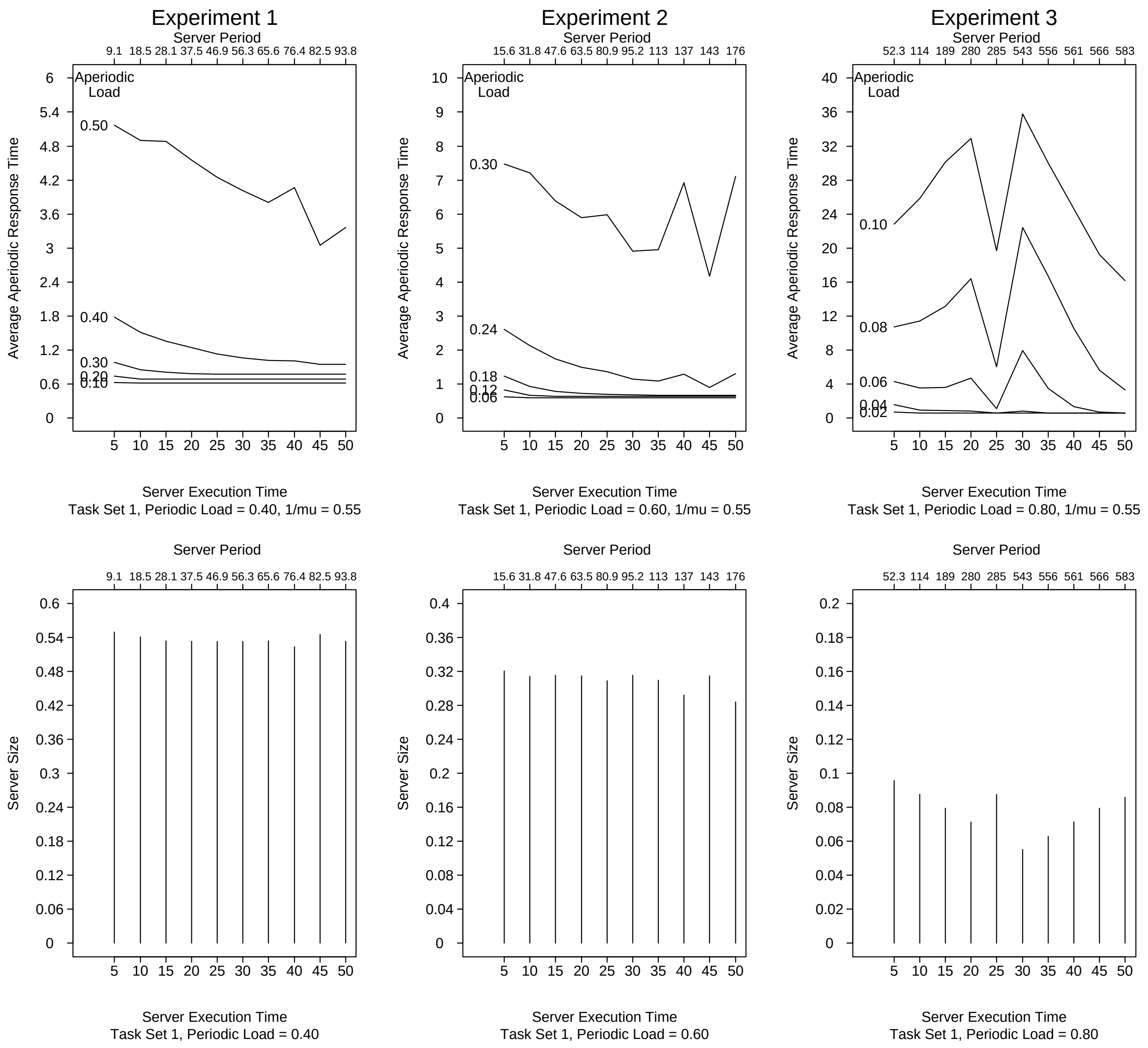


Figure 3-22: Average Aperiodic Response Time as a Function of SS Execution Time for Task Set 1 and a Mean Aperiodic Service Time of 0.55 Units

time increases from 15 to 35 units, the server size is constant and the average aperiodic response time improves or remains unchanged for all aperiodic loads. This implies that, for a given server size, a larger server execution time improves the average aperiodic response time more than the longer replenishment delay (due to the longer server period) degrades the average response time. One should also note that magnitude of improvement in average response time is greater for higher aperiodic loads. This can be explained by noting that the higher aperiodic load, the more frequently a sporadic server with a small execution time will be overrun than a sporadic server with a larger execution time. Thus, these data imply that for a constant sporadic server size, a higher frequency of server overruns is more detrimental to response time performance than a longer replenishment delay.

The average response time data for Experiment 2 in Figure 3-22 show a behavior very similar to the data for Experiment 1. The average aperiodic response time improves or remains the same whenever the sporadic server's execution time is increased while maintaining the sporadic server size constant. Note that, for the cases where a decrease in sporadic server size increases the average response time, the magnitude of the increase is larger than was observed in Experiment 1. The reason for this behavior is that the percentage decrease in server size is larger for the cases shown in Experiment 2 than for the corresponding cases in Experiment 1 and, therefore, one would expect the average response time degradation to be larger.

The data for Experiment 3 has a very different appearance from the data for Experiments 1 and 2, but the same conclusions concerning the average aperiodic response time as a function of server execution time and server size hold for these data. The maximum sporadic server size as a function of server period is very different for the 80% periodic load. This can be seen by comparing the server size graphs in Figure 3-22. The sporadic server sizes for server execution times of 5, 25, and 50 are roughly the same. Noting the corresponding average aperiodic response times for these server execution times shows an improvement in average aperiodic response time as the server execution time increases. Over the range of server execution times from 5 to 20, the server size decreases and an increase in average aperiodic response time is observed. As noted earlier, the percentage decrease in server size for the 80% periodic load is larger than for either the 40% or 60% periodic loads and this explains the larger relative increase in average aperiodic response time as the server size decreases. Over the range of server execution times from 30 to 50, the sporadic server size increases and a steady improvement average aperiodic response time is observed.

Figure 3-23 presents the average aperiodic response time as a function of server period and execution time for task set 1 with periodic loads of 40%, 60%, and 80% and mean aperiodic service times of 0.55, 1.10, 2.75, and 5.50 units. The top row of graphs in Figure 3-23 present the same data found in the top row of graphs in Figure 3-22. Note that the server sizes for each row of graphs in Figure 3-23 is the same as indicated in the bottom row of graphs in Figure 3-22.

From Figure 3-23, we see that behavior of the average aperiodic response time as a function of sporadic server execution time and period for larger mean aperiodic service times mimics the behavior for a mean service time of 0.55 units. Therefore, from these data we conclude that, for a given server size, a larger server execution time can provide a better average aperiodic response time.

3.9 Selecting the Sporadic Server Period, Execution Time, and Priority

This section uses the experience gained from the experiments and analysis of the previous sections of this chapter to create a set of guidelines for choosing the period, execution time, and priority of a sporadic server to provide good aperiodic response time performance. Although these guidelines do not cover all conditions, they should work well for many task sets.

From the experiments reported in this chapter, the following general statements concerning the design of a sporadic server can be made:

1. The larger the size of a sporadic server, the more high-priority execution time it can provide for aperiodic service.
2. For a given server size, a larger server execution time can improve the average response time performance.

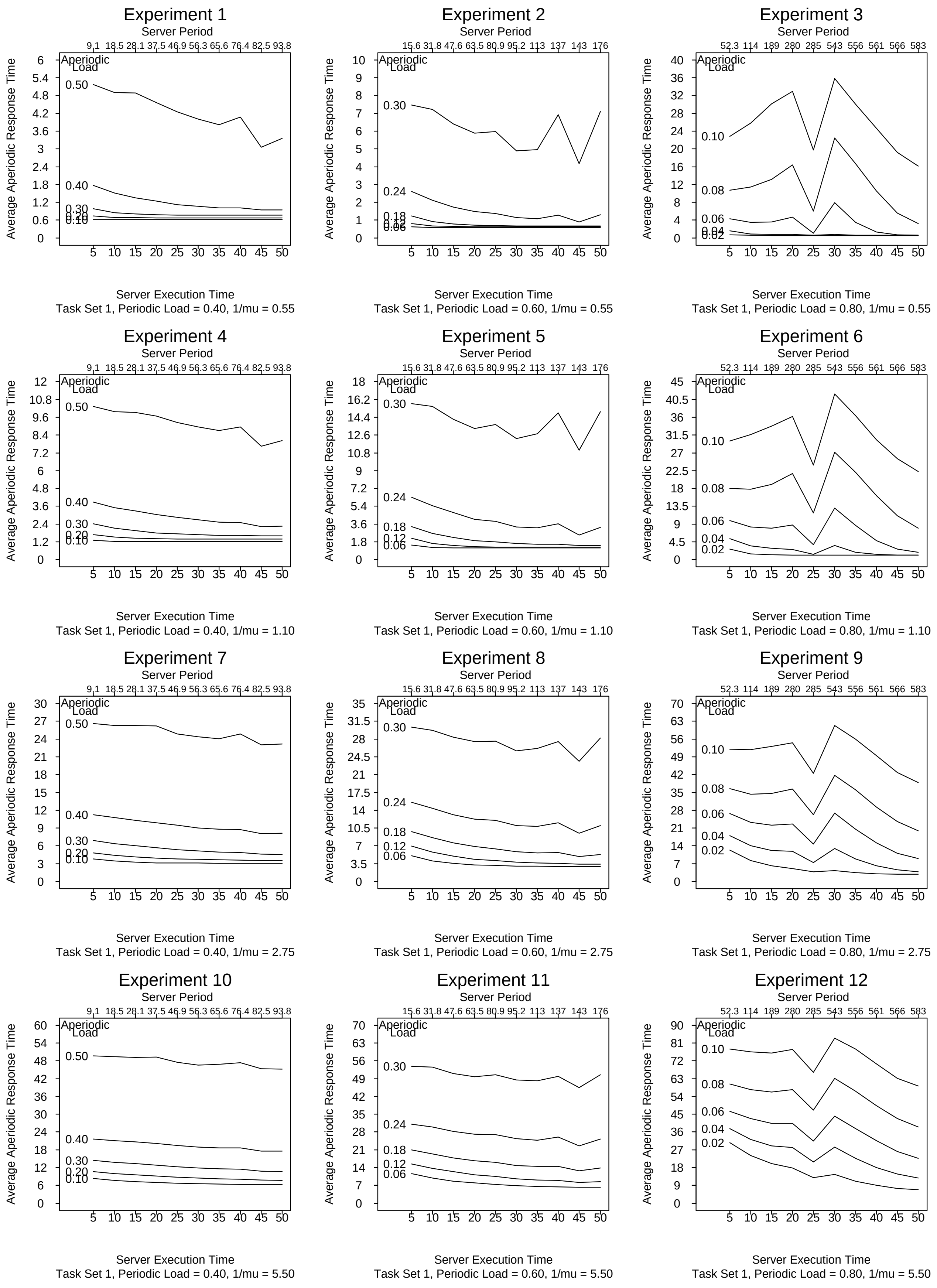


Figure 3-23: Average Aperiodic Response Time as a Function of SS Execution Time for Task Set 1 and Mean Aperiodic Service Times of 0.55, 1.10, 2.75, and 5.50 Units

3. For a given server period, the higher the server priority, the earlier the sporadic server can provide service to aperiodic tasks.

Using these guidelines, the essential data necessary to design a sporadic server can be obtained from a graph of the maximum server size and execution time as a function of server period and priority (examples of these graphs are presented in Figures 3-20 and 3-21). To satisfy the first two guidelines above, the longest server period for which the maximum server size is close to its highest value should be chosen. For example, using the data from Figure 3-21 the sporadic server periods for the 40%, 60%, and 80% periodic loads should be approximately 110, 130, and 570 respectively. Once the server period is chosen, the server execution time can be determined from the graph of maximum server execution time.

To provide the best average aperiodic response time, the sporadic server should usually be given highest priority. This is certainly the proper choice when no server size advantage exists in giving the sporadic server a slightly lower priority as is the case for the 80% periodic load in Figure 3-21. However, for cases where a server size advantage exists for a lower priority server with the same server period (as is the case for server periods greater than 110 for the 40% periodic load case in Figure 3-21), one must also consider two other factors: the mean aperiodic service time and the aperiodic load. If the mean aperiodic service time is a large percentage of the high priority sporadic server's execution time and the aperiodic load is high enough to cause the sporadic server to be overrun frequently,* then the lower priority server with a larger execution time may provide a better average response time because more aperiodic service can be delivered during the server's period. However, this should be a rare case because the server's execution time has been chosen to be large and a correspondingly large aperiodic task would be necessary for this problem to arise. Also, the response time advantage of a larger server size becomes less as the mean aperiodic service time increases.

*A method which can indicate the frequency of server overruns is presented in Section 4.2.1

4. Prediction of Aperiodic Response Time Performance

In this chapter we present models for the background, DS, and SS aperiodic service algorithms that were created to provide good preditions of average aperiodic response time. Although none of the models presented here accurately predict average aperiodic response time for all conditions, they are applicable for a non-trivial set of conditions and they do identify the dominant parameters that determine average response time. This chapter is organized in two major sections. The first section covers the models for predicting average response time using background service and the second section covers the models for predicting average response time using the DS and SS algorithms.

4.1 Modelling Background Aperiodic Service

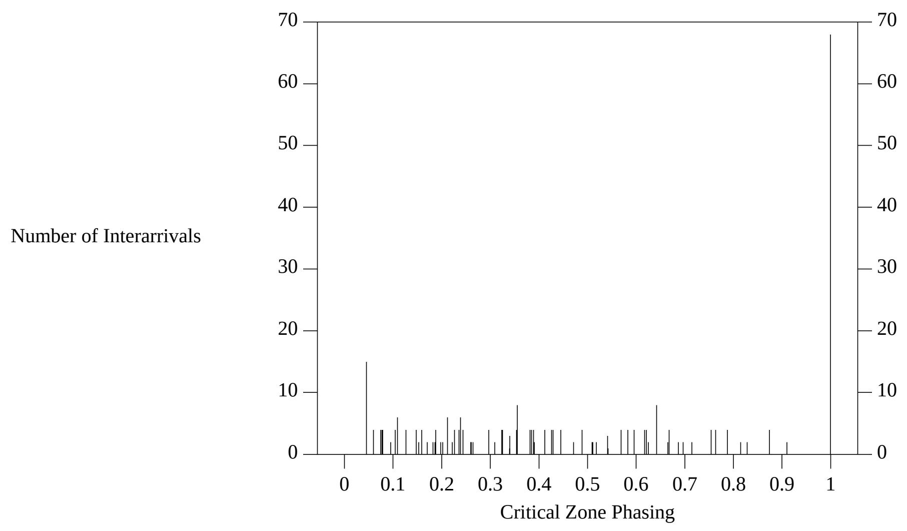
As was apparent from the experiments described in Chapter 3, background aperiodic service provides a poor average aperiodic response time. The reason for the poor performance is that background service opportunities are infrequent and independent of the arrival of aperiodic tasks. Our interest in modelling background performance is to provide an accurate bound for aperiodic response time. This section describes our attempts to predict background aperiodic response time and discusses the reasons for the success or failure of the various approaches.

4.1.1 Modelling the Periodic Tasks as a Poisson Process

Since we are attempting to model a priority queueing system, we investigated the possibility of using the queueing theory results for priority queues to predict the average response time for the aperiodic tasks. Closed-form queueing theory equations exist for predicting the average response time for a priority-ordered set of jobs whose arrival process is Poisson ([Kleinrock 76], Chapter 3). Therefore, if we could model the collection of periodic tasks as one, high-priority job stream with a Poisson arrival pattern, then we could simply apply the appropriate queueing theory equation to predict the average response time of the lower-priority aperiodic tasks.

To determine the validity of this approach, we obtained the distribution of the interarrival times for an ensemble of periodic tasks. A set of 20 periodic tasks sets with 10 periodic tasks each were chosen at random. It was only necessary to obtain the periodic task interarrival times for one hyperperiod (the least common multiple of the periodic task periods) because the sequence of periodic arrivals repeats itself every hyperperiod. The aggregate mean interarrival time for the periodic tasks was computed as:$\lambda = \sum \frac{1}{T_i}$, where T_i is the period of periodic task i. For each observed interarrival time, t, the quantity $e^{-\lambda t}$ was computed and a histogram of these values was created. A critical-zone phasing and a random phasing for each task set was examined. Figure 4-1 shows the histograms for both the critical zone and random phasing of one of the periodic task sets. This type of higtogram for a true Poisson process would show a uniform distribution.

The critical zone phasing histogram in Figure 4-1 shows the periodic task interarrival times to be fairly well dispersed. However, the histogram does *not* indicate a uniform distribution. The large spike at 1.0 represents the number of interarrival times whose value was zero. These zero interarrival times are due to the zero phasing of the periodic tasks and the harmonic periods of some of the periodic tasks. The smaller



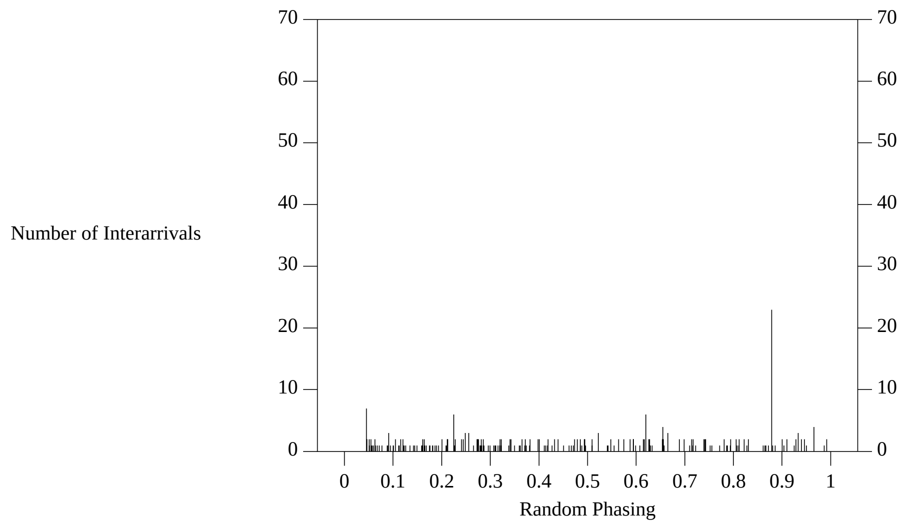


Figure 4-1: Distribution of Periodic Interarrival Times

spike at 0.5 is also due to the harmonic periodic task periods. This histogram indicates that the periodic task arrival process is most probably not Poisson in character.

The random phasing histogram in Figure 4-1 has eliminated the large spike at 1.0 and more evenly distributed the interarrival times. However, a large spike is still present. This "better" histogram suggested that the periodic task arrival process might be representable as a Poisson arrival process.

Experiments were conducted to obtain data for the average aperiodic response time of background service for comparison to the priority queue prediction. For these experiments, the setup described in Section 3.2 was used for Task Set 0 (see Appendix I). The results for these experiments is presented in Figure 4-2. The priority queueing equation has only a very limited success predicting the observed average response time for background service. For low periodic and aperiodic loads, the predictions were fairly good and improve as the mean aperiodic service time increases. However, as can be seen from Figure 4-2, the predictions become very pessimistic as either the periodic or aperiodic load is increased. The pessimistic prediction indicates that a high-priority Poisson process can be much more invasive than a set of high priority periodic tasks. We concluded that the attempt to use a Poisson process to model the deterministic arrival process of the periodic tasks was not a good approach.

4.1.2 A Deterministic Approximation for the Aperiodic Tasks

In contrast to the approach tasken in the previous section where we attempted to model the deterministic nature of the periodic tasks with a random process (i.e., a Poisson process), we now present a deterministic model for the random nature of the aperiodic tasks. We attempt to model the aperiodic tasks, which have an exponential service time distribution and a Poisson arrival process, with a stream of tasks that have deterministic (constant) service times and interarrival times. A deterministic service time and interarrival time distribution for the aperiodic tasks along with the corresponding information for the periodic tasks allows the development of an exact equation for the average aperiodic response time.

Simulations were conducted to test the validity of this approach and we found that, except for cases of low aperiodic load and small aperiodic service time, the response time for the deterministic aperiodic stream would be a much too optimistic prediction. Figure 4-3 shows the average response time for both types of aperiodic tasks for Task Set 0 (see Appendix I). From Figure 4-3, it is obvious that this approach holds little promise for predicting average aperiodic response time for aperiodic tasks with exponentially distributed service times and a Poisson arrival process.

4.1.3 Modelling the Aperiodic Queue the Case of One Periodic Task

In the last two sections, we found that modelling a deterministic process (the periodic tasks) as a random process and modelling a random process (the aperiodic tasks) as a deterministic process were not good approaches. Clearly, an average aperiodic response time model that is based upon assumptions that are gross simplifications of the actual processes involved is not going to be successful. Therefore, an approach that models both the random behavior of the aperiodic tasks and the deterministic behavior of the periodic tasks is necessary. In this section we present a model for the effect of one periodic task upon the length of aperiodic queue. This model allows us to compute the average aperiodic queue length. Using Little's formula [Little 61], we are then able to compute the average aperiodic response time.

The model of the aperiodic queue length for background service with one, high-priority periodic task is presented in Figure 4-4. At the beginning of the periodic task execution, the aperiodic queue has an initial length of $\mathbf{L_0}$ jobs. During the periodic task's execution, $\mathbf{C_p}$, no aperiodic tasks are serviced and the aperiodic queue length grows as new aperiodic tasks arrive reaching a length of **N** at the end of the periodic task execution. Once the periodic task completes its execution, the accumulated jobs in the aperiodic queue can be worked off.

We are interested in computing the average aperiodic response time, **W**. This is accomplished by computing the average aperiodic queue length, **L**, and applying Little's formula, $\mathbf{L} = \lambda\mathbf{W}$. The average aperiodic queue length is computed by determining the area under the curve of the aperiodic queue from time = 0 to time = $\mathbf{T_p}$ and dividing by $\mathbf{T_p}$. The area under the aperiodic queue curve is separated into three sections. $\mathbf{A_L}$ is the "left area" representing the area under the aperiodic queue during the execution

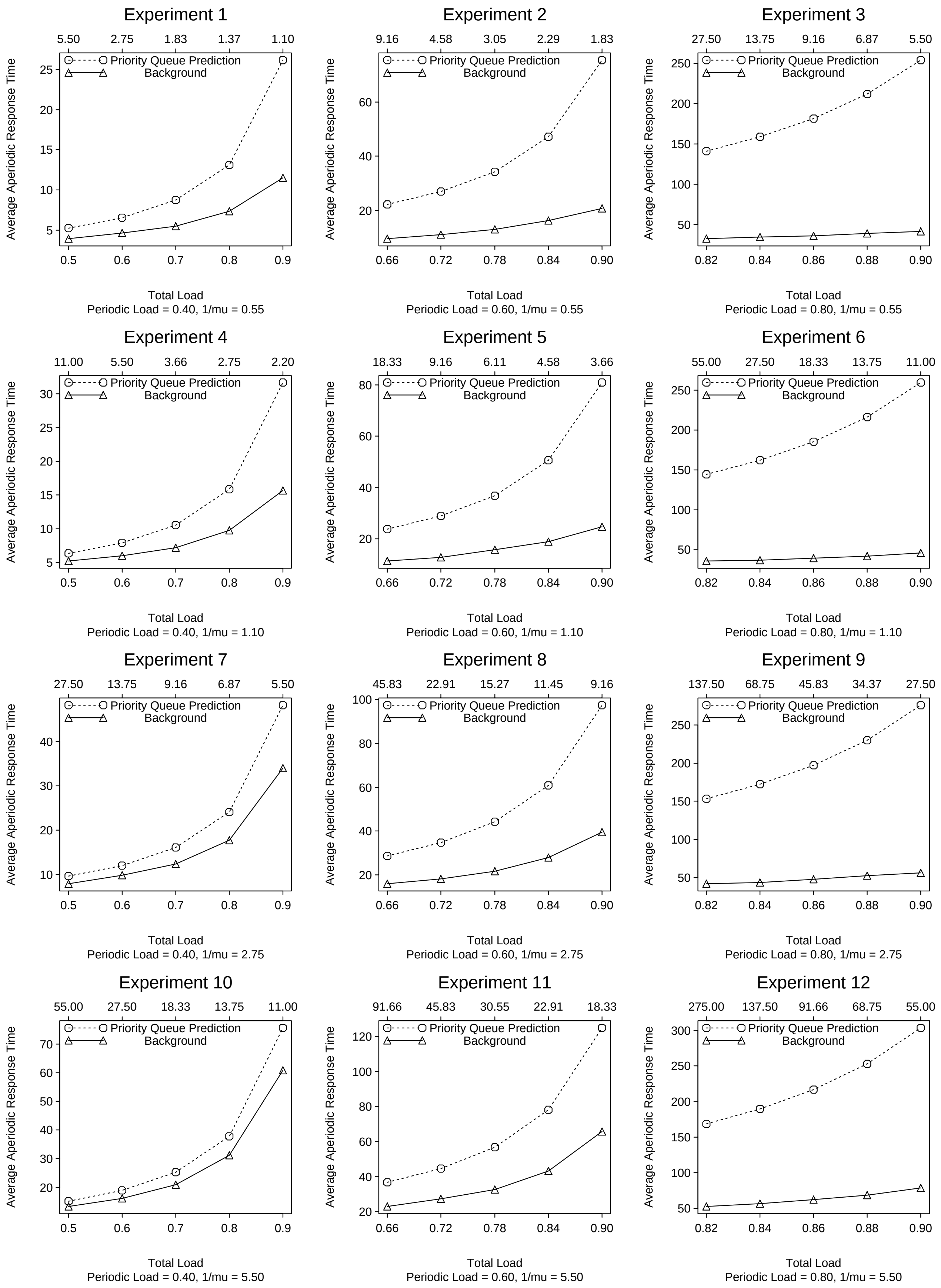


Figure 4-2: Comparison of Observed Average Aperiodic Response Time with Priority Queue Prediction for Background Aperiodic Service

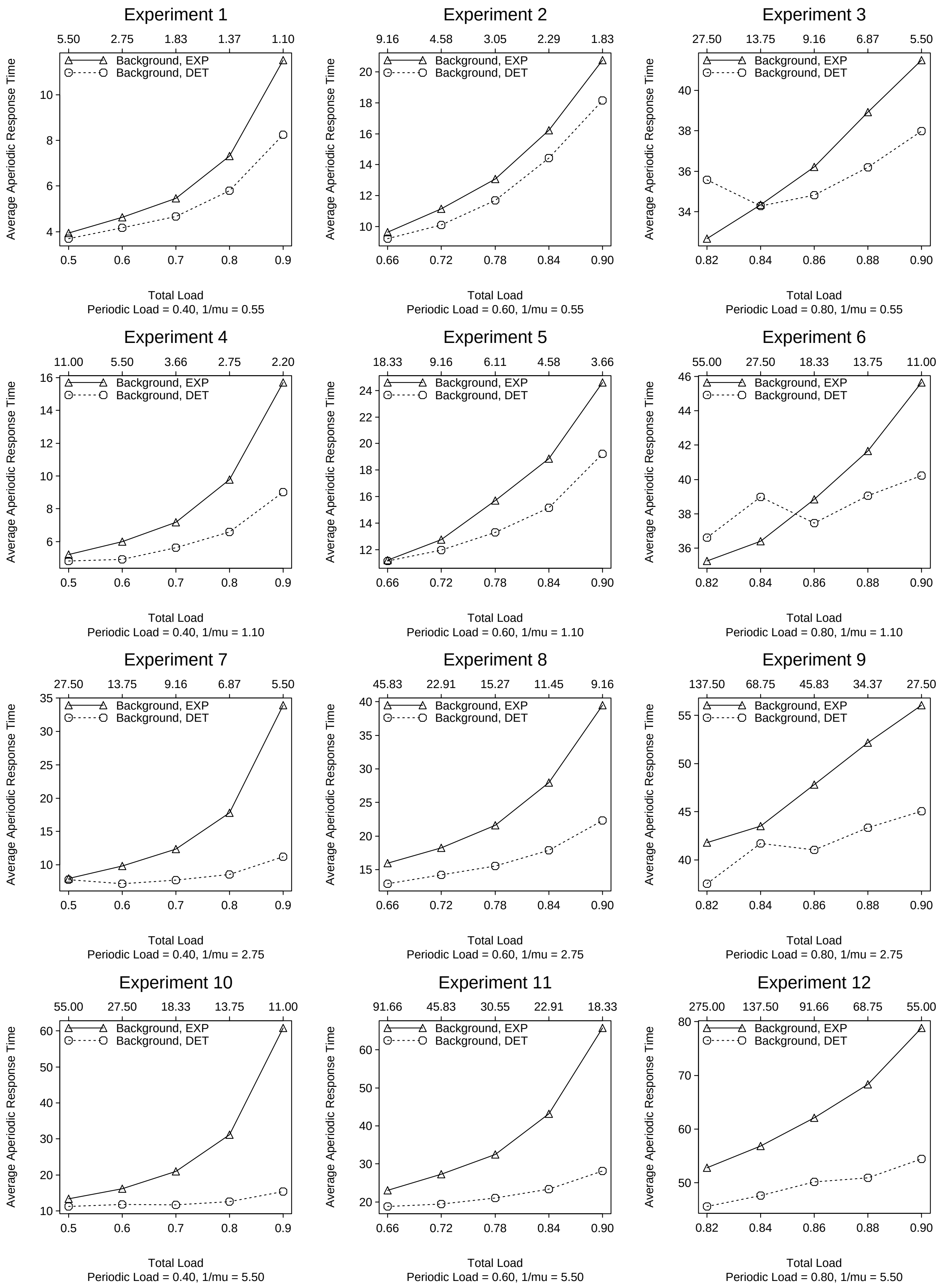
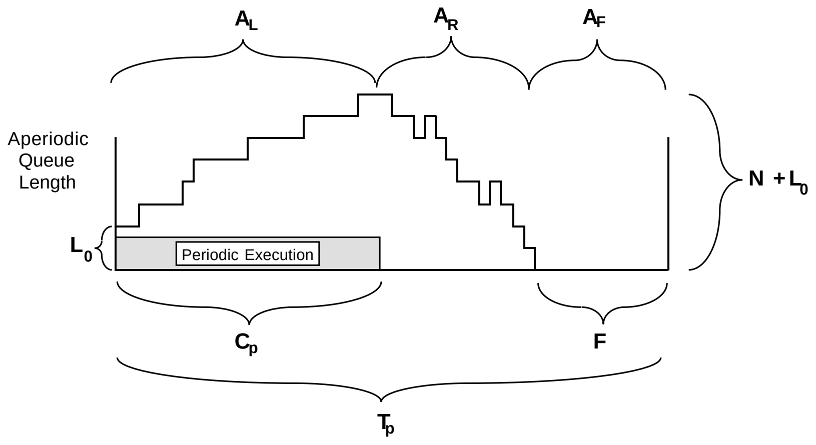


Figure 4-3: Comparison of Background Aperiodic Response Time for Aperiodic Tasks with Deterministic and Exponential Service Time and Interarrival Time Distributions



$C_p \rightarrow$ Duration of periodic blocking.

$T_p \rightarrow$ Period of periodic task.

$L_0 \rightarrow$ Length of aperiodic queue at start of periodic execution.

$N \rightarrow$ Number of aperiodic arrivals during periodic execution.

$A_L \rightarrow$ Area under aperiodic queue curve during buildup.

$A_R \rightarrow$ Area under aperiodic queue curve during workoff.

$F \rightarrow$ Interval of time after the aperiodic queue empties and before the next periodic task arrival.

$A_F \rightarrow$ Area under aperiodic queue during **F**.

Figure 4-4: Model of Aperiodic Queue and One Periodic Task

of the periodic task. $\mathbf{A_R}$ is the "right area" representing the area under the aperiodic queue for the interval after the periodic task has completed up to the first point at which the aperiodic queue becomes empty. The interval **F** represents the time between the first point at which the aperiodic queue length is zero after the end of the periodic task execution and the beginning of the next the next periodic task execution. $\mathbf{A_F}$ represents the area under the aperiodic queue for the interval **F**.

To create equations for $\mathbf{A_L}$ and $\mathbf{A_R}$, we make the assumption that the interval between the end of the periodic task execution and the beginning of the next periodic task execution is long enough for the aperiodic queue to empty and return to an M/M/1 equilibrium. Using λ as the mean arrival rate for the aperiodic tasks, μ as the mean service rate for the aperiodic tasks, and $\rho = \lambda/\mu$ as the aperiodic load, we have the following for $\mathbf{L_0}$ (from [Kleinrock 75]):

$$E[L_0] = \overline{L_0} = \frac{\rho}{1 - \rho} \tag{7}$$

$$Var[L_0] = \sigma^2_{L_o} = \frac{\rho}{(1 - \rho)^2} \tag{8}$$

The derivation of $\mathbf{A_L}$ proceeds as follows. Let N be the number of arrivals during the interval $[0,\mathbf{C_p}]$. Because the arrival process is Poisson process with parameter λ, the following are true for N:

$$E[N] = \lambda \mathbf{C_p} \tag{9}$$

$$Var[N] = \lambda \mathbf{C_p} \tag{10}$$

Let the N arrivals be represented by $\tau_1, \ldots \tau_N$. Because the arrival process is Poisson, we know that the arrivals, τ_i, are uniformly distributed over the interval $[0,\mathbf{C_p}]$. We can now write the following expression for $\mathbf{A_L}$ conditioning on the value of N and $\mathbf{L_0}$:

$$A_L|N,L_0 = L_0C_P + \sum_{i=1}^{N} (C_P - \tau_i) \tag{11}$$

Because the arrivals over the inverval $[0,\mathbf{C_p}]$ are uniformly distributed, the expected value of the $(\mathbf{C_p} - \tau_i)$ term is $\mathbf{C_p}/2$ and we have the following:

$$\begin{aligned} E\left[A_L|N,L_0\right] &= L_0C_P + \sum_{i=1}^{N} \frac{C_P}{2} \\ &= L_0C_P + \frac{NC_P}{2} \end{aligned} \tag{12}$$

By replacing N with E[N] in Equation 12 we obtain the desired equation for $\mathbf{A_L}$:

$$\overline{A_L} = E\left[A_L\right] = \frac{\rho C_P}{(1 - \rho)} + \frac{\lambda C_P^2}{2} \tag{13}$$

The derivation of $\mathbf{A_R}$ proceeds as follows. The model we use to develop the equations for $\mathbf{A_R}$ is presented in Figure 4-5. The variables displayed in Figure 4-5 have the following meanings:

- X represents the initial height of the aperiodic queue at the end of the periodic task execution.
- τ_{X-1} represents the time it takes for the length of the aperiodic queue drop from X to X-1. τ_{X-2} represents the time it takes for the aperiodic queue drop from X-1 to X-2. τ_i represents the time it takes for the aperiodic queue drop from i+1 to i.
- A_X represents the area under the aperiodic queue throughout the duration of τ_{X-1} exclusive of the shaded area during this interval.

Using the model presented in Figure 4-5, the following equation computes $\mathbf{A_R}$:

$$A_R = (X - 1)\tau_{X-1} + (X - 2)\tau_{X-2} + \cdots + 2\tau_2 + \tau_1 + \sum_{i=1}^{X} A_i \tag{14}$$

Due to the Poisson arrival process and the exponentially distributed service times for the aperiodic tasks, the values of τ_i are independent and identically distributed. For the same reasons, the values of A_i are also independent and identically distributed. Therefore, we may use Equation 14 to compute the expected value of $\mathbf{A_R}$ conditioning on the value of X as follows:

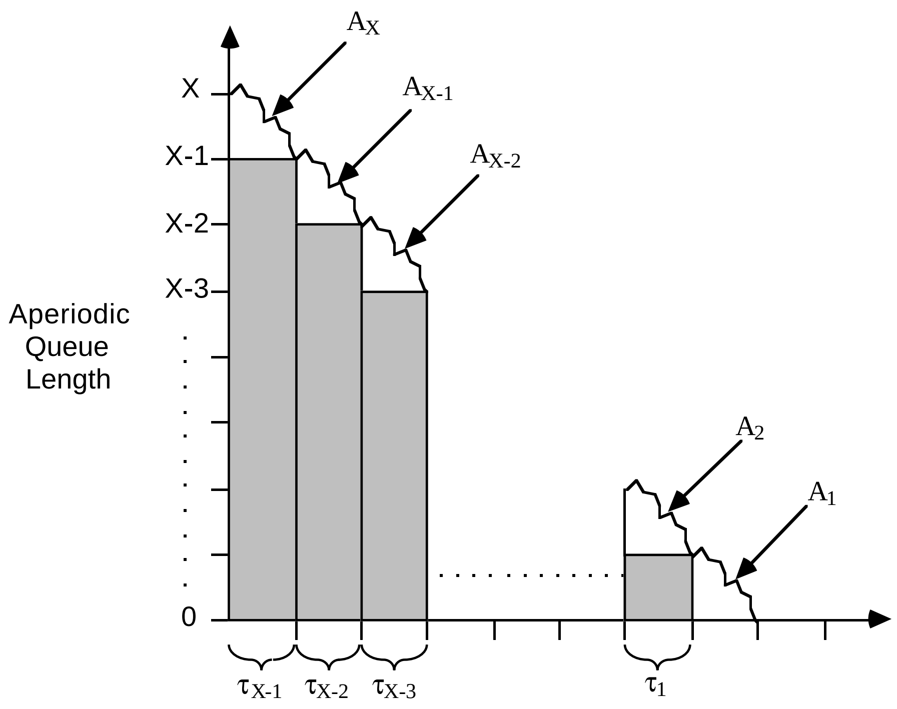


Figure 4-5: Model of Aperiodic Queue for Computing $\mathbf{A_R}$

$$E[A_R|X] = E[\tau_1]\sum_{i=1}^{X-1} i + XE[A_1]$$

$$= E[\tau_1]\frac{X(X-1)}{2} + XE[A_1] \tag{15}$$

Removing the condition from Equation 15 we have:

$$E[A_R] = \frac{E[\tau_1]E[X^2 - X]}{2} + E[X]E[A_1] \tag{16}$$

Next we need an equation for $E[\tau_1]$. $E[\tau_1]$ is the expected time to move from an aperiodic queue length of 1 to an aperiodic queue length of zero. Since we are using a Poisson arrival process with exponentially distributed service times, this corresponds to the mean busy period duration for an M/M/1 queue. Therefore, from [Kleinrock 75] we have:

$$E[\tau_1] = \frac{1}{\mu - \lambda} \tag{17}$$

Since $E[\tau_1]$ is equivalent to the expected duration of the M/M/1 busy period, $E[A_1]$ is the expected area under the curve of queue length during an M/M/1 busy period. We can compute $E[A_1]$ by multiplying the expected length of the aperiodic queue during the busy period with the expected duration of the busy period. Letting L represent the queue length for an M/M/1 queue, we have from [Kleinrock 75]:

$$E[L] = \frac{\rho}{1 - \rho} \tag{18}$$

Letting L_b represent the queue length during the busy period and remembering that the probability that the server is busy in an M/M/1 system is ρ, we can also express E[L] as:

$$E[L] = \rho E[L_b] + (1 - \rho)(0) \tag{19}$$

Solving Equation 19 for $E[L_b]$ we have:

$$E[L_b] = \frac{1}{1 - \rho} = \frac{\mu}{\mu - \lambda} \tag{20}$$

Using Equations 17 and 20 we can compute $E[A_1]$:

$$\begin{aligned} E[A_1] &= E[\tau_1]E[L_b] \\ &= \left(\frac{1}{\mu - \lambda}\right)\left(\frac{\mu}{\mu - \lambda}\right) = \frac{\mu}{(\mu - \lambda)^2} \end{aligned} \tag{21}$$

Referring to Figure 4-4 we note that X corresponds to the initial height of the aperiodic queue at the beginning of periodic task execution plus the number of aperiodic tasks that arrive during periodic task execution. As such we have the following equations:

$$X = N + L_0 \tag{22}$$

$$\begin{aligned} E[X] &= E[L_0] + E[N] \\ &= \frac{\rho}{1 - \rho} + \lambda C_P \end{aligned} \tag{23}$$

Using Equations 7, 8, 9, and 10 we have the following:

$$\begin{aligned} E\left[X^2 - X\right] &= E\left[N^2 + 2NL_0 + L_0^2 - N - L_0\right] \\ &= \overline{N}^2 + \sigma_N^2 + 2\overline{N}\,\overline{L_0} + \overline{L_0}^2 + \sigma_{L_0}^2 - \overline{N} - \overline{L_0} \end{aligned} \tag{24}$$

Substituting Equations 17, 21, 23, and 24 into Equation 16 we obtain the desired equation for $\mathbf{A_R}$:

$$\overline{A_R} = E[A_R] = \frac{\overline{N}^2 + \sigma_N^2 + 2\overline{N}\,\overline{L_0} + \overline{L_0}^2 + \sigma_{L_0}^2 - \overline{N} - \overline{L_0}}{2(\mu - \lambda)} + \frac{\mu(\overline{L_0} - \overline{N})}{(\mu - \lambda)^2} \tag{25}$$

We can compute the area $\mathbf{A_F}$, the average queue length **L**, and the average response time **W** (using Little's formula) with the following:

$$\overline{A_F} = \frac{\rho}{1-\rho} F$$

(26)

$$\overline{L} = \frac{\overline{A_L} + \overline{A_R} + \overline{A_F}}{T_p}$$

(27)

$$W = \frac{\overline{L}}{\lambda}$$

(28)

These equations work well for the cases where the above assumptions are met. Figure 4-6 presents a comparison of the observed background aperiodic response time and the predictions obtained with the above equations. The periodic task for these experiments has a period of $\mathbf{T_p}$ = 100 and $\mathbf{C_p}$ = 50. Six different mean aperiodic service times are presented for aperiodic loads ranging from 5% to 40%. In all cases, the graphs show that the predictions work well for low aperiodic load. As the aperiodic load increases, the observed average response time for background service becomes worse than the equations predict. This behavior explained by noting that as the aperiodic load increases, the higher the probability that the aperiodic queue will not empty before the next execution of the periodic task.

One can predict the ranges of aperiodic load and mean aperiodic service time for which the above assumptions are valid by considering the length of the first busy period of an M/M/1 queue with an initial number of jobs in the queue. Using $\mathbf{E[N] + E[L_0]}$ as the initial condition for an M/M/1 queue, we want to obtain an estimate of the "maximum" length of the first busy period. If we assume the first busy period duration to be normally distributed, then the mean first busy period duration plus two standard deviations should be a reasonable estimate for, $\mathbf{g_{max}}$, the "maximum" first busy period duration. This can be computed as follows using the equations for the mean and variance of the busy period from [Kleinrock 75]:

$$g_{max} = \frac{1/\mu}{1-\rho} + \frac{2}{\mu}\sqrt{\frac{1+\rho}{(1-\rho)^3}}$$

(29)

As long as $\mathbf{g_{max}}$ is less than the interval between the end of one periodic task execution and the start of the next periodic task execution, the assumptions above should hold and the equations above should provide a good prediction.

This estimated maximum first busy period data for the experiments of Figure 4-6 are presented in Figure 4-7. The point at which the maximum first busy period curve rises above 50 should correspond to the aperiodic load at which the observed background response time becomes larger than the predicted background response time. Figure 4-8 presents an expanded view of the data presented in Figure 4-6 which focusses upon the departure of the observed background response time from the predicted

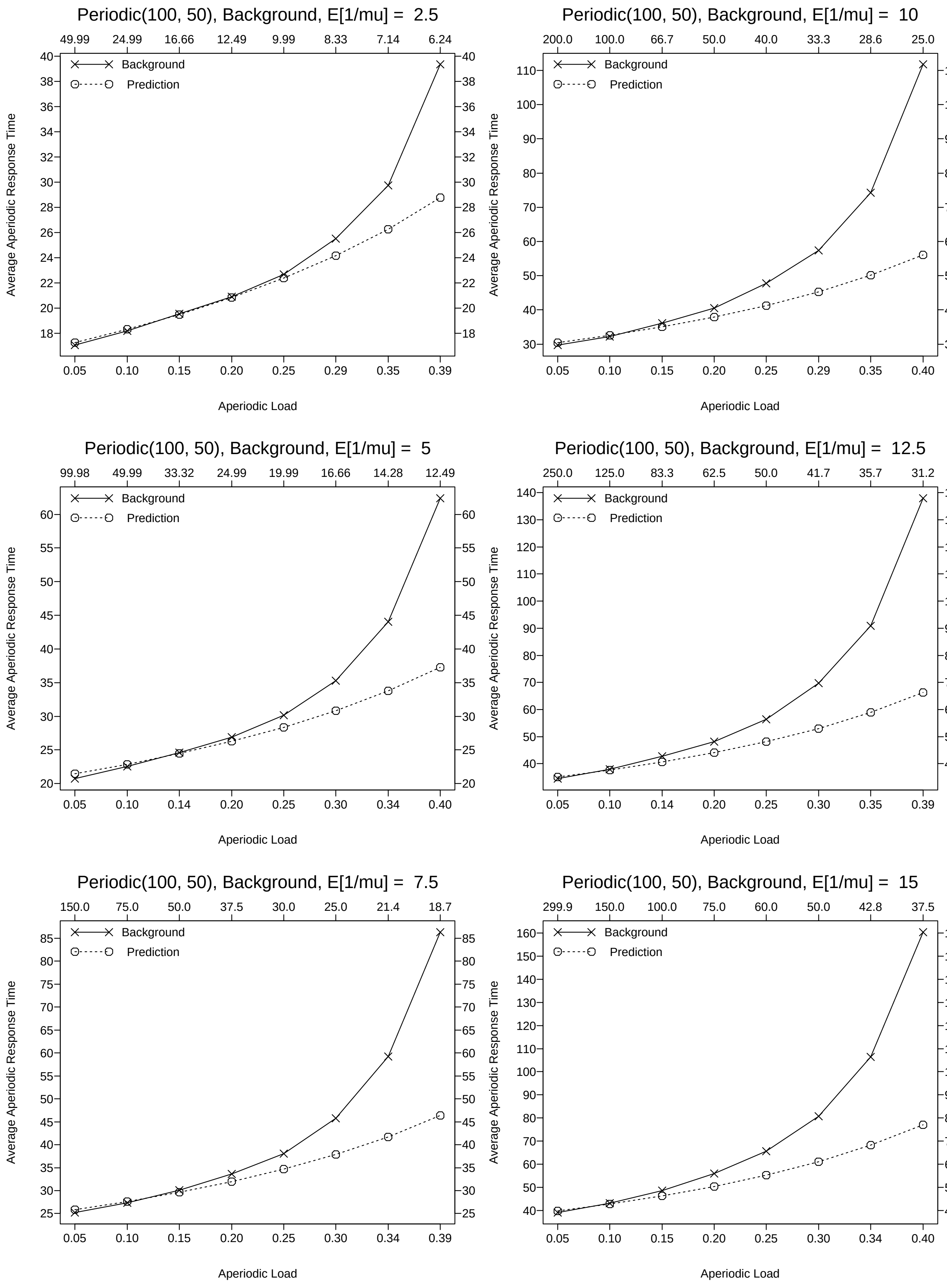


Figure 4-6: Comparison of Observed Average Background Response Time with Predicted Background Response Time

background response time. In Figure 4-8, the observed background response time is plotted relative to the predicted background response time (i.e., a value of 1.0 on this graph indicates a match of the observed to the predicted response time and a value greater indicates an observed response time larger than the predicted resposne time). A horizontal dashed line has been drawn which indicates the points at which the observed background response time has risen 6% higher than the predicted response time. Note that the aperiodic loads at which the observed response time curves cross the 6% line correspond well with the aperiodic loads at which the corresponding maximum first busy period curve in Figure 4-7 rises above 50.

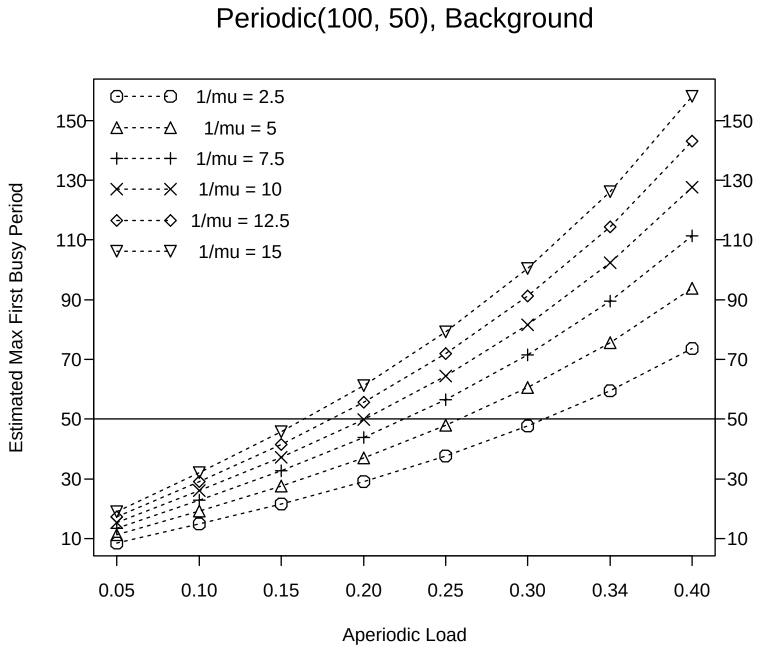


Figure 4-7: Estimated Maximum First Busy Period

Thus, we have a good predictor of background performance using equations 13, 25, 26, 27, and 28 whenever their assumptions hold. We also can calculate a good estimate for when the above assumptions will hold. The next section describes our attempts to apply this approach to the more complex problem of many periodic tasks.

4.1.4 Modelling the Aperiodic Queue for the Case of Many Periodic Tasks

In this section we describe a model for the aperiodic queue length for background aperiodic service with many periodic tasks. This approach is similar to the approach described in the previous section in that it describes the aperiodic queue during intervals of periodic task execution and "idle" intervals during which no periodic tasks execute. However, for one periodic task, the periodic task execution and idle intervals are constant. This is not the case for many periodic tasks. For the many periodic task case, many different durations are possible for periodic task execution and idle intervals.

Figure 4-9 shows the periodic task execution and idle intervals for a small periodic task set. The hyperperiod for this task set is 210. This example demonstrates how a simple task set can produce many different combinations of periodic task execution and idle intervals. The approach described in this

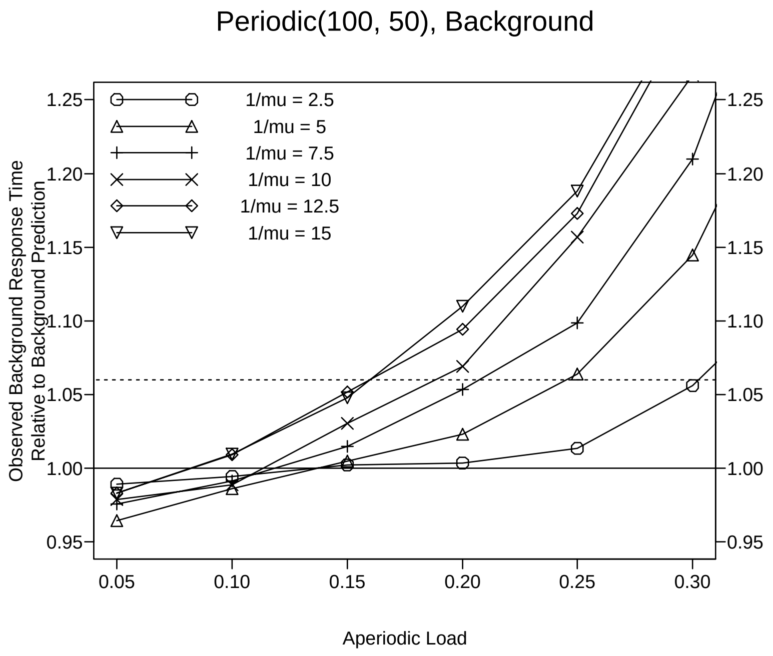
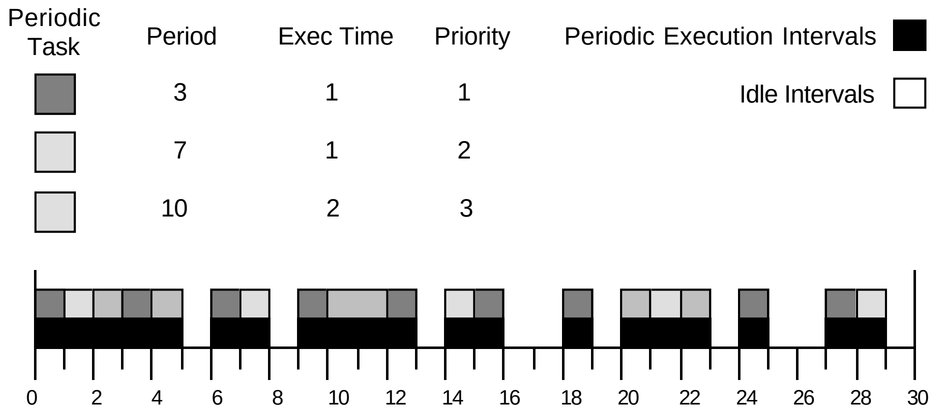


Figure 4-8: Observed Average Background Response Time Relative to Predicted Background Response Time

section requires the determination of the durations of each continuous interval of periodic task execution and each idle interval in order to model the aperiodic queue length.



Periodic Task	Period	Exec Time	Priority
▨	3	1	1
▢	7	1	2
▢	10	2	3

Figure 4-9: Example of Periodic Task Execution and Idle Intervals

Unfortunately, for the many periodic task case, the assumptions necessary to apply the equations described in the last section rarely hold. With many periodic tasks, the idle intervals are often short and many are preceded by long periodic task execution intervals. This leads to the problem of "spill-over" which describes the cases where the idle interval following a periodic task execution interval is not long enough to empty the aperiodic queue before the next periodic task execution interval begins. In these cases, the aperiodic work "spills over" into the next periodic task execution interval. This exacerbates the problem of predicting the average aperiodic queue length because a spill-over makes the value for $\mathbf{L_0}$ at the beginning of the next periodic execution dependent upon the aperiodic work uncompleted during the previous periodic execution interval. Also, note that once a spill-over occurs it is more likely that a spill-over will occur in the following idle interval. This can cause a "dominoe" effect in which the length of the aperiodic queue at the beginning of successive periodic task execution intervals is dependent upon the all the periodic execution and idle intervals since the last time the aperiodic queue was emptied.

Since the assumptions of the equations from the last section do not hold in general for the case of many periodic tasks, we tested the following simplification. Equation 13 for $\mathbf{A_L}$ is valid as long as we have a good value for $\mathbf{L_0}$. However, Equation 25 for $\mathbf{A_R}$ is not valid because it assumes no spill-over. Therefore, for the cases where spill-over is likely, we assumed a linear decrease in aperiodic queue length (with slope = $\mu - \lambda$) during an idle interval. We then use the area of the trapezoid depicted in Figure 4-10 as an approximation for $\mathbf{A_R}$. We also use the assumption of a linear decrease in aperiodic queue length to predict the value of $\mathbf{L_0}$ for the calculation of $\mathbf{A_L}$. By applying this approximation to every periodic execution and idle interval throughout the entire hyperperiod, we can obtain an approximation for the average queue length and the average aperiodic response time.

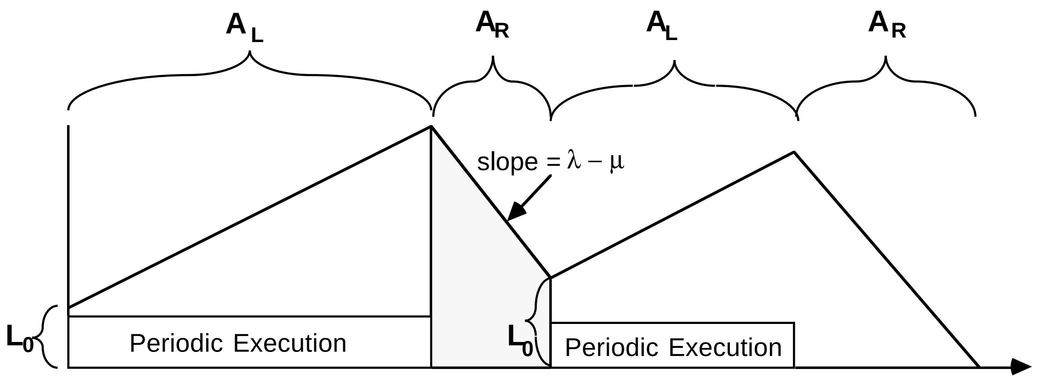


Figure 4-10: Spill-Over Approximation

This approximation method was tested with the same task set used in Figures 4-2 and 4-3 and the results are presented in Figure 4-11. As can be seen from the graphs in Figure 4-11, this prediction method is too optimistic. The only cases where this prediction is reasonably close are those with low aperiodic load. Clearly, the linear approximation for E[Q(t)] (the expected aperiodic queue length as a function of t) and/or the trapezoid area approximation for $\mathbf{A_R}$ are to blame for these poor predictions.

To determine the source of the error with the above approach, we collected data for the values of $\mathbf{L_0}$ and **N** (see Figure 4-4) for every periodic execution interval throughout the hyperperiod of a periodic task set with a total periodic load of 70% and an aperiodic load of 15%.[*] From this experiment we learned the following:

[*]The simulation to collect this data was was executed for 2000 hyperperiods. Each hyperperiod had over 100 periodic execution intervals. This one simulation produced over 16 Megabytes of data. The long computation time and large storage requirements prohibited us from doing a more complete set of experiments comparable to the experiments reported in Figure 4-11.

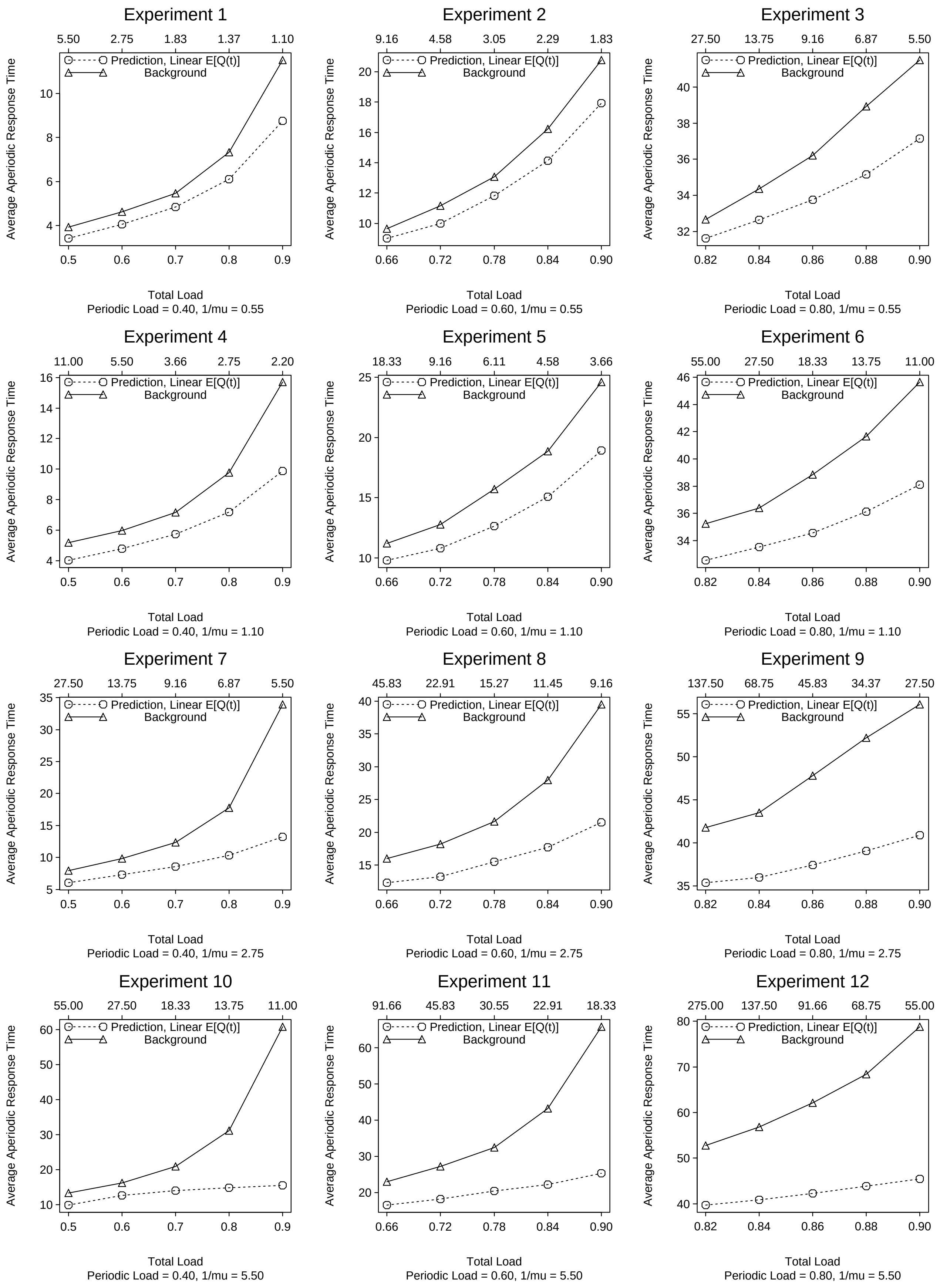


Figure 4-11: Comparison of Observed Average Aperiodic Response Time with Prediction for Background Aperiodic Service based upon Linear E[Q(t)]

- The linear prediction for E[Q(t)] given the initial value for Q(0) (**N**) is only good for values of Q(0) >> E[Q(∞)], the equilibrium M/M/1 queue length or small values of t. For smaller values of Q(0) or larger values of t, the linear prediction is lower than the observed value of E[Q(t)]. This confirms a result found earlier by Odoni and Roth in [Odoni 83].
- Using the observed values for $\mathbf{L_0}$, Equation 13 predicts very well the observed values for $\mathbf{A_L}$. The sum of the predicted values of $\mathbf{A_L}$ for each periodic execution interval is greater than the observed sum by only 0.1%. The sum of the values of $\mathbf{A_L}$ for each periodic execution interval account for 80% of the total average area under the curve of the aperiodic queue.
- Using the the observed values for **N** and $\mathbf{L_0}$, Equation 25 predicts fairly well the observed values for $\mathbf{A_R}$. The sum of the predicted values of $\mathbf{A_R}$ for each periodic execution interval is greater than the observed sum by 5%. The sum of the values of $\mathbf{A_R}$ for each periodic execution interval account for 20% of the total average area under the curve of the aperiodic queue.
- Equation 13 with the observed values of $\mathbf{L_0}$ and the trapezoid approximation for $\mathbf{A_R}$ with the observed values for **N** and $\mathbf{L_0}$ yields an average aperiodic response time prediction that is only 0.3% greater than the observed average aperiodic response time. Since the $\mathbf{A_R}$ areas account for only 20% of the area under the curve of the aperiodic queue, the trapezoid approximation error for $\mathbf{A_R}$ does not significantly affect the average aperiodic response time prediction.

Thus, the key to making this approach work is the accurate prediction of the average values for $\mathbf{L_0}$ and **N** for each periodic execution interval. Given a particular $\mathbf{L_0}$, the succeeding value of **N** is calculated as:

$$\mathbf{N = L_0 + \lambda C_p} \tag{30}$$

(where C_p is the duration of periodic execution interval). The difficult prediction is the the succeeding value for $\mathbf{L_0}$ given the value for **N**. This problem corresponds the the problem of transient analysis for the M/M/1 queue to provide E[Q(t)] given an initial condition of Q(0). This is a difficult problem and has received considerable attention [Odoni 83, Vanas 86, Gross 84, Kelton 85, Everitt 84, Pegden 82, Rider 76]. But as Gross and Miller point out in reference to the solution for the transient probabilities of a Markov process: "Nice analytical solutions rarely exist. Even for the M/M/1 queue, the solution is a rather formidable expression in terms of modified Bessel functions" [Gross 84]. Indeed, the solution involves an infinite series of modified Bessel functions. Numerical methods are usually used to solve these problems. However, we want a solution that does not rely upon numerical methods.

However, we found that even if we had a perfect prediction for E[Q(t)] given the observed average value for **N** as Q(0), we would not get an accurate prediction of the observed value for the succeeding $\mathbf{L_0}$. The reason for this discrepency is that the distribution of **N** is not being considered here, only its average value. We empirically obtained the distribution for **N** which indicates the probability of **N** being 0, 1, 2, etc., and found that if we use a linear approximation for E[Q(t)] for Q(0) values of 0, 1, 2, etc. and weight the predictions according to the distribution of **N** we obtain a good prediction for $\mathbf{L_0}$. Therefore, if we can obtain the distribution for **N** we can use the linear approximation for E[Q(t)] to predict the average value for the succeeding $\mathbf{L_0}$. Note that this requires that we also be able to predict the distribution for each $\mathbf{L_0}$ because this distribution and the duration of the periodic execution interval determines the distribution for the succeeding **N**.

We attempted to use the busy period distribution for an M/M/1 queue to predict the distribution of L. The following approximation method was used. Obtain the distribution of the first busy period for an M/M/1 queue given an initial condition of Q(0). Let T be the duration of the idle interval. Split the distribution of busy periods into two sets: (1) those that are greater than T and (2) those that are less than or equal to T. For the values in the first set, assume that the distribution of the aperiodic queue length at T is equivalent to the equilibrium M/M/1 distribution. For the values in the second set, plot a line from (0,Q(0)) to (length-of-busy-period, 0) and determine the height of the line at time T. Assume this value is the length of the aperiodic queue at time T. Combine the predictions for both sets to obtain a prediction for the distribution of $\mathbf{L_0}$ where E[Q(T)] = $\mathbf{L_0}$.

We found that this approach does not provide a good prediction of the distribution of $\mathbf{L_0}$. The predicted distributions are dissimilar to the observed predictions and the mean values of the observed and predicted distributions does not compare well. One reason for the poor predictions is that this approach could never predict a value for **L** that is greater than Q(0).

4.1.5 Summary of Background Aperiodic Service Modelling

We have identified a technique that provides a good approximation of average aperiodic response time for the case of one periodic task and background service. The technique predicts the average aperiodic response time by estimating the average aperiodic queue length and applying Little's formula $\mathbf{W} = \mathbf{L}/\lambda$. The curve of the aperiodic queue is broken down into two intervals: an interval of periodic task execution and an idle interval during which no periodic task executes. Under the assumption that the idle interval is long enough to allow the aperiodic queue to empty before the next interval of periodic task execution, Equations 13, 25, 26, 27, and 28 will provide a good prediction of average aperiodic response time. By estimating the maximum duration of the first busy period, $\mathbf{g_{max}}$, using Equation 29 one can determine the range of aperiodic load for which this technique applies.

We have also identified a technique that provides a good approximation of the average aperiodic response time for the case of many periodic tasks and background service. As in the case of one periodic task, this approach estimates the average aperiodic queue length by computing the areas under the aperiodic queue during periodic task execution and idle intervals. The approach relies upon Equations 13, 25, and 26 where applicable. However, for the cases where the idle interval is too short to use these equations, the area of the trapezoid depicted in Figure 4-10 is used for the value of $\mathbf{A_R}$. This approach works very well when the average aperiodic queue lengths at the beginning and the end of each periodic task execution interval can be predicted. However, we were unable to predict these values in cases of spill-over.

4.2 Modelling Aperiodic Service for the DS and SS Algorithms

As was shown in Chapter 3, the DS and SS algorithms provide substantial improvements in aperiodic response time performance compared to the background and polling service algorithms. In this section, we describe performance models for these algorithms that allow us, in some cases, to predict their average aperiodic response time performance. We begin by examining the conditions under which periodic task execution is not likely to cause any delay in the servicing of aperiodic tasks. Equations are then developed that provide a very good prediction of the range of aperiodic load for which the SS and DS algorithms can provide aperiodic service that is unaffected by periodic task execution.

4.2.1 Attaining Ideal Aperiodic Response Time Performance with the DS and SS Algorithms

In this section, we develop equations that estimate the aperiodic load at which the execution of periodic tasks begins to delay the servicing of aperiodic tasks. Below this aperiodic load, the DS and SS algorithms are able to provide an aperiodic response time performance very close to the response time performance obtainable by removing the periodic tasks from the system. As before, we model the interarrival times for the aperiodic tasks with a Poisson arrival process. However, for the aperiodic service times we consider both an exponential and a deterministic service time distribution. Thus, our goal is to compute the maximum aperiodic load at which either an M/M/1 or an M/D/1 queueing model accurately predicts the average aperiodic response time. In this section, we refer to the M/M/1 or M/D/1 average response time as the "ideal" response time.

The aperiodic response time performance of the DS and SS algorithms is accurately described with the following equation:

$$\mathbf{W} = \mathbf{W_F} + \mathbf{MD} \tag{31}$$

where:

W is the average aperiodic response time.

$\mathbf{W_F}$ is the average aperiodic response time for a system free of periodic tasks.

M is the fraction of aperiodic tasks delayed by periodic task execution.

D is the average delay duration of the delayed aperiodic tasks.

To test equation 31, the values for **M** and **D** were obtained via simulation for the DS and SS algorithms using task set 4 (see Appendix I) and Poisson arrivals and exponentially distributed service times for the aperiodic tasks. The data for **M** and **D** for the DS algorithm are presented in Figures 4-12 and 4-13. Figure 4-14 presents the observed average aperiodic response time and the response time prediction using the **M** and **D** data from Figures 4-12 and 4-13. As can be seen from Figure 4-14, Equation 31 accurately predicts the average response time performance of the DS algorithm. The corresponding data for the SS algorithm is presented in Figures 4-15, 4-16, and 4-17. As with the DS algorithm, Equation 31 accurately predicts the average aperiodic response time for the SS algorithm as can be seen in Figure 4-17.

To attain our stated goal, we want to be able to predict the range of aperiodic loads for which the contribution of the **MD** term of Equation 31 is very close to zero. In doing so, the average response time prediction approaches the standard queueing prediction for the average response time for an M/M/1 or an M/D/1 queue. From Figures 4-12 and 4-15 we can see that, whenever the percentage of aperiodic tasks delayed is close to zero, the response time performance of the DS or SS algorithm approximates the M/M/1 average response time. Therefore, we should be able to attain our goal if we can predict the range of aperiodic loads for which the percentage of aperiodic tasks delayed is very close to zero.

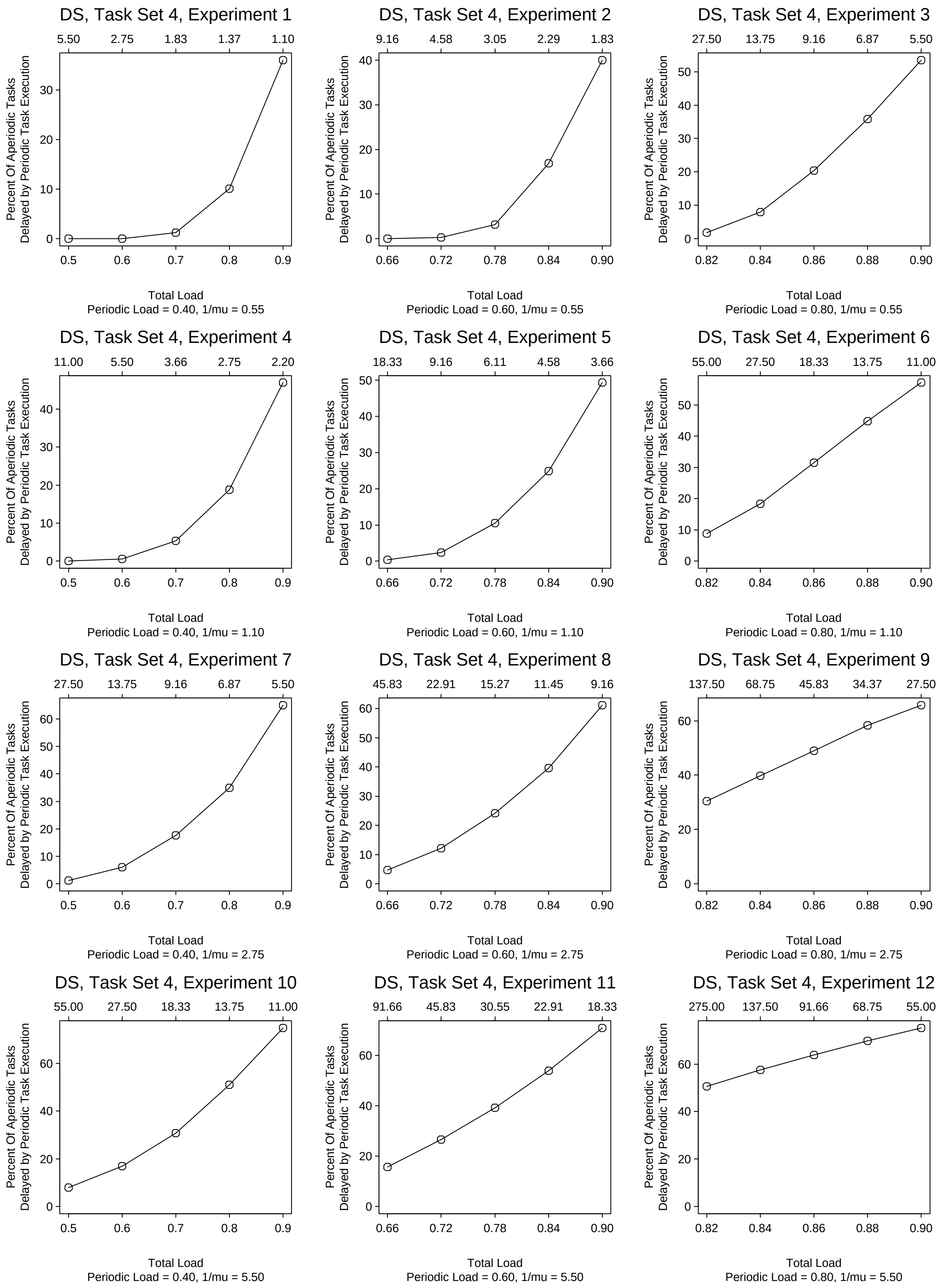


Figure 4-12: Percentage of Aperiodic Tasks Delayed by Periodic Task Execution for Task Set 4 and the DS Algorithm

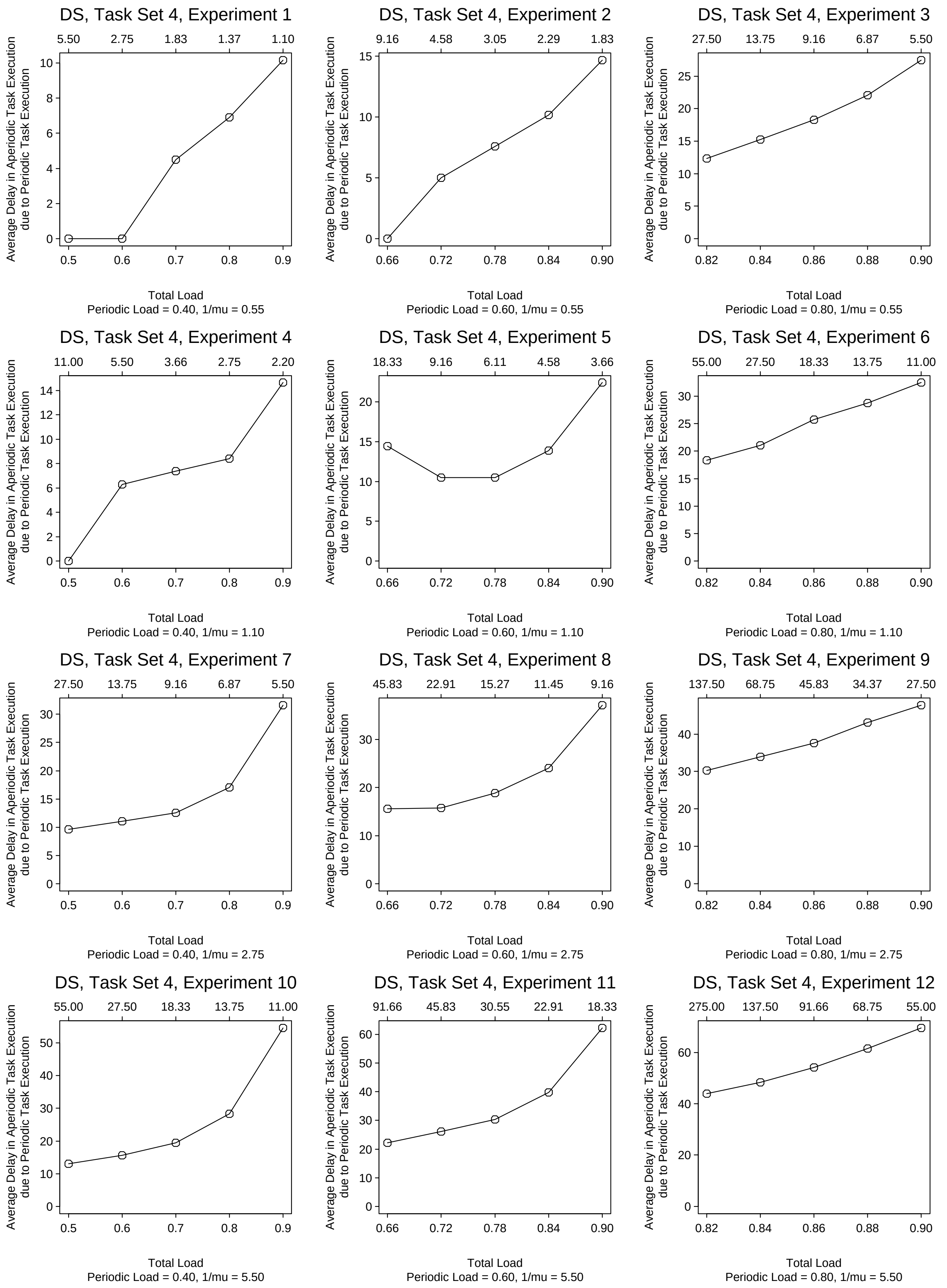


Figure 4-13: Average Delay Duration for Aperiodic Tasks for Task Set 4 and the DS Algorithm

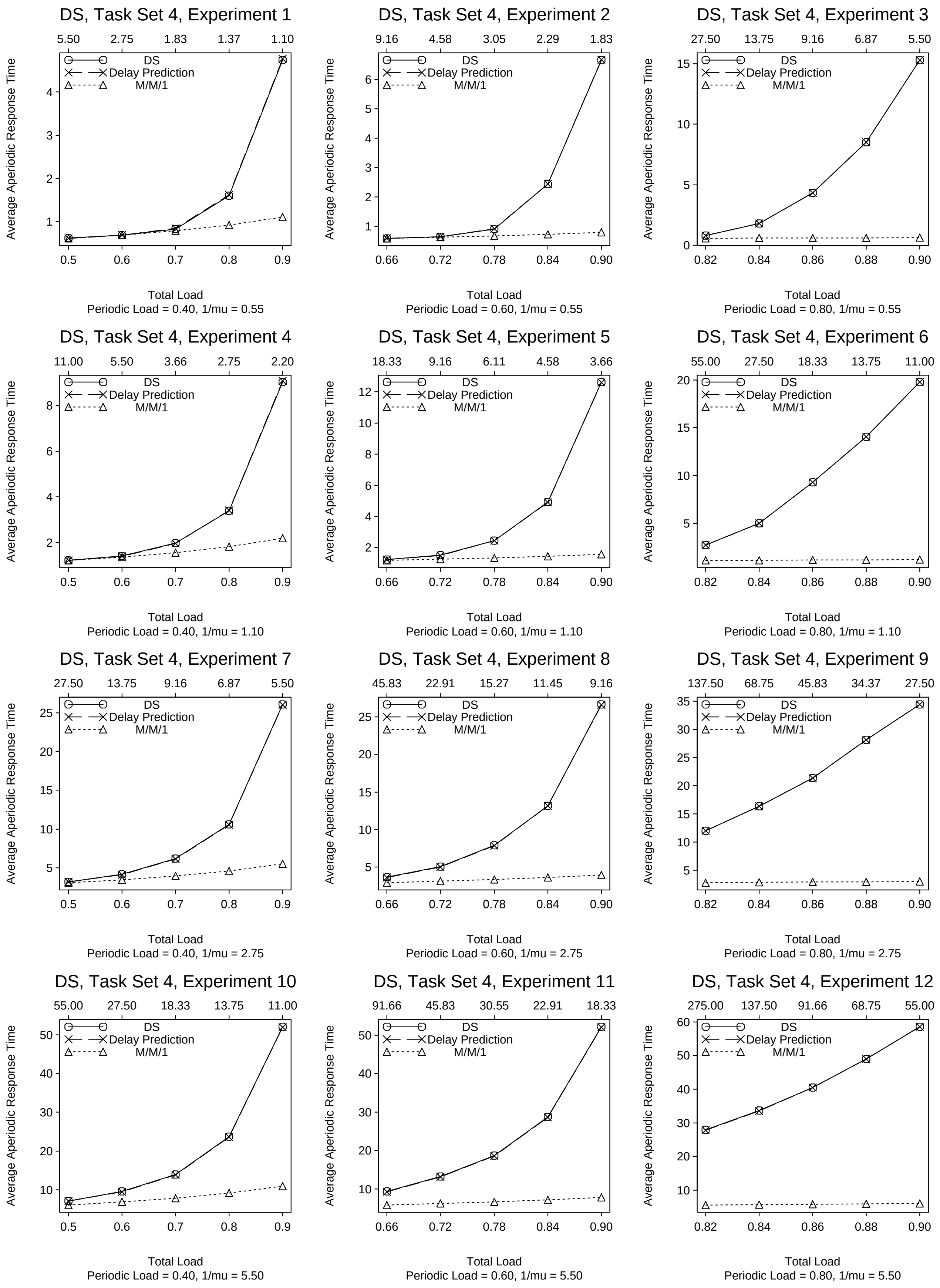


Figure 4-14: Prediction of Average Response Time Using Delay Data for Task Set 4 and the DS Algorithm

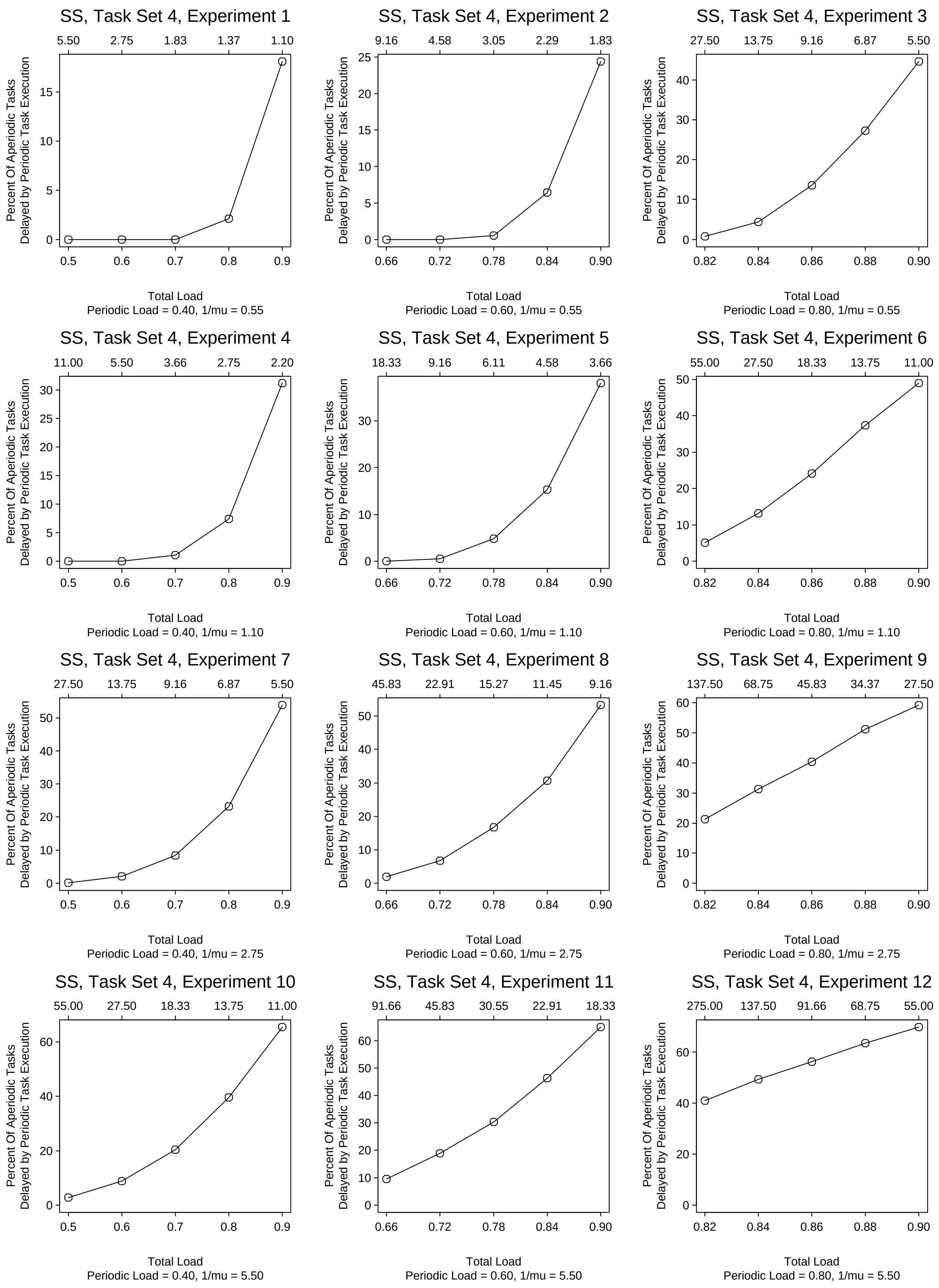


Figure 4-15: Percentage of Aperiodic Tasks Delayed by Periodic Task Execution for Task Set 4 and the SS Algorithm

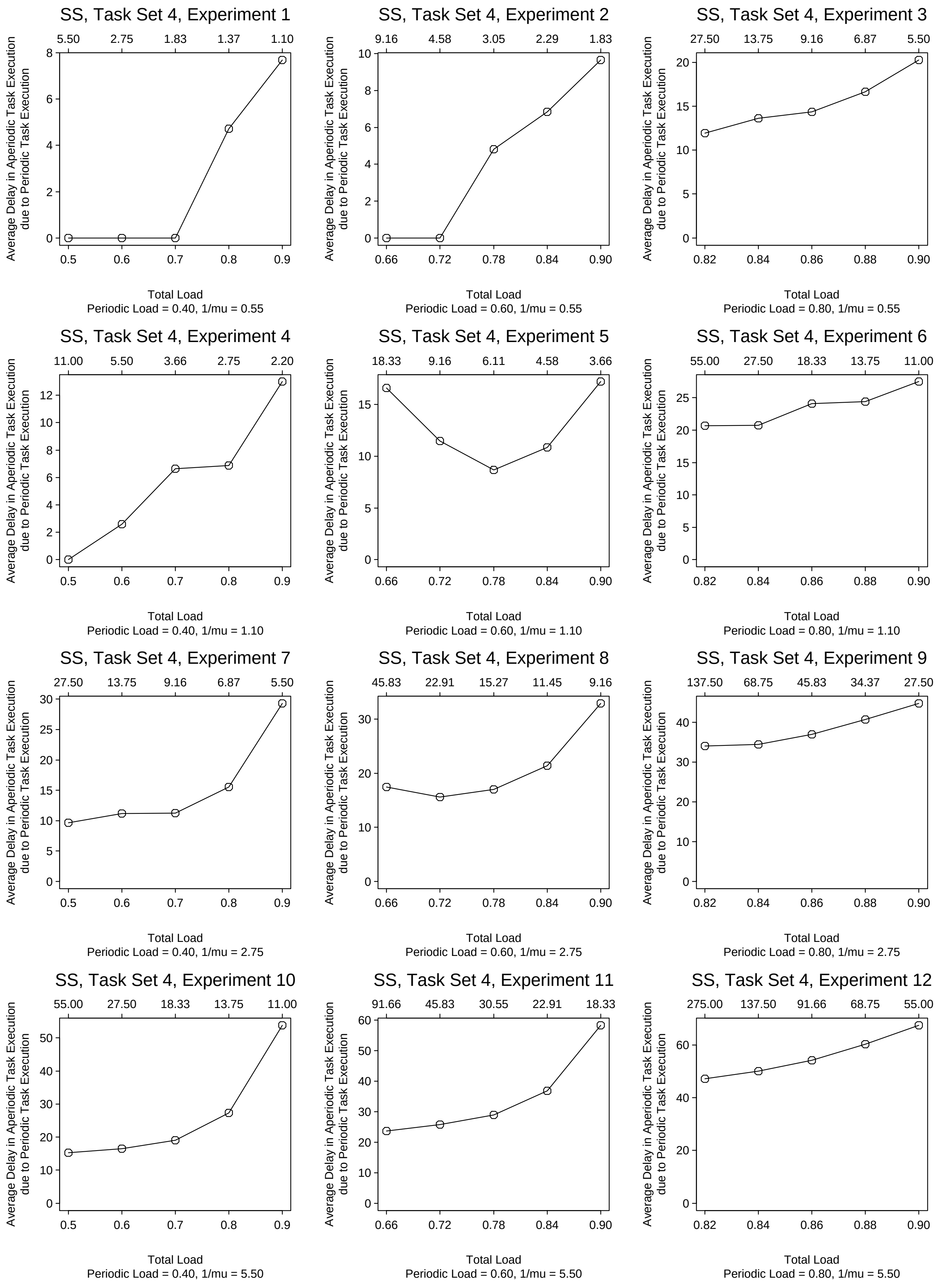


Figure 4-16: Average Delay Duration for Aperiodic Tasks for Task Set 4 and the SS Algorithm

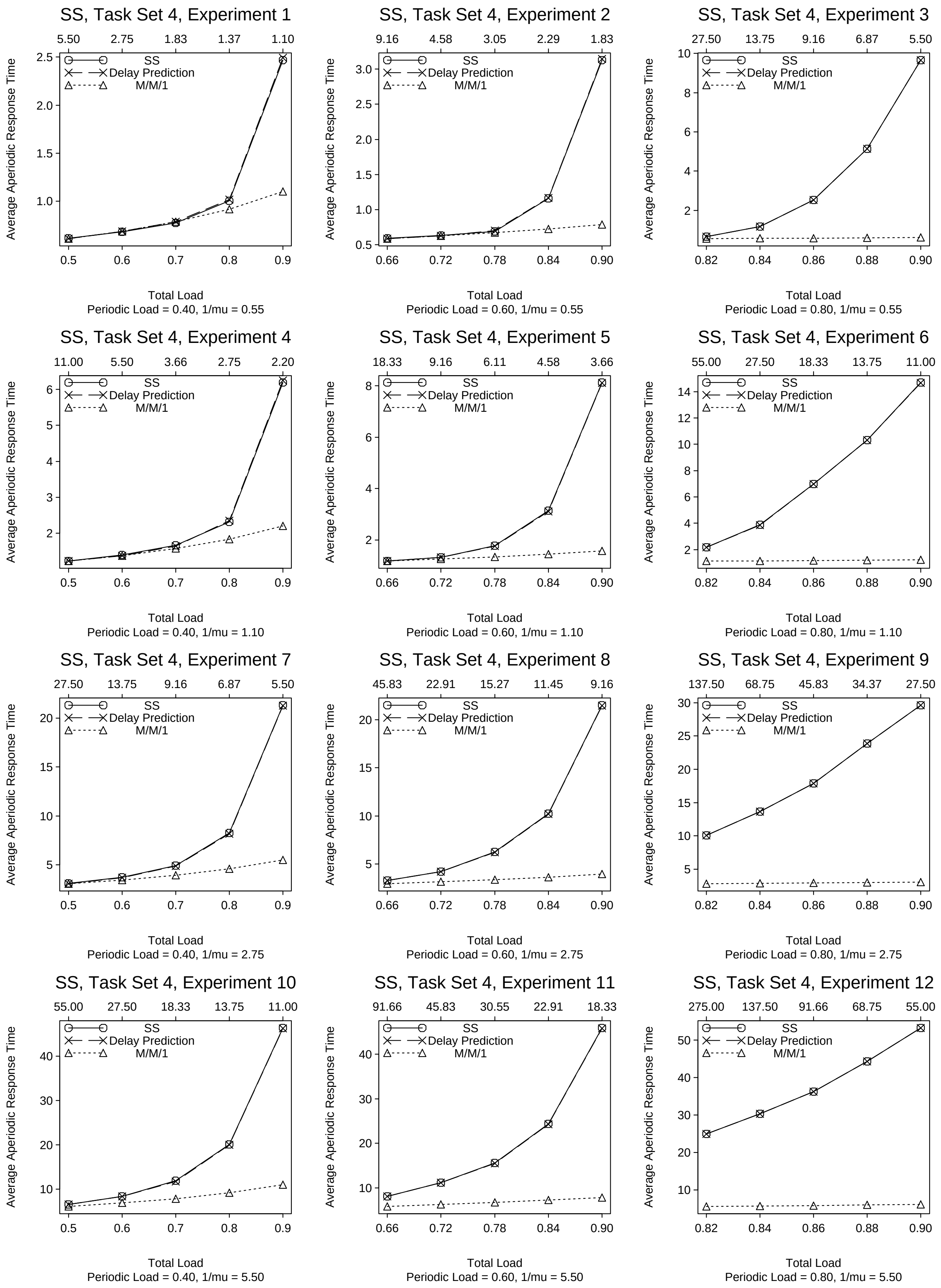


Figure 4-17: Prediction of Average Response Time Using Delay Data for Task Set 4 and the SS Algorithm

Using the DS or SS algorithms, aperiodic tasks will only encounter a delay in execution due to the execution of periodic tasks when the DS or SS algorithm has exhausted its execution time and both periodic tasks and aperiodic tasks are pending. Because the DS or SS server has no high priority execution time available, a pending periodic task will be selected for execution and this will cause a delay in the servicing of the aperiodic tasks. We refer to this as an aperiodic server *overrun* and, whenever an overrun occurs, the average aperiodic response time will be larger than the ideal response time. We can confirm this characterization of aperiodic response time performance by referring to Figure 4-18 which shows the average time spent in server overruns for DS and SS algorithms using the experiments described in Section 3.2. By comparing the response time performance of the DS and SS algorithms presented in Figure 3-4 to the percentage of time spent in overruns in Figure 4-18, one can clearly see that whenever the percentage of time spent in overruns is zero, the average aperiodic response time is equal to the ideal M/M/1 average response time. Therefore, to predict the maximum aperiodic load at which an ideal aperiodic response time is attainable, we need to predict the aperiodic load at which server overruns begin to occur.

An overrun can occur when the amount of aperiodic work that arrives during the server's period exceeds the full execution time of the server. To predict the conditions where overruns are likely to occur, we need an estimate of how much aperiodic work can arrive during the server's period. If the arriving work exceeds the server's capacity, then an overrun can occur and cause the average aperiodic response time to be worse than the ideal response time. Because we are modelling the aperiodic tasks with a Poisson arrival process, the probability of an overrun occurring is never zero. However, the probability of server overrun can be very low. Therefore, we want to develop an equation that indicates the aperiodic load for which a small percentage of aperiodic server periods are overrun. Overruns will begin to occur whenever the amount of aperiodic work that arrives during the aperiodic server's period just begins to equal the server's execution time.

We use the following parameters to estimate the probability of the occurrence of a server overrun:

λ: the the mean arrival rate of the aperiodic tasks
μ: the mean service rate of aperiodic tasks
C: the server's execution time
T: the server's period.

We first consider the case of a Poisson arrival process (with rate λ) and exponentially distributed service times (with parameter μ). We define τ to be the first time at which the total work of the arriving aperiodic tasks equals or exceeds the server's execution time, **C**. Let S_i represent the service time of the i'th arriving aperiodic task. Let N(t) be the number of aperiodic task arrivals. We define A(t) to be the accumulated aperiodic work at time t as:

$$A(t) = \sum_{i=1}^{N(t)} S_i \qquad (32)$$

We define τ to be the earliest time t such that A(t) equals or exceeds C as:

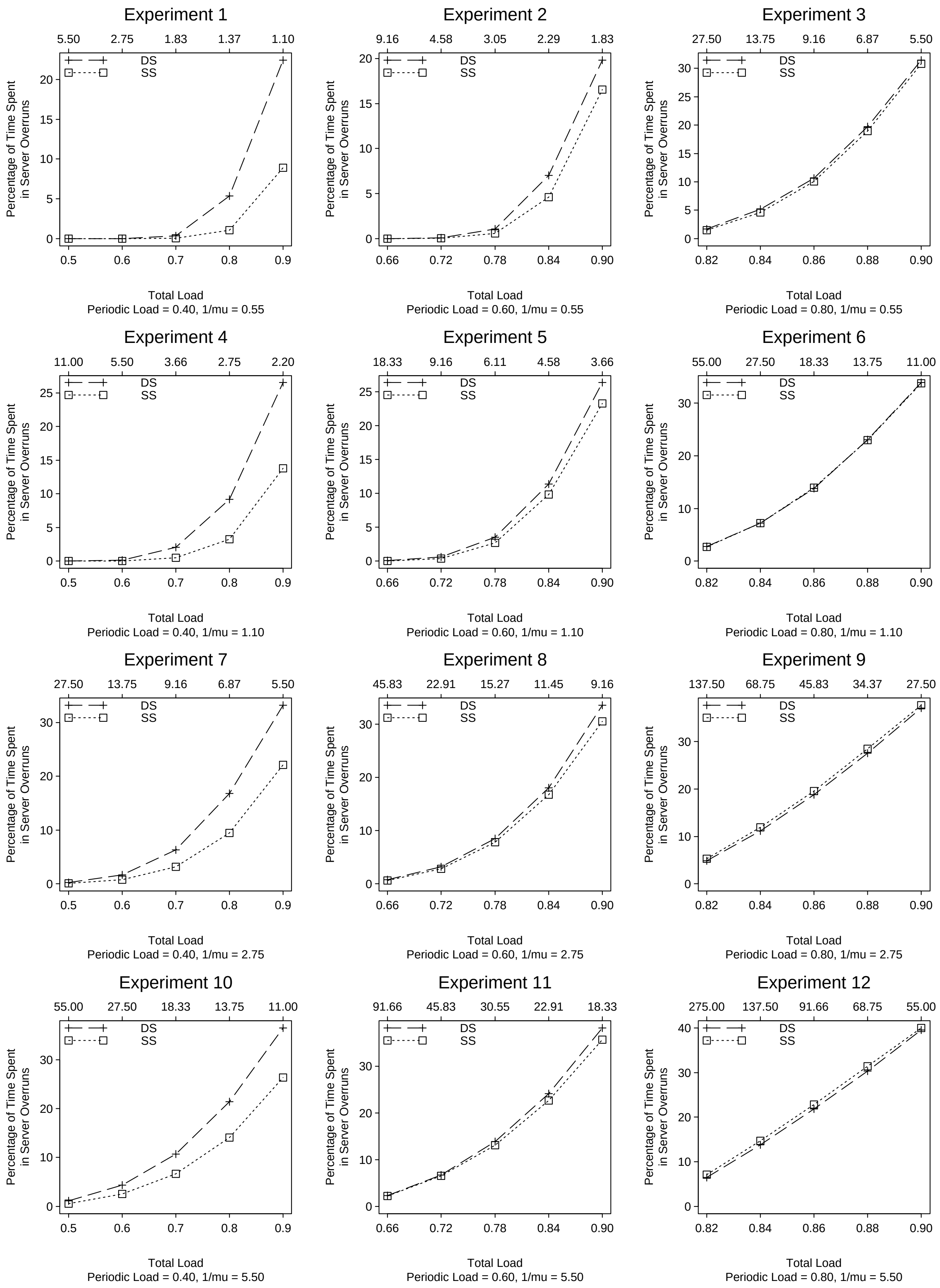


Figure 4-18: Average Time Spent in Server Overruns for the DS and SS Algorithms

$$\tau = \min\{t \mid A(t) \geq C\} \tag{33}$$

Let M be the number of aperiodic tasks necessary to require at least **C** units of work:

$$M = \min\{m \mid S_1 + \cdots + S_m \geq C\} \tag{34}$$

We now can express the following probability:

$$P(M = m) = P(S_1 + \cdots + S_m \geq C) - P(S_1 + \cdots + S_{m-1} \geq C) \tag{35}$$

We can now derive the Laplace transform of the distribution of τ by noting that it will take M aperiodic tasks to reach τ and the time to reach τ has a gamma distribution with parameters M and λ (the distribution of the time of the M'th event in a Poisson process). Therefore, from the transform of a gamma distribution with parameters M and λ we have:

$$E\left[e^{-s\tau} \mid M\right] = \left(\frac{\lambda}{\lambda + s}\right)^M \tag{36}$$

The derivation of the Laplace transform of tau proceeds as follows:

$$
\begin{aligned}
f(s) &= E\left[e^{-s\tau}\right] \\
&= E\left[E\left[e^{-s\tau} \mid M\right]\right] = E\left[\left(\frac{\lambda}{\lambda+s}\right)^{M}\right] \\
&= \sum_{m=1}^{\infty}\left(\frac{\lambda}{\lambda+s}\right)^{m} P(M=m) \\
&= \sum_{m=1}^{\infty}\left(\frac{\lambda}{\lambda+s}\right)^{m}\left[P(S_1+\cdots+S_m \geq C) - P(S_1+\cdots+S_{m-1} \geq C)\right] \\
&= \left(1 - \frac{\lambda}{\lambda+s}\right)\sum_{m=1}^{\infty}\left(\frac{\lambda}{\lambda+s}\right)^{m}\left[P(S_1+\cdots+S_m \geq C)\right] \\
&= \frac{s}{\lambda+s}\sum_{m=1}^{\infty}\left(\frac{\lambda}{\lambda+s}\right)^{m}\int_{C}^{\infty}\frac{\mu^m x^{m-1} e^{-\mu x}}{(m-1)!}dx \\
&= \frac{s}{\lambda+s}\int_{C}^{\infty}\exp(-\mu x)\left(\sum_{m=1}^{\infty}\left(\frac{\lambda\mu}{\lambda+s}\right)^{m}\frac{x^{m-1}}{(m-1)!}\right)dx \\
&= \frac{s}{\lambda+s}\int_{C}^{\infty}\exp(-\mu x)\left(\frac{\lambda\mu}{\lambda+s}\sum_{m=0}^{\infty}\frac{\left(\frac{\lambda\mu x}{\lambda+s}\right)^{m}}{m!}\right)dx \\
&= \frac{s}{\lambda+s}\int_{C}^{\infty}\exp(-\mu x)\,\frac{\lambda\mu}{\lambda+s}\,\exp\left(\frac{\lambda\mu x}{\lambda+s}\right)dx \\
&= \frac{s}{\lambda+s}\,\frac{\lambda\mu}{\lambda+s}\int_{C}^{\infty}\exp\left(-x\mu\left(1 - \frac{\lambda}{\lambda+s}\right)\right)dx \\
&= \frac{\lambda}{\lambda+s}\int_{C}^{\infty}\frac{\mu s}{\lambda+s}\exp\left(-x\mu\frac{s}{\lambda+s}\right)dx \\
f(s) &= \frac{\lambda}{\lambda + s}\exp\left(\frac{-\mu s C}{\lambda + s}\right)
\end{aligned}
\tag{37}
$$

Equation 37 can be used to determine the mean and the variance of τ, the time at which the sum of the aperiodic work arriving between $t = 0$ and $t = \tau$, equals **C**.

$$f'(0) = E[\tau] = \frac{1 + \mu C}{\lambda} \tag{38}$$

$$\sigma_\tau^2 = E[\tau^2] - (E[\tau])^2 = f''(0) - (f'(0))^2 = \frac{1 + 2\mu C}{\lambda^2} \tag{39}$$

A simple example can be used to understand Equation 38. Let the mean aperiodic service time ($1/\mu$) be 5, the mean interarrival time ($1/\lambda$) be 10, and the server execution time (**C**) be 20. On the average, one would expect 20 units of aperiodic work to arrive in (20/5) * 10 = 40 units of time. However, it will take, on the average, 10 units of time for the first work to arrive. So, the average value for τ should be 50, which is the value given by the Equation 38.

Given μ, **T**, and **C**, we want to predict the mean arrival rate ,λ, at which the aperiodic server begins to be overrun. If we assume that τ is normally distributed, we can get an estimate of the minimum time during which aperiodic work equal to **C** can arrive during the server's period by computing the value of τ for which 5% of the server periods will be overrun as follows:

$$\tau_{over} = T = E[\tau] - 1.645\sigma_\tau \tag{40}$$

The aperiodic load for which τ_{over} equals the server's period should be close to the aperiodic load at which the average aperiodic response time becomes greater than the M/M/1 response time. Given Equation 40 and the values for **C**, **T**, and μ, we can compute the corresponding mean arrival rate, λ_{over}:

$$\lambda_{over} = \frac{1 + \mu C - 1.645\sqrt{1 + 2\mu C}}{T} \tag{41}$$

Figures 4-19 and 4-20 present the computed value for λ_{over} and the average aperiodic response times for the DS and SS algorithms using task set 4 (see Appendix I). The vertical lines marked with "5%" in the graphs of Figures 4-19 and 4-20 indicate the aperiodic load and the value of λ_{over} for the specified mean aperiodic service time and the server execution time and period (see Appendix I for the server execution times for task set 4). Note that for aperiodic loads less than indicated by the λ_{over} line, the average aperiodic response time for both the DS and SS algorithm is very close to the ideal M/M/1 response time curve. Also, note that the value for λ_{over} is less than the minimum aperiodic load simulated for some of the experiments in Figures 4-19 and 4-20 and no vertical line for λ_{over} is present for these graphs. From Figures 4-19 and 4-20 we see that value of λ_{over} gives a good indication of the maximum aperiodic load at which M/M/1 performance can be obtained.

The average aperiodic response time for a significant number of the experiments presented in Figures 4-19 and 4-20 is greater than the M/M/1 average response time (e.g. Experiments 6, 8, 9, 10, 11, and 12 in Figure 4-20). If we change the aperiodic service time distribution to a deterministic distribution (i.e., a constant service time) the decrease in the variability of the aperiodic service time should allow us to use an M/D/1 queueing model to predict a larger range of DS and SS performance than is possible with an exponential service time distribution and an M/M/1 queueing model.

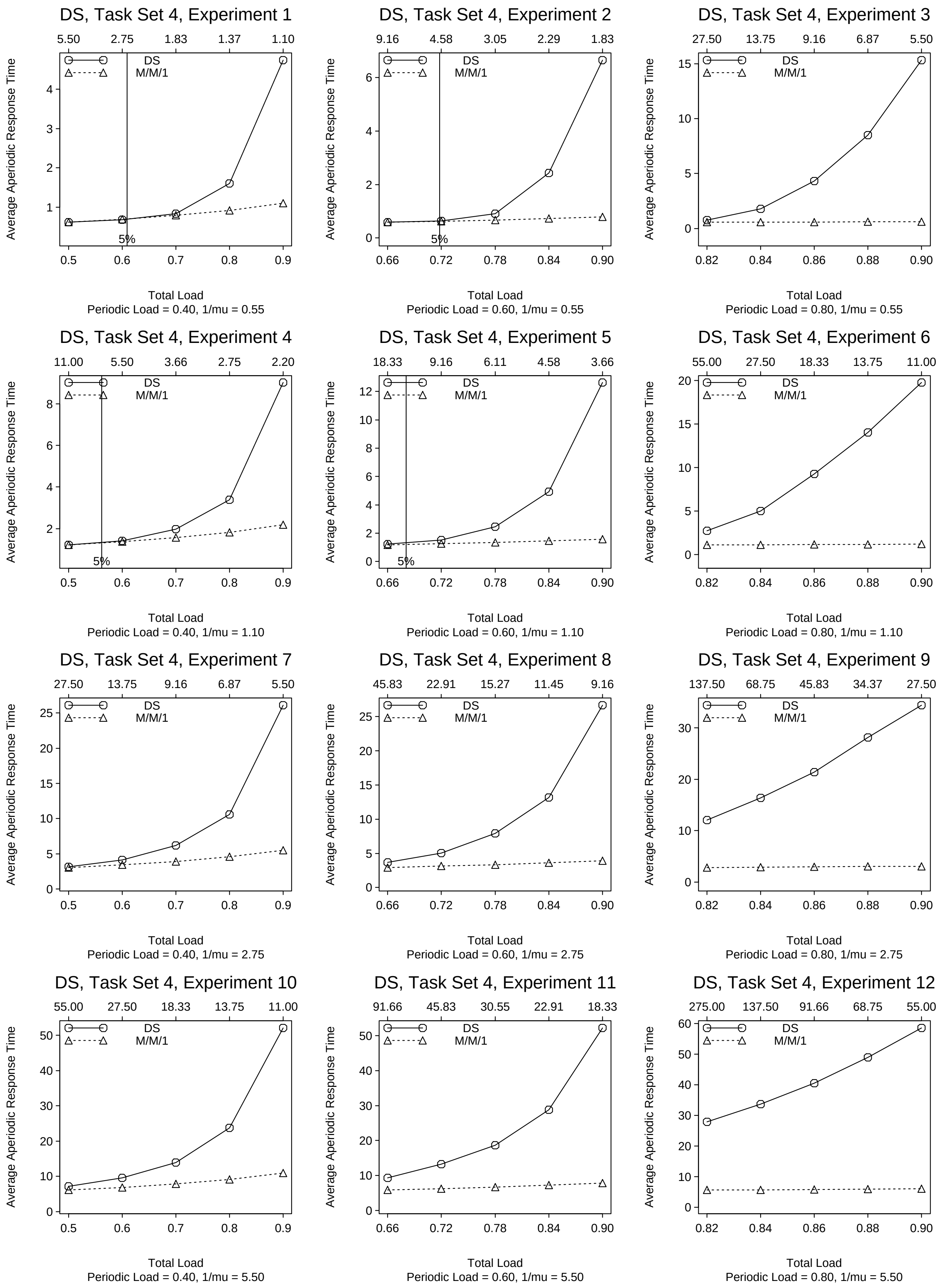


Figure 4-19: Estimate of M/M/1 Response Time Range for the Task Set 4 and the DS Algorithm

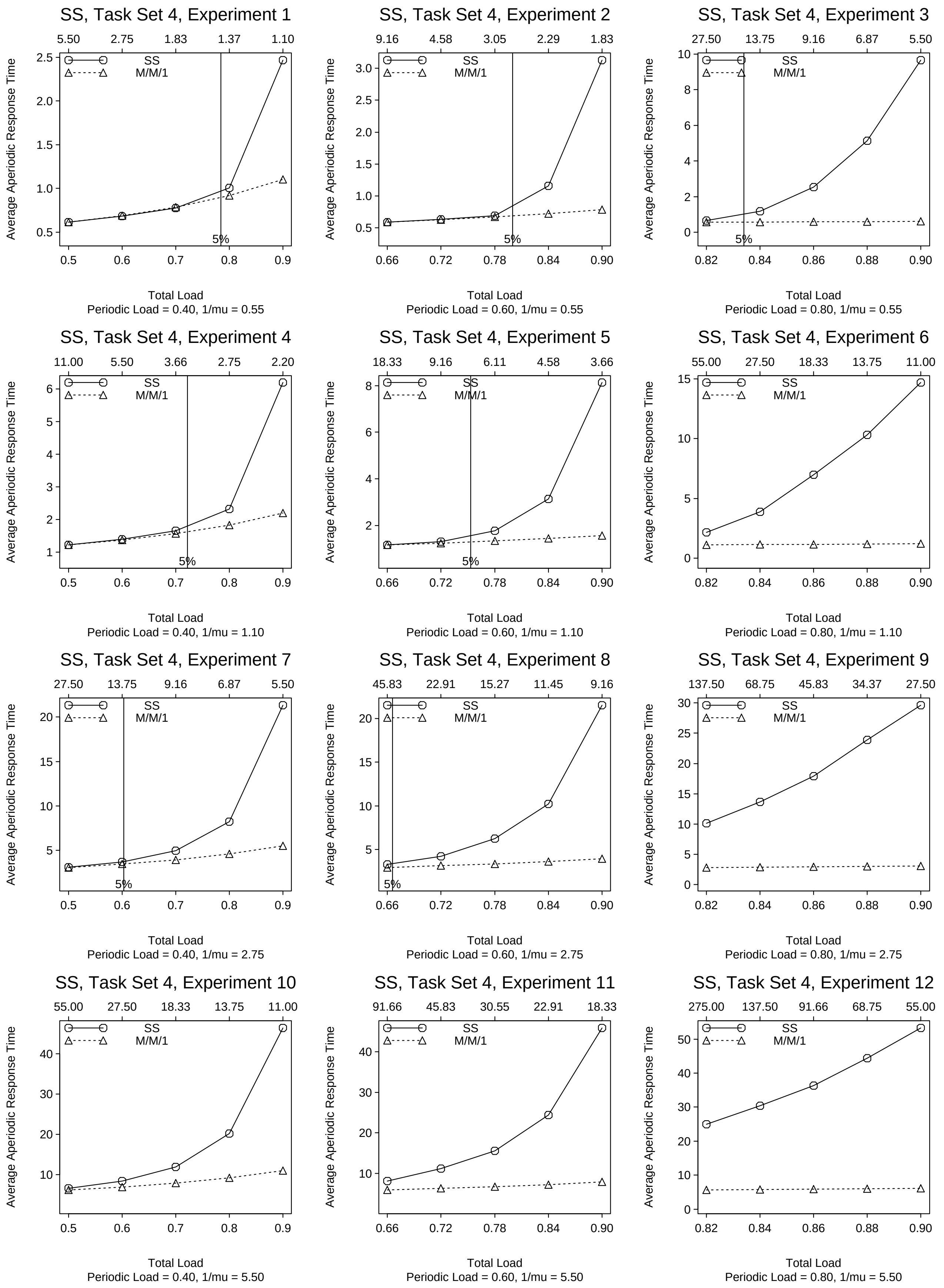


Figure 4-20: Estimate of M/M/1 Response Time Range for the Task Set 4 and the SS Algorithm

The derivation of λ_{over} for aperiodic tasks with Poisson arrivals and deterministic service times proceeds as follows. Since the aperiodic service time is constant, we need only to determine the probability that the number of arrivals the Poisson process meets or exceeds the number of aperiodic tasks required to exhaust the aperiodic server's capacity. Let L be the number of aperiodic jobs that must arrive during one server period for an overrun to occur. The value of L can be directly calculated using the aperiodic service time ($1/\mu$) and the server's execution time (**C**):

$$L = \lfloor \mu C \rfloor + 1 \tag{42}$$

Next, let N(T) be the number of aperiodic tasks that arrive in one server period. N(T) has a Poisson distribution with mean λT. We want to compute the probability that $N(T) \geq L$. We want to assume that N(T) is normally distributed (which is a valid assumption for large λT) and in switching from the discrete Poisson distribution for N(T) to a continuous normal distribution we make the following approximation:

$$P(N(T) \geq L) \approx P(N(T) > L - \frac{1}{2}) \tag{43}$$

The above can be expressed as:

$$P(N(T) > L - \frac{1}{2}) \approx \sum_{i=1}^{\infty} \exp(-\lambda T)\frac{(\lambda T)^i}{i!} \tag{44}$$

However, form is somewhat unusable for our purposes. Instead, we can put the expression for N(T) in Equation 43 in standard normal form as:

$$P(N(T) \geq L) \approx P\left(\frac{N(T) - \lambda T}{\sqrt{\lambda T}} > \frac{L - \frac{1}{2} - \lambda T}{\sqrt{\lambda T}}\right)$$

$$\approx 1 - \Phi\left(\frac{L - \frac{1}{2} - \lambda T}{\sqrt{\lambda T}}\right) \tag{45}$$

If we define λ_{over} for a Poisson arrival process with deterministic service times to the the arrival rate for which only 1% of the server periods are overrun we can use Equation 45 to compute λ_{over} as follows:

$$1 - \Phi\left(\frac{L - \frac{1}{2} - \lambda T}{\sqrt{\lambda T}}\right) = 0.01$$

$$\frac{L - \frac{1}{2} - \lambda T}{\sqrt{\lambda T}} = 2.33 \tag{46}$$

Solving Equation 46 for λ we obtain the equation for λ_{over} as:

$$\lambda_{over} = \left[\frac{2.33\sqrt{T} - \sqrt{(2.33)^2T + 4T(L - \frac{1}{2})}}{-2T}\right]^2 \tag{47}$$

Figures 4-19 and 4-20 present the computed value for λ_{over} and the average aperiodic response times for the DS and SS algorithms using task set 4 for aperiodic tasks with Poisson arrivals and a deterministic service time. The vertical lines marked with "1%" in the graphs of Figures 4-19 and 4-20 indicate the aperiodic load and the value of λ_{over} for the specified mean aperiodic service time and the server execution time and period. As in Figures 4-19 and 4-20, the computed values for λ_{over} in Figures 4-21 and 4-22 provide a good indication of range of aperiodic load for which a standard queueing model can be used to predict the average aperiodic response time performance for the DS and SS algorithms. Also, note that the range of aperiodic loads for which an M/D/1 queueing model accurately predicts aperiodic response time performance for deterministic aperiodic service times is larger than the range for exponentially distributed service times and an M/M/1 queueing model.

To summarize the approach described in this section, we derive equations for ρ_{over}, the estimate of the maximum aperiodic load for which either an M/M/1 or M/D/1 queueing theory model can be used to predict the average aperiodic response time. The equations below were obtained from the equations for λ_{over} (Equations 41 and 47) by dividing by μ. We also present the queueing theory equations for the average response times for the M/M/1 and M/D/1 queueing systems [Kleinrock 75]. In these equations we note the aperiodic load as $\rho = \lambda/\mu$.

M/M/1:

$$\rho_{over} = \frac{1 + \mu C - 1.645\sqrt{1 + 2\mu C}}{\mu T} \tag{48}$$

$$\text{for } \rho \le \rho_{over},\ W \approx \frac{\frac{1}{\mu}}{(1 - \rho)} \tag{49}$$

M/D/1:

$$\rho_{over} = \left[\frac{2.33\sqrt{T} - \sqrt{(2.33)^2T + 4T(L - \frac{1}{2})}}{-2T\sqrt{\mu}}\right]^2 \tag{50}$$

$$\text{for } \rho \le \rho_{over},\ W \approx \frac{\rho}{2\mu(1 - \rho)} + \frac{1}{\mu} \tag{51}$$

Appendix III presents a summary and an example of how to apply the rate monotonic approach with the above equations to schedule a real-time task set.)

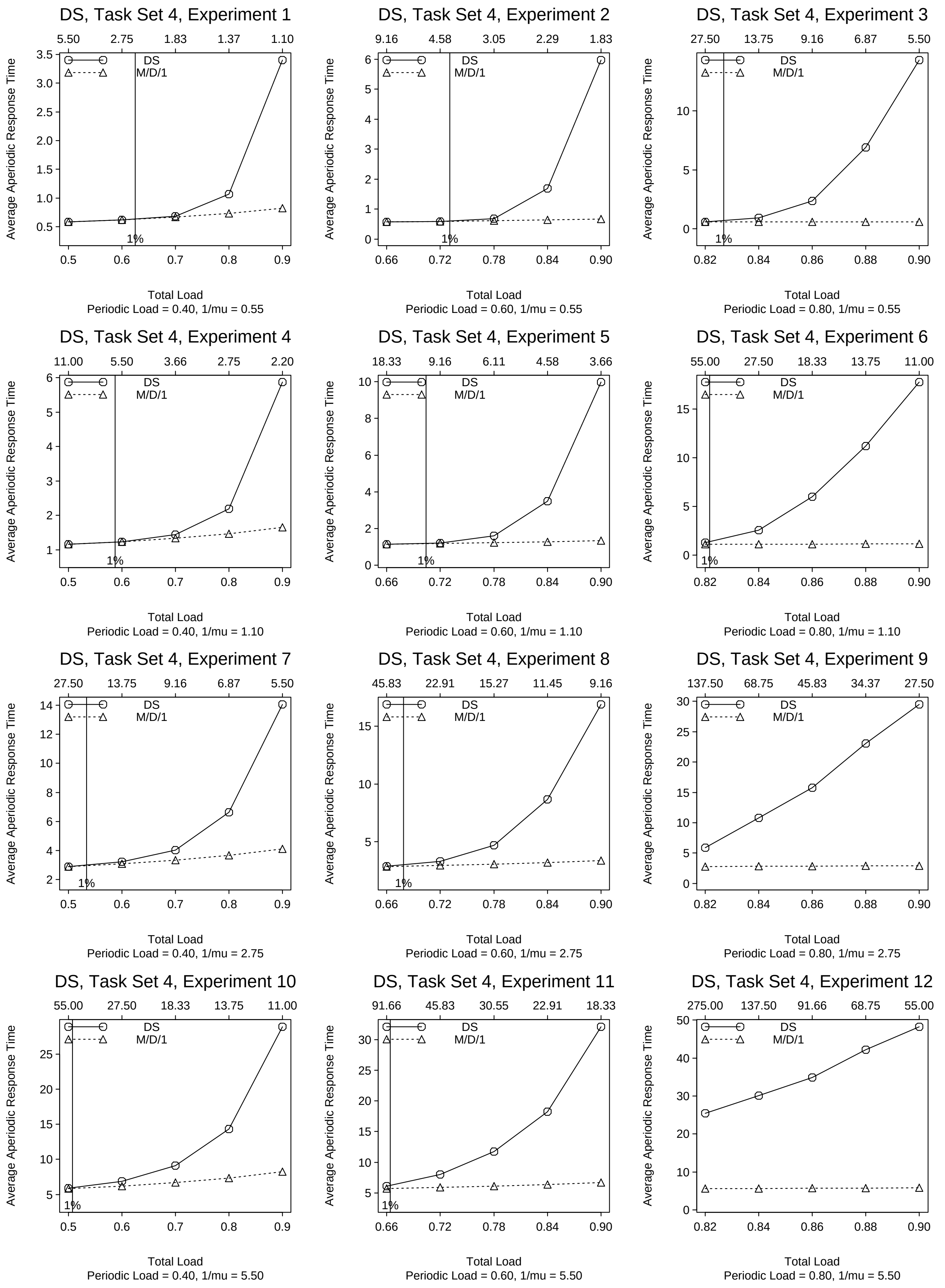


Figure 4-21: Estimate of M/D/1 Response Time Range for Task Set 4 and the DS Algorithm

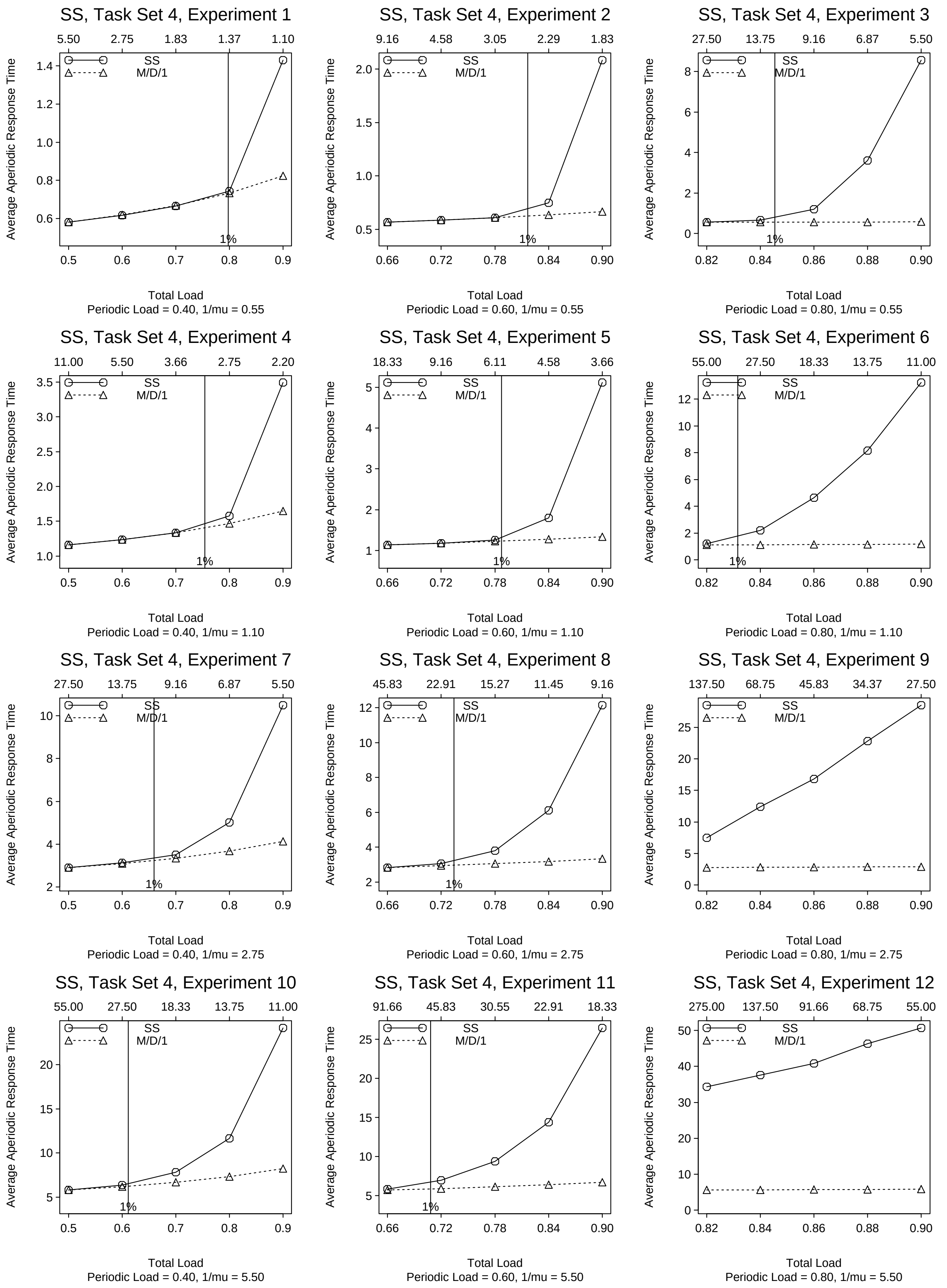


Figure 4-22: Estimate of M/D/1 Response Time Range for the Task Set 4 and the SS Algorithm

5. Sporadic Server Implementation

5.1 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 a high priority task when some sporadic server capacity is replenished (as is possible in the full Ada runtime implementation described in Section 5.2).

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 later in Section 5.3.4). 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 from the sporadic server, 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 exit of 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 5-1. 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 5-2.

The task *Sporadic_Server* presented in Figure 5-1 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 the *accept* statement 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 5-1 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 the 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 that is 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. Next, it is 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 upon 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.

It is worth mentioning here that the operation of the *delay* alternative of the SWDA can be different from what is desired. If the evaluation of the *delay* expression is preempted after reading the clock but before executing the *delay*, the effect will be to make the delay longer than desired [Borger 89]. This will at best result in wasted server capacity because the server will be replenished at a later time than desired. At worst, a preemption 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 available. A solution to this problem would be to support a *delay_until* [Borger 89] 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 preemption problem described above.

```
with Calendar; use Calendar;
with Aperiodic_Tasks; use Aperiodic_Tasks;
with SS_Replenishment_Queue_Manager; use SS_Replenishment_Queue_Manager;

package body Sporadic_Service is

    SS_Period          : constant duration := Period of the Sporadic Server;
    SS_Max_Exec_Time   : constant duration := Maximum Execution Budget of the Sporadic Server;

    Available_Exec_Time : duration := SS_Max_Exec_Time;
    Consumed_Exec_Time  : duration := 0.0;

    Exec_Begin_Time     : time;
    Next_Rep            : replenishment;
    Reps_Are_Pending    : boolean := FALSE;

    task body Sporadic_Server is
    begin
        loop
            select
                when Available_Exec_Time >= Aperiodic_Task_1_Max_Exec_Time =>
                accept Obtain_Service_For_Aperiodic_Task_1;

                    Exec_Begin_Time := Clock;
                    Service_Aperiodic_Task_1;
                    Consumed_Exec_Time := Max_Exec_Time_Aperiodic_Task_1;

                end Obtain_Service_For_Aperiodic_Task_1;

            or  ...

            or
                when Available_Exec_Time >= Aperiodic_Task_N_Max_Exec_Time =>
                accept Obtain_Service_For_Aperiodic_Task_N;

                    Exec_Begin_Time := Clock;
                    Service_Aperiodic_Task_N;
                    Consumed_Exec_Time := Max_Exec_Time_Aperiodic_Task_N;

                end Obtain_Service_For_Aperiodic_Task_N;

            or  when Reps_Are_Pending => delay Next_Rep.Rep_Time - Clock;

                if Replenishment_Queue_Empty then
                    Available_Exec_Time := SS_Max_Exec_Time;
                    Reps_Are_Pending := FALSE;
                else
                    Available_Exec_Time := Available_Exec_Time + Next_Rep.Rep_Amount;
                    Dequeue_Replenishment(Next_Rep);
                end if;

            end select;

            if Consumed_Exec_Time > 0.0 then

                Available_Exec_Time := Available_Exec_Time - Consumed_Exec_Time;

                if Reps_Are_Pending then
                    Enqueue_Replenishment((Rep_Time   => Exec_Begin_Time + SS_Period,
                                           Rep_Amount => Consumed_Exec_Time));
                else
                    Next_Rep.Rep_Time := Exec_Begin_Time + SS_Period;
                    Next_Rep.Rep_Amount := Consumed_Exec_Time;
                    Reps_Are_Pending := TRUE;
                end if;

                Consumed_Exec_Time := 0.0;

            end if;

        end loop;
    end Sporadic_Server;
end Sporadic_Service;
```

Figure 5-1: Application-Level Sporadic Server

```
package Aperiodic_Tasks is

    Aperiodic_Task_1_Max_Exec_Time : constant duration := max exec time;
    procedure Service_Aperiodic_Task_1;

    Aperiodic_Task_2_Max_Exec_Time : constant duration := max exec time;
    procedure Service_Aperiodic_Task_2;

    ...

    Aperiodic_Task_N_Max_Exec_Time : constant duration := max exec time;
    procedure Service_Aperiodic_Task_N;

end Aperiodic_Tasks;

package SS_Replenishment_Queue_Manager is

    type replenishment is record
        Rep_Time    : time;
        Rep_Amount  : duration;
    end record;

    procedure Enqueue_Replenishment(New_Replenishment : in replenishment);

    procedure Dequeue_Replenishment(Old_Replenishment : out replenishment);

    function  Replenishment_Queue_Empty return boolean;

end SS_Replenishment_Queue_Manager;
```

Figure 5-2: Specifications for the Aperiodic_Tasks and SS_Replenishment_Queue_Manager Packages

5.2 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 2.1 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.

5.2.1 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 [Sha 89b], 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.

5.2.2 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.

5.2.3 Data Structures for Scheduling Aperiodic Tasks that Use Sporadic Servers

5.2.3.1 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 of 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 5.2.3.3. 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).

5.2.3.2 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 [Sha 89a]).

5.2.3.3 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 5.2.5.4):

- *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 5.2.3.7 and 5.2.4).
- *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.

5.2.3.4 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 5.2.5.

- *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.

5.2.3.5 Sporadic Server Data Structure Example

Figure 5-3 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 5-3. 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 5.2.5 and 5.2.6.

In Figure 5-3, 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 5-3, 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.

5.2.3.6 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

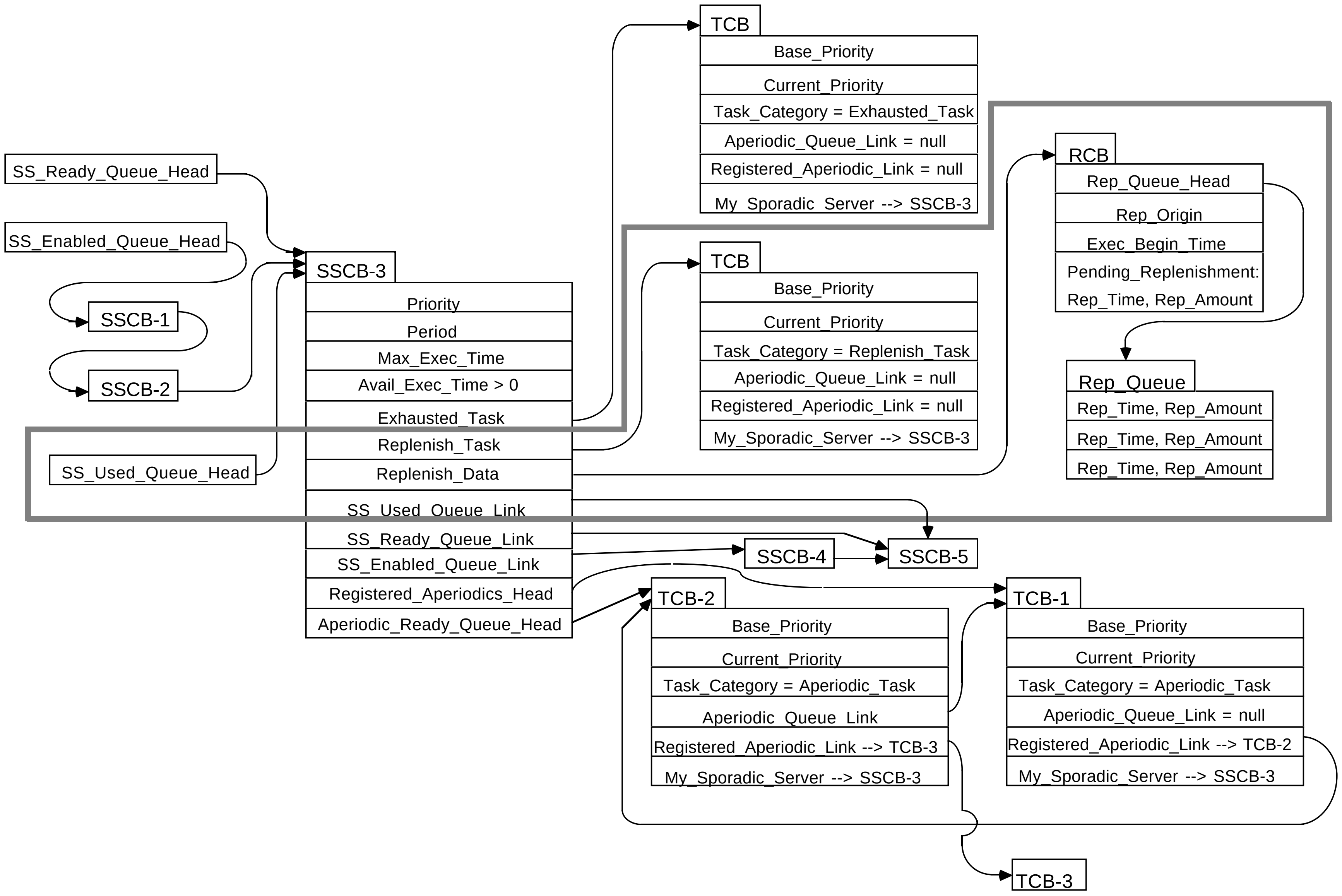


Figure 5-3: Sporadic Server Runtime Data Structures

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 5-4. 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 5.2.5.

```
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;

procedure Remove_Task_From_Ready_Queue(task : TCB) is
begin

  Remove the task from the Task_Ready_Queue;

  if task.Task_Category = Aperiodic_Task then

      Remove the task from its sporadic server's Aperiodic_Ready_Queue;

      if task.My_Sporadic_Server.On_SS_Ready_Queue and then
         (task.My_Sporadic_Server.Aperiodic_Ready_Queue_Head = null or else
          task.My_Sporadic_Server.Avail_Exec_Time = 0.0) then

         Remove the task's sporadic server from the SS_Ready_Queue;

      end if;

  end if;

end Remove_Task_From_Ready_Queue;
```

Figure 5-4: Add_Task_To_Ready_Queue and Remove_Task_From_Ready_Queue

5.2.3.7 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 preempted 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 5.2.6.1).

5.2.4 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 the 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 5.2.6.1). The time at which execution begins is recorded in the sporadic server's Replenishment_Control_Block (the RCB is discussed in Section 5.2.5).

```
procedure Execute_Next_Task is

  Next_Task : TCB := Task_Ready_Queue;  -- the head of the Task_Ready_Queue

begin

  if SS_Enabled_Queue /= null then  --  Are any sporadic servers are enabled?

      --------
      --  Check if an aperiodic task can execute now using its sporadic server.
      --------
      if SS_Ready_Queue /= null and then
         SS_Ready_Queue.Priority >= Task_Ready_Queue.Priority and then
         Task_Ready_Queue.Priority > SS_Ready_Queue.Aperiodic_Queue.Base_Priority
         then

          Next_Task := SS_Ready_Queue.Aperiodic_Queue;

          --------
          --  Update the sporadic server information in the TCB.
          --------
          Next_Task.Current_Priority            := SS_Ready_Queue.Priority;
          Next_Task.Using_Sporadic_Server       := True;

          Move the aperiodic task to the head of the Task_Ready_Q;

          --------
          --  Place the Exhausted_Task on the delay queue with a delay of
          --  Avail_Exec_Time time units.
          --------
          Add_Task_To_Delay_Queue (SS_Ready_Queue.Exhausted_Task,
                                   SS_Ready_Queue.Avail_Exec_Time);

          --------
          --  Remember the time when execution began for the aperiodic task.
          --------
          Next_Task.Rep_Data.Exec_Begin_Time := Clock;

      end if;

      --------
      --  Mark the consumption of sporadic server execution time if we are
      --  changing to a new task and the last task was using a sporadic
      --  server.
      --------
      if Current_Task.Using_Sporadic_Server and Current_Task /= Next_Task then
          Mark_SS_Consumption;
      end if;

      --------
      --  Remember the priority level of the previously executing task and the
      --  the next task to execute.
      --------
      Previous_Priority_Level := Next_Priority_Level;
      Next_Priority_Level := Next_Task.Current_Priority;

      --------
      --  Set the active/idle status of sporadic server priority levels and
      --  schedule any replenishments as necessary.
      --------
      Check_SS_Replenishment;

  end if;

  Current_Task := Next_Task;

  Execute the task at the head of the Task_Ready_Queue;

end Execute_Next_Task;
```

Figure 5-5: Execute_Next_Task

Figure 5-5 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 an 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 5.2.5 and 5.2.6.

5.2.5 Data Structures For Scheduling Sporadic Server Replenishments

5.2.5.1 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.

5.2.5.2 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.

5.2.5.3 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.

5.2.5.4 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 5.2.5.5, contains all the information concerning the outstanding replenishments for a sporadic server.

5.2.5.5 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 2-4 in Section 2.1).
- *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.

5.2.5.6 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 5-3 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.

5.2.6 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.

5.2.6.1 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 preempted by a higher priority task.
2. The aperiodic task suspends.
3. The aperiodic task completes execution.
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 5-5) 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 5-6 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.*

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.

*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.

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 Figure 5.2.6.1). 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 5.2.3.6).

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 5.2.6.3. 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 5.2.6.2) will schedule the sporadic server's replenishment when its priority level becomes idle.

```
procedure Mark_SS_Consumption is
    My_SS : access sporadic_server renames Current_Task.My_Sporadic_Server;
    Consumed_Execution_Time : duration;
begin

    --------
    --  Should the priority of the aperiodic task be changed?
    --------
    if Current_Task is not on the Task_Ready_Queue or
       My_SS.Exhausted_Task is not on the Delay_Queue then

        Current_Task.Using_Sporadic_Server := False;
        Current_Task.Current_Priority := Current_Task.Base_Priority;

    end if;

    --------
    --  Should the aperiodic task be re-queued on the Task_Ready_Queue?
    --------
    if Current_Task is on the Task_Ready_Queue and My_SS.Exhausted_Task is not on the Delay_Queue then

        Re-queue the Current_Task on the Task_Ready_Queue;

    end if;

    --------
    --  Should the Exhausted_Task be removed from the Delay_Queue?
    --------
    if My_SS.Exhausted_Task is on the Delay_Queue then

        Remove My_SS.Exhausted_Task from the Delay_Queue;

    end if;

    --------
    --  Adjust the sporadic server’s available execution time.
    --------
    Consumed_Exec_Time := Clock - My_SS.Replenish_Data.Exec_Begin_Time;
    My_SS.Avail_Exec_Time := My_SS.Avail_Exec_Time - Consumed_Exec_Time;

    --------
    --  Should the sporadic server be removed from the SS_Ready_Queue?
    --------
    if My_SS.Aperiodic_Ready_Queue = null or My_SS.Avail_Exec_Time <= 0.0 then

        Remove My_SS from the SS_Ready_Queue;

    end if;

    --------
    --  Accumulate the execution time consumed during this active period.
    --------
    My_SS.Replenish_Data.Pending_Replenishment.Rep_Amount :=
        My_SS.Replenish_Data.Pending_Replenishment.Rep_Amount + Consumed_Exec_Time;

    --------
    --  If the sporadic server’s execution time is exhausted, then schedule the
    --  Pending_Replenishment.  Otherwise, add the sporadic server to the SS_Used_Queue.
    --------
    if My_SS.Avail_Exec_Time <= 0.0 then

        My_SS.Replenish_Data.Pending_Replenishment.Rep_Time :=
            My_SS.Replenish_Data.Rep_Origin + My_SS.Period;

        Queue_Pending_Replenishment(My_SS);

        My_SS.Avail_Exec_Time := 0.0;

    elsif not My_SS.On_SS_Used_Queue then

        Add My_SS to the SS_Used_Queue;

    end if;

end Mark_SS_Consumption;
```

Figure 5-6: Mark_SS_Consumption

5.2.6.2 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 5-5). The pseudo-code for Track_Active_Idle_Status is presented in Figure 5-7. 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 5.2.6.1) 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 5.2.6.3) 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 2-4 shows an example of this). The case is tested by the procedure Replenish_Sporadic_Server (discussed later in Section 5.2.6.4) 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.

```
procedure Track_Active_Idle_Status is
    a_ss : access sporadic_server;
    now  : time;
begin

    if Previous_Priority_Level < Next_Priority_Level then

        --------
        --  The priority levels in the range from just above the previous
        --  priority level up through the next priority level will become
        --  active when Execute_Next_Task executes the next task.  Set the
        --  Rep_Origin for each sporadic server within this range.
        --------

        now := Clock;

        --------
        --  Find the first sporadic server in the newly active range.
        --------
        a_ss := SS_Enabled_Queue;
        while a_ss /= null and then a_ss.Priority > Next_Priority_Level loop
            a_ss := a_ss.SS_Enabled_Queue_Link;
        end loop;

        --------
        --  Set the Rep_Origin for all the sporadic servers with priorities
        --  in the newly active range.
        --------
        while a_ss /= null and then a_ss.Priority > Previous_Priority_Level loop

            a_ss.Replenish_Data.Rep_Origin := now;

            a_ss := a_ss.SS_Enabled_Queue_Link;

        end loop;

    else

        --------
        --  The priority levels from the previous priority level down to the
        --  priority level just above the next priority level will become
        --  idle when Execute_Next_Task executes the next task.  Schedule
        --  replenishments for any sporadic servers within this range of
        --  priority levels that are currently on the SS_Used_Queue.
        --------

        --------
        --  Move down the SS_Used_Queue and, process each sporadic server with
        --  a priority greater than the next priority level.
        --------
        a_ss := SS_Enabled_Queue;
        while a_ss /= null and then a_ss.Priority > Next_Priority_Level loop

            Queue_Pending_Replenishment(a_ss);

            a_ss := a_ss.SS_Used_Queue_Link;

        end loop;

    end if;

end Track_Active_Idle_Status;
```

Figure 5-7: Track_Active_Idle_Status

5.2.6.3 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 5.2.6.4. 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 to 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 5-8.

```
procedure Queue_Pending_Replenishment (a_ss : access sporadic_server) is
begin

    --------
    --  Add the sporadic server's pending replenishment to the Rep_Queue.
    --------
    if  the sporadic server's Rep_Queue is not full  then

        Add the pending replenishment to the Rep_Queue;

        a_ss.Replenish_Data.Pending_Replenishment.Rep_Amount := 0.0;

        Remove the sporadic server from the SS_Used_Queue;

        --------
        --  If the Rep_Queue was just previously empty, then the sporadic
        --  server's Replenish_Task must be put on the Delay_Queue.
        --------
        if  the Rep_Queue has only one entry  then

            Add a_ss.Replenish_Task to the Delay_Queue to wake up at Pending_Replenishment.Rep_Time;

        end if;

    end if;

end Enqueue_Pending_Replenishment;
```

Figure 5-8: Queue_Pending_Replenishment

5.2.6.4 Replenishing Sporadic Servers

The procedure Replenish_Sporadic_Server (presented in Figure 5-9) 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 the 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 5.2.6.2. 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.

```
procedure Replenish_Sporadic_Server  (a_ss : access sporadic_server) is

    This_Rep : Replenishment;

begin

    --------
    --  Before this replenishment is made, if the sporadic server's available
    --  execution time is zero, then the Rep_Origin should be set to the
    --  current time.
    --------
    if a_ss.Avail_Exec_Time := 0.0 then

        a_ss.Replenish_Data.Rep_Origin := Clock;

    end if;

    --------
    --  Dequeue the next replenishment from the sporadic server's Rep_Queue.
    --------
    This_Rep := Dequeue_Replenishment(a_ss);

    --------
    --  Increment the sporadic server's available execution time.
    --------
    a_ss.Avail_Exec_Time := a_ss.Avail_Exec_Time + This_Rep.Rep_Amount;

    --------
    --  If the Rep_Queue is not empty, place the Replenish_Task back on the
    --  Delay Queue to wake up at the Rep_Time of the replenishment at the
    --  head of the Rep_Queue.
    --------
    if the Rep_Queue is not empty then


        Add a_ss.Replenish_Task to the Delay Queue to wake up at the Rep_Time at the head of the Rep_Queue;

    end if;

    --------
    --  If the Rep_Queue was just previously full and the sporadic server's
    --  pending replenishment has a positive Rep_Amount, then add the pending
    --  replenishment to the Rep_Queue and reset the Rep_Amount to zero.
    --------
    if the Rep_Queue only has one free slot then

        Queue_Pending_Replenishment(a_ss);

    end if;

    --------
    --  Since some of the sporadic server's execution time has been
    --  replenished, check if the sporadic server should be placed on the
    --  SS_Ready_Queue.
    --------
    if not a_ss.On_SS_Ready_Queue and then a_ss.Aperiodic_Ready_Queue /= null then

        Add the sporadic server to the SS_Ready_Queue;

    end if;

end Replenish_Sporadic_Server;
```

Figure 5-9: Replenish_Sporadic_Server

5.3 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 5.2.

5.3.1 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?

5.3.2 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.

5.3.3 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.

5.3.4 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 5.2.6.2). 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 earliest replenishment schedule as shown in [Sprunt 89b], 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 2-1). The drawback of this simple replenishment policy is that some replenishments do not occur as early as possible (the aperiodic response time advantage of the full policy over the simple policy is discussed Section 3.5).

Using this simple replenishment policy allows the elimination of the procedure Track_Active_Idle_Status (Section 5.2.6.2). Replenishments must then be scheduled as each portion of sporadic server execution time is consumed. The procedure Mark_SS_Consumption (Section 5.2.6.1) can be modified to schedule replenishment for sporadic server execution time as it is consumed.

5.3.5 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 5-4 and 5-9).

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.

5.4 Implementing Sporadic Servers on the Futurebus+

This section presents a high-level description of an implementation method for supporting sporadic servers on the Futurebus+ [FutureBus 89]. The Futurebus+ specification is a standard interface for a high-performance bus for next generation commercial and military systems. The sporadic server implementation described herein is for a shared, high-priority sporadic server that can provide quick responsiveness for soft-deadline, asynchronous bus transfers.

5.4.1 Scheduling Real-Time Data Transfers on the Futurebus+

The scheduling goals for data transfers between modules in a real-time system are very similar to scheduling goals for real-time tasks that share a processor. Like most real-time tasks, data transfers in a real-time system fall into two basic categories based upon their arrival pattern. Synchronous transfers have regular interarrival times and asynchronous transfers have irregular arrival times. Also like real-time tasks, synchronous data transfers typically have hard deadlines and asynchronous transfers can have either soft or hard deadlines. However, scheduling data transfers to meet hard deadlines on a hardware bus is typically much more difficult than the scheduling real-time tasks to meet hard deadlines on a processor. The key difference between software scheduling and hardware scheduling resides in the mechanisms available to implement a priority-based arbitration for system resources.

Arbitration for processor access by software tasks is typically done in software by the operating system (or the runtime system as is the case for Ada). This allows a priority-based arbitration scheme to be implemented for the scheduling of real-time tasks in which the highest priority task is always selected for execution, preempting any lower priority tasks as necessary.

Arbitration for hardware access by a processor is typically very different from priority-based scheduling for software tasks. The key difference is that most hardware based arbitration schemes determine the priority of a module independently of the software task(s) for which the module provides service. Many different priority schemes exist for backplane buses. Common examples are round-robin and daisy chain arbitration schemes, priorities based upon the module’s position on the bus, and switch-selectable priority on the module itself. Note that none of these schemes allow the priority of a hardware device to be set dynamically based upon the priority of the software task which is using the device. This can cause a scheduling problem known as *priority inversion*. A priority inversion occurs when a low priority task blocks the execution of a higher priority task. Priority inversion has a detrimental effect upon the schedulability of a real-time system because hard-deadline tasks must be able to tolerate the maximum amount of time they can be blocked by lower priority tasks.

Priority inversion can easily occur when a consistent mapping between software task priorities and hardware device priorities is not possible. Consider, for example, the case of a DMA controller that has a fixed, high-priority for accessing the system’s memory bus. If a low-priority task initiates a DMA transfer and a high-priority task then becomes ready to be executed, the DMA controller can prevent the high-priority task from accessing memory. This is a priority inversion because the DMA controller is actually performing a service for the low priority task. If the DMA transfer is long and/or the deadline is short for the high-priority task, the high-priority task may miss its deadline.

The Futurebus+ provides a priority-based arbitration scheme that can overcome many of the priority inversion problems associated with most hardware arbitration schemes. The Futurebus+ implements a priority-based arbitration scheme with 256 priority levels for bus access. The key feature of this arbitration scheme is that the priority of a module is specified by the module during the arbitration process. This allows the module to dynamically set its priority based upon the priority of the software task for which it is providing service. This allows a uniform priority scheme to be implemented for both software tasks and hardware devices which enables the real-time system designer to eliminate many sources of priority inversion.

5.4.2 A Shared Sporadic Server Implementation for the Futurebus+

A shared sporadic server can be implemented using the Futurebus+ to provide quick responsiveness for asynchronous, soft-deadline data transfers. The capacity of the sporadic server is shared in the sense that any module on the bus can use the sporadic server.

The key design issues for creating a shared sporadic server are the efficient and distributed management of the sporadic server’s capacity. First, no module should have to gain bus access in order to determine if the sporadic server has available capacity. Otherwise, this could cause many bus arbitration cycles and bus accesses for which no real work is accomplished. These "extra" bus cycles could also delay other bus transfers causing another form of priority inversion. Second, the bus operations to decrement and replenish the available sporadic server capacity must update each module’s view of the sporadic server’s capacity with as few bus cycles as possible.

Unnecessary bus transactions to determine the available capacity of a sporadic server can be avoided by keeping a copy of the sporadic server’s available capacity in local memory on each module using the sporadic server. If a module wants to use the sporadic server for an asynchronous data transfer, the module checks its local sporadic capacity table. If enough capacity is available to complete the data transfer, the module then participates in the next bus arbitration cycle using the priority of the sporadic server. Once bus access is gained, the module performs the data transfer.

The efficient and distributed management of sporadic server capacity can be accomplished by using the broadcast write capability of the Futurebus+. Whenever a sporadic server’s capacity must be updated, a broadcast write of the updated capacity value is done. Since each module will process the broadcast write, the overhead for updating the sporadic server capacity tables is independent of the number of modules which can use the sporadic server.

The management of sporadic server capacity is the responsibility of the module that consumes the capacity. To update the sporadic server capacity tables after some capacity has been consumed, the module first determines the new capacity by decrementing the available capacity by the amount consumed during the data transfer and then performs a broadcast write of the new value. This broadcast is performed after the data transfer is made and before the module relinquishes the bus. To update the sporadic server capacity at the time a replenishment is due, the module arbitrates for bus access using the priority of the sporadic server. Once bus access is gained, the module determines the new capacity value by adding the amount consumed to the available capacity and then performs a broadcast write of the new capacity value.

The list below summarizes the above discussion by listing the sequence of steps necessary to use a sporadic server on the Futurebus+:

1. Upon determining that an asynchronous data transfer is ready to be made, the module checks its local sporadic server capacity table to determine if sufficient sporadic server capacity is available to complete the data transfer. Note that this does not require bus access.
2. If sufficient capacity is available, the module participates in the next arbitration cycle for the bus using the sporadic server’s priority.

3. If the module wins the arbitration cycle, it performs the data transfer. Otherwise, go to step 1.

4. After the data transfer is completed, the module determines the new value for the sporadic server's capacity and performs a broadcast write to update each module with the new sporadic server capacity. The module then relinquishes the bus.

5. At the replenishment time for the consumed sporadic server capacity, the module determines the new sporadic server capacity and participates in the next bus arbitration cycle using the sporadic server's priority. Once bus access is obtained, a broadcast write is performed to update each module with the new sporadic server capacity.

6. Conclusions

In this final chapter we summarize the results and research contributions of this thesis and suggest directions for future research.

6.1 Summary of Results

In the previous chapters, we have studied the problem of scheduling aperiodic tasks in hard real-time systems. In order to guarantee hard timing constraints, it is necessary that the scheduling solution provide predictable response time performance for tasks with hard deadlines. However, aperiodic tasks by definition have unpredictable arrival times. This unpredictable nature of aperiodic tasks makes it difficult to create a predictable scheduling solution that is able to meet hard timing constraints. It is also difficult to provide quick responsiveness for soft-deadline aperiodic tasks while not endangering the deadlines of hard-deadline tasks. We showed that the typical aperiodic task scheduling approaches of background or polling service offer poor responsiveness for soft-deadline aperiodic tasks and are often unable to guarantee deadlines for sporadic tasks (i.e., hard-deadline aperiodic tasks). Also, although the Deferrable Server (DS) and Priority Exchange (PE) algorithms have been shown to provide substantially better responsiveness for soft-deadline aperiodic tasks and improved support for sporadic tasks, the DS algorithm suffers a schedulability penalty and the PE algorithm has a prohibitively complex implementation. In addition to these limitations, the problem of scheduling short-deadline sporadic tasks is not addressed by any of these algorithms. This thesis develops the Sporadic Server (SS) algorithm that overcomes these limitations of the DS and PE algorithms while maintaining their advantages.

Chapter 2 begins with the definition of the SS algorithm. The SS algorithm is similar to the DS algorithm in that it maintains its execution time at a high priority level until needed by an aperiodic task. This allows the SS algorithm to provide good aperiodic responsiveness and avoids the costly implementation overhead of the PE algorithm that must manage execution time across all priority levels in the system. The key difference between the DS and SS algorithms lies in their replenishment policies for consumed execution time. The SS algorithm schedules the replenishment of consumed execution time based upon when the execution time is consumed unlike the DS algorithm that replenishes execution time independently of when it is consumed. It is this difference that allows the SS algorithm to have a larger server size than the DS algorithm. Several examples of the use of the SS algorithm for soft-deadline aperiodic tasks are presented.

Next, Chapter 2 discusses the SS algorithm’s performance, schedulability, and operation for sporadic tasks. The expected performance improvement of the SS algorithm over polling service is explained. Although it is not the case that the SS algorithm can always provide better responsiveness for aperiodic tasks than polling, we do prove that the SS algorithm always provides high priority aperiodic service at times equal to or earlier than a polling server. We then argue that the typical case is the one in which the sporadic server provides earlier aperiodic service than polling, resulting in a better average response time. Next, we prove that a sporadic server, unlike a deferrable server, can be treated as an equivalently-sized periodic task for schedulability analysis. To provide a guarantee for a sporadic task, a sporadic server is created to exclusively service the sporadic task. For sporadic tasks with deadlines that are shorter than their minimum interarrival time, it is necessary to use a deadline monotonic priority assignment rather than a rate monotonic priority assignment. To schedule sporadic servers with a deadline monotonic priority assignment, it is shown that the execution of the sporadic tasks can be treated as blocking time in the schedulability analysis.

Chapter 2 closes with a discussion of the interaction of the SS algorithm and the priority inheritance protocols. The priority inheritance protocols regulate the sharing of data among tasks in a real-time system to avoid the problems of unbounded priority inversion and low schedulable utilization. A discussion of the implementation costs is presented. The recommended policy is to not allow tasks that inherit the priority of a sporadic server to consume the sporadic server's execution time. Since the high priority execution time of a sporadic server may be exhausted during the execution of a critical section that protects shared data, an aperiodic task using a sporadic server may both preempt lower priority tasks and block higher priority tasks during the critical section. Thus, a greater schedulability penalty must be incurred when aperiodic tasks using sporadic servers share data with other tasks. Therefore, the recommendation is made to disallow the possibility of exhausting SS execution time during a critical section (options for doing so are discussed in Chapter 5).

Chapter 3 presents a simulation study of the Background, Polling, Deferrable Server, Priority Exchange, and Sporadic Server algorithms for a wide range of periodic and aperiodic loads. In all of the cases examined, the SS algorithm provides a substantially better average aperiodic response time than background or polling service. For some cases, the SS algorithm is able to provide an order of magnitude improvement in average aperiodic response time over polling service. Also, the SS algorithm provides an average aperiodic response time comparable to or better than the average response time of the DS and PE algorithms. For the cases where the SS algorithm has a significant server size advantage over the DS algorithm, the SS algorithm is able to provide better aperiodic responsiveness for high aperiodic loads than the DS algorithm. The SS algorithm was also shown to provide an aperiodic response time variance better than background and polling and comparable to the DS and PE algorithms.

Next, the context swap overhead for these aperiodic service algorithms is compared and a new service order policy to reduce context swap overhead is presented. The context swap overhead of the PE, DS, and SS algorithms is shown to be higher than that of the background and polling algorithms due to the more responsive aperiodic service provided by these algorithms. The context swap overhead of the SS algorithm was shown to be comparable to the DS for aperiodic loads where sporadic server overruns occur infrequently. However, for higher aperiodic loads, the context swap overhead of the SS algorithm becomes higher than the DS algorithm due to the "spreading out" of the SS's execution time caused by its replenishment policy. In these cases, the SS algorithm does not always show a performance benefit (i.e., a better response time) for the larger overhead. A new service order policy is then defined which grants service to an aperiodic task only if the aperiodic task can be serviced to completion before the server is exhausted. The Complete Service (CS) policy is shown to improve the average aperiodic response time in many cases and to provide a context swap overhead equal to or less than the original policy in all cases. Under the CS policy, for every case where the overhead for the SS algorithm is larger than the overhead for the DS algorithm, the SS algorithm also provides a significantly better average aperiodic response time than the DS algorithm.

Chapter 3 then presents a performance comparison of the full SS replenishment policy with the simple SS replenishment policy that requires less implementation overhead than the full replenishment policy but may degrade performance. The results indicate that, in most cases, the performance improvement associated with the full replenishment policy does not compensate for the overhead required to implement the full replenishment policy. The only cases where the advantage of the full replenishment policy makes a significant difference in response time performance are cases of high aperiodic load supported by a sporadic server executing at a medium to low priority level. Therefore, implementations can usually use the simpler replenishment policy without sacrificing response time performance.

Chapter 3 closes with a study of the effect of sporadic server period, execution time, and priority upon average aperiodic response time and establishes several guidelines for choosing these parameters of sporadic servers. The simulation results indicate that, for a given server size and server priority, a larger server execution time and period reduces the average aperiodic response time even though the larger server period increases the replenishment delay of the server. Simulation results also indicate that, for cases where a lower priority allows a larger server size, the larger server size may provide a reduction in average response time if the mean aperiodic service time is a significant percentage of the server's execution time and the aperiodic load is high. So, the guidelines suggest the selection of a high priority server with a large server period to yield both a large server execution time and server size. However, for task sets where a lower server priority allows a significant increase in the server's size and both the mean aperiodic service time is large and the aperiodic load is high, a lower server priority with a larger size may provide a better average aperiodic response time.

Chapter 4 presents models for predicting the average aperiodic response time performance for the background, DS, and SS aperiodic service algorithms. For background aperiodic service, we found that modelling the deterministic nature of the periodic tasks as a random process with Poisson arrivals and exponential service times or modelling the random nature of the aperiodic tasks with a deterministic periodic task produced poor predictions for the average aperiodic response time. A model which incorporates both the deterministic nature of the periodic tasks and the random nature of the aperiodic tasks was shown to produce good predictions of the average response time for background service. However, this model breaks down for cases where the aperiodic work accumulated during periodic task execution cannot be completed before the next execution of the periodic tasks begins. We did find that this model provides a good prediction of aperiodic response time if the average length of the aperiodic queue at the beginning of each interval of periodic task execution can be predicted. For the DS and SS algorithms, Chapter 4 presents a model for aperiodic task execution and derives equations that allow the computation of the range of aperiodic load over which a standard M/M/1 or M/D/1 queueing theory model will provide a good prediction of average aperiodic response time.

Chapter 5 investigates software and hardware implementations of sporadic servers. For software implementations, two methods are considered for real-time systems programmed in Ada. The first implementation method requires no modifications to the Ada runtime and implements a sporadic server as an application-level Ada task. Because an application-level task cannot accurately track the execution time it consumes, this implementation uses the simple SS replenishment policy and assumes that the worst-case execution time of an aperiodic task is always consumed. The basic mechanism of this implementation is the Ada selective wait with a delay alternative. The accept statements with guards are used to grant or deny sporadic service and the delay alternative is used to replenish consumed sporadic server execution time. The second implementation method supports the full SS replenishment policy but requires modifications to the Ada runtime. For a hardware implementation, a method to implement sporadic servers for the Futurebus+ is described that takes advantage of bus' priority-based arbitration scheme to provide high-priority bus access and the bus' broadcast write capability to provide global management of the server's capacity.

6.2 Research Contributions

This thesis focussed upon an important and difficult problem in real-time systems: providing responsive service for soft-deadline aperiodic tasks and guaranteed deadlines for hard-deadline aperiodic tasks while not endangering guarantees for hard-deadline periodic tasks. Although previous work in this area produced promising results, these solutions suffered from schedulability penalties or high implementation complexity. Also, previous work in this area did not address the problem of scheduling hard-deadline aperiodic tasks with short deadlines or predicting the average response time performance for aperiodic tasks.

The research reported in this thesis was carried out in four phases: algorithm development, performance evaluation, performance prediction, and implementation. A number of significant research contributions were presented in this thesis. These contributions can be summarized as follows:

- The SS algorithm overcomes the schedulability penalty of the DS algorithm and the implementation complexity of the PE algorithm while providing a responsiveness for soft-deadline aperiodic tasks that is comparable to or better than these previous aperiodic service algorithms.
- The SS algorithm provides a uniform scheduling solution for both soft-deadline and hard-deadline aperiodic tasks.
- An improved schedulability analysis was developed that can be used with the SS algorithm to schedule hard-deadline aperiodic tasks with deadlines that are shorter than their minimum interarrival time.
- The interaction of the SS algorithm and the priority inheritance protocols and the associated schedulability analysis has been defined, providing a scheduling solution for aperiodic and periodic tasks that share data.
- Sporadic servers have been proven to be treatable as an equivalently-sized periodic task in the schedulability analysis for a real-time system scheduled using the rate monotonic algorithm.
- Since sporadic servers can be treated as periodic tasks and can be used with the priority inheritance protocols, all of the scheduling theories and tools developed for periodic tasks scheduled with the rate monotonic algorithm can now also be applied to the problem of scheduling aperiodic tasks. This also implies that we now have a scheduling solution for real-time systems composed primarily of aperiodic tasks.
- Guidelines have been developed for setting the priority, execution time, and period of sporadic servers to provide good aperiodic responsiveness. Although these guidelines do not cover all cases, they identify the significant performance considerations and tradeoffs associated with the design of a sporadic server.

- Models and equations have been developed that allow the use of standard queueing theory models to predict the average response time performance of both the DS and SS algorithms.
- Implementation methods have been developed to support the SS algorithm in both software and hardware. Two implementation strategies were developed to allow the integration of the SS algorithm into Ada, the Department of Defense's recommended language for designing real-time systems. An implementation strategy has been developed for the SS algorithm on the Futurebus+, a standard bus for the next generation commercial and military systems.

6.3 Directions for Future Research

The research work presented in this thesis can be extended in a number of areas to improve response time performance, increase the range of performance prediction, use the SS algorithm to solve other real-time scheduling problems, and provide support for fault-tolerant systems. We describe briefly each of these areas below.

The execution time, period, and priority of the sporadic servers described in this thesis are determined from a worst-case schedulability analysis in which all hard-deadline tasks request service at the same instant in time. This approach guarantees that the resulting sporadic server will never cause a hard deadline to be missed. However, this assumption is probably too pessimistic and the worst-case phasing in typical systems may allow larger aperiodic servers to be created, improving their ability to provide quick responsiveness for soft-deadline aperiodic tasks. Further research is needed to develop a schedulability analysis approach that considers the actual worst-case phasing of a given task set and to establish the potential gain of this technique.

In many real-time systems, transient overloads may 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 [Sha 86]. 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. One should be able to use the sporadic server to implement this 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. Further work is necessary to investigate the schedulability effects of this approach to establish the limits for which this approach is feasible, given that a period transformation increases the context swap overhead for the task set.

In a real-time system, a typical Producer/Consumer scheduling problem occurs when a device can produce data at a burst rate that is faster than the average rate at which the data can be consumed. Since the burst production rate is greater than the average consumption rate, the data are placed in a queue as they are produced. The consumer removes data from the queue as soon as possible. Two sporadic servers can be used to schedule the producer and consumer tasks. The producer sporadic server would be given a deadline monotonic priority assignment in order to meet the short timing constraints of the maximum burst rate. The consumer sporadic server would be given a rate monotonic priority assignment consistent with the average rate of data production. With the proper characterization of the data production rate, it should be possible to perform a schedulability analysis that bounds the maximum queue length and predicts the average and worst-case time to consume an item.

The models developed in this thesis to predict aperiodic response time for soft-deadline aperiodic tasks are limited. Further research is needed to extend the predictive range of these models to higher aperiodic loads and to cover the cases where a sporadic server executes at a priority level lower than the highest priority level and where multiple sporadic servers are executing at different priority levels.

Sporadic servers could also be used to provide support for fault detection and containment in real-time systems. For example, 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 is to check if the task has not 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, a fault has been detected. Since the sporadic server will suspend the execution of the periodic task once all of its execution time has been consumed, the errors resulting from any fault that has caused the periodic task to execute longer than it should are contained to the faulty task and not allowed to influence the execution of other tasks.

Many fault-tolerant systems use physical redundancy to recover from faults. The SS algorithm could be used to provide temporal redundancy to recover from transient faults in real-time systems. In such a system, the sporadic server would be created to provide immediate resource utilization for the partial or full re-execution of tasks that have faulted. This should improve the level of fault recovery, but fault models and schedulability analysis techniques need to be developed to verify the potential improvement of this scheme and establish its range of applicability.

I. Task Set and Aperiodic Server Parameters for Average Aperiodic Response Time Experiments

Task Set 0:

Periodic Load 40%

ID	Period	Exec Time	Phase	Util
1	55.0000	3.5579	25.1700	0.0647
2	66.0000	3.2917	43.6500	0.0499
3	77.0000	0.3799	62.8700	0.0049
4	85.5556	6.3983	3.8500	0.0748
5	105.0000	6.4946	14.5200	0.0619
6	115.5000	2.8009	72.1700	0.0243
7	154.0000	3.6195	103.3000	0.0235
8	177.6923	11.1649	133.0500	0.0628
9	330.0000	1.0750	164.2000	0.0033
10	385.0000	11.5651	220.7400	0.0300

Aperiodic Server Period = 55.00

Server Execution Time:	**30.69**	**Polling, PE, SS**
	21.57	**DS**

Periodic Load 60%

ID	Period	Exec Time	Phase	Util
1	55.0000	5.3369	25.1700	0.0970
2	66.0000	4.9375	43.6500	0.0748
3	77.0000	0.5699	62.8700	0.0074
4	85.5556	9.5974	3.8500	0.1122
5	105.0000	9.7419	14.5200	0.0928
6	115.5000	4.2014	72.1700	0.0364
7	154.0000	5.4293	103.3000	0.0353
8	177.6923	16.7474	133.0500	0.0942
9	330.0000	1.6125	164.2000	0.0049
10	385.0000	17.3477	220.7400	0.0451

Aperiodic Server Period = 55.00

Server Execution Time:	**18.56**	**Polling, PE, SS**
	14.38	**DS**

Periodic Load 80%

ID	Period	Exec Time	Phase	Util
1	55.0000	7.1159	25.1700	0.1294
2	66.0000	6.5834	43.6500	0.0997
3	77.0000	0.7599	62.8700	0.0099
4	85.5556	12.7966	3.8500	0.1496
5	105.0000	12.9892	14.5200	0.1237
6	115.5000	5.6018	72.1700	0.0485
7	154.0000	7.2391	103.3000	0.0470
8	177.6923	22.3298	133.0500	0.1257
9	330.0000	2.1500	164.2000	0.0065
10	385.0000	23.1303	220.7400	0.0601

Aperiodic Server Period = 55.00

Server Execution Time:	**6.43**	**Polling, PE, SS**
	5.47	**DS**

```
Task Set 1:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         1.5917        52.9600        0.0289
       2        67.9412         2.1209        60.3300        0.0312
       3        72.1875         1.5818        22.6500        0.0219
       4        82.5000         3.5458        46.8400        0.0430
       5        88.8462         1.8444        14.3800        0.0208
       6       105.0000         5.5961        84.9400        0.0533
       7       288.7500        17.1078       119.9300        0.0592
       8       385.0000        11.7605       275.7500        0.0305
       9       462.0000        23.6036       442.9900        0.0511
      10       577.5000        34.6648       106.7800        0.0600

      Aperiodic Server Period = 55.00

      Server Execution Time:    29.08   Polling, PE, SS
                                23.07   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         2.3876        52.9600        0.0434
       2        67.9412         3.1813        60.3300        0.0468
       3        72.1875         2.3728        22.6500        0.0329
       4        82.5000         5.3187        46.8400        0.0645
       5        88.8462         2.7666        14.3800        0.0311
       6       105.0000         8.3942        84.9400        0.0799
       7       288.7500        25.6618       119.9300        0.0889
       8       385.0000        17.6408       275.7500        0.0458
       9       462.0000        35.4053       442.9900        0.0766
      10       577.5000        51.9972       106.7800        0.0900

      Aperiodic Server Period = 55.00

      Server Execution Time:    16.95   Polling, PE, SS
                                15.56   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         3.1835        52.9600        0.0579
       2        67.9412         4.2418        60.3300        0.0624
       3        72.1875         3.1637        22.6500        0.0438
       4        82.5000         7.0916        46.8400        0.0860
       5        88.8462         3.6888        14.3800        0.0415
       6       105.0000        11.1923        84.9400        0.1066
       7       288.7500        34.2157       119.9300        0.1185
       8       385.0000        23.5211       275.7500        0.0611
       9       462.0000        47.2071       442.9900        0.1022
      10       577.5000        69.3296       106.7800        0.1201

      Aperiodic Server Period = 55.00

      Server Execution Time:    5.15    Polling, PE, SS
                                4.72    DS
```

```
Task Set 2:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         1.9904        12.8800        0.0362
       2        66.0000         1.8613         5.1200        0.0282
       3        67.9412         1.0073        52.4300        0.0148
       4        85.5556         2.9941         1.9200        0.0350
       5        92.4000         2.9689        13.3800        0.0321
       6       135.8824         8.0578       129.2900        0.0593
       7       144.3750        10.2616        68.9000        0.0711
       8       154.0000         5.0885        18.8300        0.0330
       9       177.6923         9.9142       176.0800        0.0558
      10      1155.0000        39.7645       370.2400        0.0344

      Aperiodic Server Period = 55.00

      Server Execution Time:    27.47   Polling, PE, SS
                                20.38   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         2.9855        12.8800        0.0543
       2        66.0000         2.7919         5.1200        0.0423
       3        67.9412         1.5110        52.4300        0.0222
       4        85.5556         4.4911         1.9200        0.0525
       5        92.4000         4.4533        13.3800        0.0482
       6       135.8824        12.0866       129.2900        0.0889
       7       144.3750        15.3924        68.9000        0.1066
       8       154.0000         7.6327        18.8300        0.0496
       9       177.6923        14.8713       176.0800        0.0837
      10      1155.0000        59.6467       370.2400        0.0516

      Aperiodic Server Period = 55.00

      Server Execution Time:    15.88   Polling, PE, SS
                                11.91   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         3.9807        12.8800        0.0724
       2        66.0000         3.7226         5.1200        0.0564
       3        67.9412         2.0146        52.4300        0.0297
       4        85.5556         5.9882         1.9200        0.0700
       5        92.4000         5.9377        13.3800        0.0643
       6       135.8824        16.1155       129.2900        0.1186
       7       144.3750        20.5233        68.9000        0.1422
       8       154.0000        10.1769        18.8300        0.0661
       9       177.6923        19.8283       176.0800        0.1116
      10      1155.0000        79.5289       370.2400        0.0689

      Aperiodic Server Period = 55.00

      Server Execution Time:    6.11    Polling, PE, SS
                                4.50    DS
```

Task Set 3:

Periodic Load 40%

ID	Period	Exec Time	Phase	Util
1	55.0000	3.9591	21.0400	0.0720
2	66.0000	2.0096	50.1000	0.0304
3	70.0000	3.4862	56.2100	0.0498
4	92.4000	3.1100	16.7700	0.0337
5	100.4348	4.0949	46.3500	0.0408
6	105.0000	2.6210	39.1900	0.0250
7	110.0000	5.4548	22.7500	0.0496
8	121.5789	7.3546	104.3600	0.0605
9	165.0000	6.1673	53.4900	0.0374
10	210.0000	0.1916	118.5000	0.0009

Aperiodic Server Period = 55.00

Server Execution Time:	28.33	Polling, PE, SS
	19.53	DS

Periodic Load 60%

ID	Period	Exec Time	Phase	Util
1	55.0000	5.9387	21.0400	0.1080
2	66.0000	3.0144	50.1000	0.0457
3	70.0000	5.2293	56.2100	0.0747
4	92.4000	4.6650	16.7700	0.0505
5	100.4348	6.1424	46.3500	0.0612
6	105.0000	3.9314	39.1900	0.0374
7	110.0000	8.1821	22.7500	0.0744
8	121.5789	11.0319	104.3600	0.0907
9	165.0000	9.2509	53.4900	0.0561
10	210.0000	0.2873	118.5000	0.0014

Aperiodic Server Period = 55.00

Server Execution Time:	15.13	Polling, PE, SS
	11.16	DS

Periodic Load 80%

ID	Period	Exec Time	Phase	Util
1	55.0000	7.9183	21.0400	0.1440
2	66.0000	4.0192	50.1000	0.0609
3	70.0000	6.9725	56.2100	0.0996
4	92.4000	6.2199	16.7700	0.0673
5	100.4348	8.1898	46.3500	0.0815
6	105.0000	5.2419	39.1900	0.0499
7	110.0000	10.9095	22.7500	0.0992
8	121.5789	14.7092	104.3600	0.1210
9	165.0000	12.3346	53.4900	0.0748
10	210.0000	0.3831	118.5000	0.0018

Aperiodic Server Period = 55.00

Server Execution Time:	1.82	Polling, PE, SS
	1.39	DS

```
Task Set 4:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         4.2689        16.2800       0.0776
       2        62.4324         5.5505         6.4600       0.0889
       3        79.6552         5.1671        12.1400       0.0649
       4        85.5556         3.1302        31.3100       0.0366
       5       121.5789         1.1091        41.9500       0.0091
       6       128.3333         0.5739        80.3000       0.0045
       7       135.8824         9.5392       130.0800       0.0702
       8       154.0000         2.0960        98.9500       0.0136
       9       256.6667         1.3284        34.1000       0.0052
      10       462.0000        13.6115        42.7400       0.0295

      Aperiodic Server Period = 55.00

      Server Execution Time:    30.15   Polling, PE, SS
                                18.45   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         6.4033        16.2800       0.1164
       2        62.4324         8.3258         6.4600       0.1334
       3        79.6552         7.7506        12.1400       0.0973
       4        85.5556         4.6953        31.3100       0.0549
       5       121.5789         1.6636        41.9500       0.0137
       6       128.3333         0.8609        80.3000       0.0067
       7       135.8824        14.3088       130.0800       0.1053
       8       154.0000         3.1440        98.9500       0.0204
       9       256.6667         1.9925        34.1000       0.0078
      10       462.0000        20.4172        42.7400       0.0442

      Aperiodic Server Period = 55.00

      Server Execution Time:    17.81   Polling, PE, SS
                                12.01   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         8.5377        16.2800       0.1552
       2        62.4324        11.1010         6.4600       0.1778
       3        79.6552        10.3342        12.1400       0.1297
       4        85.5556         6.2604        31.3100       0.0732
       5       121.5789         2.2181        41.9500       0.0182
       6       128.3333         1.1478        80.3000       0.0089
       7       135.8824        19.0784       130.0800       0.1404
       8       154.0000         4.1919        98.9500       0.0272
       9       256.6667         2.6567        34.1000       0.0104
      10       462.0000        27.2230        42.7400       0.0589

      Aperiodic Server Period = 55.00

      Server Execution Time:    5.47    Polling, PE, SS
                                3.75    DS
```

```
Task Set 5:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         1.3273         2.6900       0.0241
       2        66.0000         0.5518        50.2700       0.0084
       3        72.1875         6.1973        23.0800       0.0859
       4       110.0000         6.6921        75.2100       0.0608
       5       128.3333         5.9071       103.4200       0.0460
       6       135.8824         3.2174       132.6800       0.0237
       7       177.6923         0.7400       150.0100       0.0042
       8       231.0000        10.3348        50.5500       0.0447
       9       385.0000        21.1341        16.1500       0.0549
      10       770.0000        36.4258       292.9300       0.0473

      Aperiodic Server Period = 55.00

      Server Execution Time:    32.19   Polling, PE, SS
                                23.39   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         1.9909         2.6900       0.0362
       2        66.0000         0.8278        50.2700       0.0125
       3        72.1875         9.2960        23.0800       0.1288
       4       110.0000        10.0381        75.2100       0.0913
       5       128.3333         8.8606       103.4200       0.0690
       6       135.8824         4.8262       132.6800       0.0355
       7       177.6923         1.1100       150.0100       0.0062
       8       231.0000        15.5022        50.5500       0.0671
       9       385.0000        31.7012        16.1500       0.0823
      10       770.0000        54.6387       292.9300       0.0710

      Aperiodic Server Period = 55.00

      Server Execution Time:    20.81   Polling, PE, SS
                                19.42   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         2.6545         2.6900       0.0483
       2        66.0000         1.1037        50.2700       0.0167
       3        72.1875        12.3946        23.0800       0.1717
       4       110.0000        13.3842        75.2100       0.1217
       5       128.3333        11.8142       103.4200       0.0921
       6       135.8824         6.4349       132.6800       0.0474
       7       177.6923         1.4801       150.0100       0.0083
       8       231.0000        20.6696        50.5500       0.0895
       9       385.0000        42.2682        16.1500       0.1098
      10       770.0000        72.8517       292.9300       0.0946

      Aperiodic Server Period = 55.00

      Server Execution Time:    9.44    Polling, PE, SS
                                8.79    DS
```

```
Task Set 6:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         3.7382        15.4800       0.0680
       2        59.2308         2.1456        44.7900       0.0362
       3        72.1875         0.0844        22.1300       0.0012
       4        77.0000         4.0308         2.7900       0.0523
       5       105.0000         3.9773        48.0400       0.0379
       6       110.0000         5.8033         2.9700       0.0528
       7       121.5789         5.1908        73.6000       0.0427
       8       144.3750         6.8417       127.1000       0.0474
       9       154.0000         1.5651       149.6300       0.0102
      10       288.7500        14.8509        27.0800       0.0514

      Aperiodic Server Period = 55.00

      Server Execution Time:    30.47   Polling, PE, SS
                                20.81   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         5.6073        15.4800       0.1020
       2        59.2308         3.2184        44.7900       0.0543
       3        72.1875         0.1266        22.1300       0.0018
       4        77.0000         6.0462         2.7900       0.0785
       5       105.0000         5.9660        48.0400       0.0568
       6       110.0000         8.7049         2.9700       0.0791
       7       121.5789         7.7862        73.6000       0.0640
       8       144.3750        10.2625       127.1000       0.0711
       9       154.0000         2.3476       149.6300       0.0152
      10       288.7500        22.2764        27.0800       0.0771

      Aperiodic Server Period = 55.00

      Server Execution Time:    18.24   Polling, PE, SS
                                13.30   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----         ------
       1        55.0000         7.4763        15.4800       0.1359
       2        59.2308         4.2911        44.7900       0.0724
       3        72.1875         0.1687        22.1300       0.0023
       4        77.0000         8.0616         2.7900       0.1047
       5       105.0000         7.9546        48.0400       0.0758
       6       110.0000        11.6065         2.9700       0.1055
       7       121.5789        10.3816        73.6000       0.0854
       8       144.3750        13.6833       127.1000       0.0948
       9       154.0000         3.1302       149.6300       0.0203
      10       288.7500        29.7019        27.0800       0.1029

      Aperiodic Server Period = 55.00

      Server Execution Time:    6.00    Polling, PE, SS
                                5.15    DS
```

```
Task Set 7:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         2.8728        52.1500       0.0522
       2        59.2308         3.0524        24.4600       0.0515
       3        67.9412         0.8487        38.3000       0.0125
       4        96.2500         6.0479        67.2600       0.0628
       5       100.4348         5.3117        37.5200       0.0529
       6       154.0000         0.6390       125.7200       0.0041
       7       165.0000         3.0367        58.1400       0.0184
       8       256.6667         4.0252       249.2200       0.0157
       9       385.0000        20.8082       215.9600       0.0540
      10      2310.0000       174.9493       318.6900       0.0757

      Aperiodic Server Period = 55.00

      Server Execution Time:    32.94   Polling, PE, SS
                                23.71   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         4.3091        52.1500       0.0783
       2        59.2308         4.5786        24.4600       0.0773
       3        67.9412         1.2730        38.3000       0.0187
       4        96.2500         9.0719        67.2600       0.0943
       5       100.4348         7.9676        37.5200       0.0793
       6       154.0000         0.9586       125.7200       0.0062
       7       165.0000         4.5551        58.1400       0.0276
       8       256.6667         6.0378       249.2200       0.0235
       9       385.0000        31.2123       215.9600       0.0811
      10      2310.0000       262.4240       318.6900       0.1136

      Aperiodic Server Period = 55.00

      Server Execution Time:    21.99   Polling, PE, SS
                                19.63   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         5.7455        52.1500       0.1045
       2        59.2308         6.1047        24.4600       0.1031
       3        67.9412         1.6974        38.3000       0.0250
       4        96.2500        12.0959        67.2600       0.1257
       5       100.4348        10.6234        37.5200       0.1058
       6       154.0000         1.2781       125.7200       0.0083
       7       165.0000         6.0734        58.1400       0.0368
       8       256.6667         8.0505       249.2200       0.0314
       9       385.0000        41.6164       215.9600       0.1081
      10      2310.0000       349.8986       318.6900       0.1515

      Aperiodic Server Period = 55.00

      Server Execution Time:    10.94   Polling, PE, SS
                                10.73   DS
```

```
Task Set 8:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         2.6453        39.9200       0.0481
       2        67.9412         1.5583        14.6200       0.0229
       3       115.5000         2.2323         7.3500       0.0193
       4       165.0000         7.8573       155.2600       0.0476
       5       177.6923         3.7477       177.4300       0.0211
       6       192.5000         5.1221        70.1000       0.0266
       7       288.7500        21.2875        99.6100       0.0737
       8       330.0000        19.7984       322.2700       0.0600
       9       462.0000        18.5417       162.4000       0.0401
      10       770.0000        31.1742       229.0800       0.0405

      Aperiodic Server Period = 55.00

      Server Execution Time:    30.58   Polling, PE, SS
                                25.32   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         3.9680        39.9200       0.0721
       2        67.9412         2.3374        14.6200       0.0344
       3       115.5000         3.3484         7.3500       0.0290
       4       165.0000        11.7859       155.2600       0.0714
       5       177.6923         5.6216       177.4300       0.0316
       6       192.5000         7.6832        70.1000       0.0399
       7       288.7500        31.9312        99.6100       0.1106
       8       330.0000        29.6976       322.2700       0.0900
       9       462.0000        27.8126       162.4000       0.0602
      10       770.0000        46.7613       229.0800       0.0607

      Aperiodic Server Period = 55.00

      Server Execution Time:    18.35   Polling, PE, SS
                                16.95   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----       ------
       1        55.0000         5.2907        39.9200       0.0962
       2        67.9412         3.1166        14.6200       0.0459
       3       115.5000         4.4646         7.3500       0.0387
       4       165.0000        15.7146       155.2600       0.0952
       5       177.6923         7.4955       177.4300       0.0422
       6       192.5000        10.2443        70.1000       0.0532
       7       288.7500        42.5750        99.6100       0.1474
       8       330.0000        39.5967       322.2700       0.1200
       9       462.0000        37.0835       162.4000       0.0803
      10       770.0000        62.3484       229.0800       0.0810

      Aperiodic Server Period = 55.00

      Server Execution Time:    6.22    Polling, PE, SS
                                5.79    DS
```

```
Task Set 9:

    Periodic Load 40%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         2.3564        34.0000        0.0428
       2        67.9412         3.3899         1.7700        0.0499
       3        70.0000         3.1052        52.6200        0.0444
       4        85.5556         3.2375         2.2800        0.0378
       5       135.8824         4.0951        80.9700        0.0301
       6       165.0000         5.1385       132.5300        0.0311
       7       330.0000         3.7619        67.4600        0.0114
       8       462.0000        34.3960       322.3000        0.0745
       9       770.0000        41.3422       338.5200        0.0537
      10      1155.0000        28.0214       725.3900        0.0243

      Aperiodic Server Period = 55.00

      Server Execution Time:    30.90   Polling, PE, SS
                                21.57   DS


    Periodic Load 60%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         3.5346        34.0000        0.0643
       2        67.9412         5.0848         1.7700        0.0748
       3        70.0000         4.6579        52.6200        0.0665
       4        85.5556         4.8562         2.2800        0.0568
       5       135.8824         6.1427        80.9700        0.0452
       6       165.0000         7.7078       132.5300        0.0467
       7       330.0000         5.6429        67.4600        0.0171
       8       462.0000        51.5940       322.3000        0.1117
       9       770.0000        62.0134       338.5200        0.0805
      10      1155.0000        42.0321       725.3900        0.0364

      Aperiodic Server Period = 55.00

      Server Execution Time:    18.88   Polling, PE, SS
                                17.92   DS


    Periodic Load 80%

      ID         Period      Exec Time          Phase         Util
      --         ------      ---------          -----        ------
       1        55.0000         4.7127        34.0000        0.0857
       2        67.9412         6.7797         1.7700        0.0998
       3        70.0000         6.2105        52.6200        0.0887
       4        85.5556         6.4750         2.2800        0.0757
       5       135.8824         8.1903        80.9700        0.0603
       6       165.0000        10.2771       132.5300        0.0623
       7       330.0000         7.5238        67.4600        0.0228
       8       462.0000        68.7919       322.3000        0.1489
       9       770.0000        82.6845       338.5200        0.1074
      10      1155.0000        56.0428       725.3900        0.0485

      Aperiodic Server Period = 55.00

      Server Execution Time:    6.86    Polling, PE, SS
                                6.43    DS
```

II. SS-SIM: A Simulator for Periodic and Aperiodic Tasks

The simulator used to conduct many of the experiments discussed in this thesis is written in Simscript II.5 [Simscript 88] and is called *ss-sim*. This appendix describes how to use the simulator to conduct experiments to predict the average aperiodic response time for a set of real-time tasks.

II.1 Setting Up and Running an Experiment to Determine Average Aperiodic Response Time Performance

The operation of the simulator will be described for several examples. For each example, the dialog between the user and ss-sim and the results generated by ss-sim will be described. For the first three examples, it is assumed that the user wants to investigate the aperiodic response time performance of the set of periodic and aperiodic tasks presented in Figure II-1. The user wants to answer the following three questions:

1. What is the maximum execution time for a sporadic server with a period of 10 units using this task set?
2. What is the average aperiodic response time for background aperiodic service?
3. What is the average aperiodic response time using a sporadic server?

Periodic Tasks:

Period	Execution Time	Initial Phase
10	2	0
15	3	0
50	15	0

Aperiodic Task:

Mean Interarrival Time	Mean Execution Time
10	2

Figure II-1: Task Set Used For Simulator Examples

II.1.1 Finding the Maximum Execution Time for a Sporadic Server

The first example of using ss-sim will be to answer the first question above. The dialog between the user and ss-sim for this example is presented in Figure II-2. In this figure and in the other dialog figures presented later, the user’s input is presented in bold-italic and is underlined. The user begins by entering ss-sim at the system prompt. This is followed by a welcome message printed by ss-sim and ss-sim then asks which simulation mode to use. The user then selects mode 2: **Find Sporadic Server Size**. A message is then printed to warn the user that, the priority of the sporadic server should be different from the priorities of the periodic tasks (ss-sim interprets low numbers as high priorities). Next, the user is asked for the sporadic server’s period and priority. The user is then asked to enter the number of periodic tasks and the priority, period, and execution time of each periodic task. Once all this data has been entered, ss-sim uses a binary search technique described in Section 3.7.1 to compute the maximum

```
Welcome to Brink’s simulator...

Choose Mode:
    1) Normal Simulation
    2) Find Sporadic Server Size
    3) Generate Task Sets

2

Note:
    For this mode of operation, it is necessary to specify
    a priority for the sporadic server that is different
    from the priorities of all the periodic tasks.

Enter SS period:
10

Enter SS priority:
0

Enter the number of periodic tasks:
3

Enter the periodic tasks as follows:
     priority period   exec.time

For Example:  periodic task  1: 1 10 4

periodic task    1:
1 10 2
periodic task    2:
2 15 3
periodic task    3:
3 50 15

Server Type     = Sporadic Server
Server Period   =      10.0000
Server Priority =       0.

scale.now =   1.0000
scale.now =    .5000
scale.now =    .2500
scale.now =    .3750
scale.now =    .3125
scale.now =    .2813
scale.now =    .2656
scale.now =    .2578
scale.now =    .2617
scale.now =    .2598
scale.now =    .2607
scale.now =    .2603
scale.now =    .2600
scale.now =    .2599
scale.now =    .2599

final scale =    .2597

Server:
     10.0000 period
      2.5975 execution time
       .2597 utilization
```

Figure II-2: Computing Maximum Sporadic Server Execution Time

sporadic server execution time. Once the search is complete, ss-sim indicates a maximum sporadic server execution time of 2.5975 units and exits.

II.1.2 Simulating Background Aperiodic Service

The second example demonstrates the use of ss-sim to predict the average aperiodic response time for background aperiodic service (the second question above). Figure II-3 presents the dialog between the user and ss-sim for this example. To run an aperiodic task simulation, the user selects the **Normal Simulation** mode. Next, ss-sim asks for the number of sporadic servers to be used. To simulate background aperiodic service, the user indicates that zero sporadic servers are to be used. Ss-sim then asks the user to choose the aperiodic service time distribution. The user selects an exponential service time distribution. Next, the user is asked for the number of aperiodic tasks and the mean interarrival time and the mean service time for each aperiodic task. Next, ss-sim asks for the parameters of the periodic task set. The user indicates that 3 periodic tasks will be used and enters the priority, period, execution time, and initial start time for each periodic task using the information from Figure II-1. Next, ss-sim asks for the length of the simulation. Ss-sim has now obtained all the information it requires and begins the simulation. Ss-sim prints the simulation time ten times during the simulation to indicate that progress is being made. A final message is printed when the simulation is finished.

All of the results from aperiodic simulations are stored in a file named *ss-sim.out* before the simulator exits. Figures II-4 and II-5 present the contents of the ss-sim.out file for the simulation described above. First, the input data concerning the aperiodic service algorithm and the aperiodic tasks are summarized. Next, a summary of the periodic task information is given followed by a summary of the priority levels the simulator created from the aperiodic server and periodic task information. For each priority level, the amount of time available for periodic task use and sporadic server use during the indicated period is listed. Next, the actual length of the simulation is listed followed by the desired length of the simulation (i.e., the target simulation stop time). If the simulation detects no errors or missed deadlines, these two lengths will be equal.

Figure II-5 presents the results for the simulation described by the information in Figure II-4. The first line of Figure II-5 indicates that all the hard deadlines of the periodic tasks were met during the simulation. Next, the total number of swap-ins and swap-outs is indicated. This is followed by the swap-in and swap-out data on a per task basis for both the periodic and aperiodic tasks. The last information provided is the aperiodic response time data. For example, these data indicate that the mean aperiodic response time was 24.4462 units.

II.1.3 Simulating Aperiodic Service Using a Sporadic Server

The third example demonstrates the use of ss-sim to predict the average aperiodic response time using a sporadic server. This example uses the periodic task set shown in Figure II-1 and the sporadic server execution time data determined earlier and displayed in Figure II-2. The dialog between ss-sim and the user for this example is presented in Figures II-6 and II-7. The dialog for this example is similar to that presented for background aperiodic service. However, serveral differences are present in this sporadic server example. First, the user indicates that a sporadic server is to be used. Ss-sim then asks the user to selece a full or a simple replenishment policy for the sporadic servers. The difference between these policies is described in Section 3.5. Next, the user enters the priority, execution time, and period of the sporadic server. The last difference is that, when entering the aperiodic task information, the user also indicates the priority of the sporadic server that is to provide service for the aperiodic task.

```
Welcome to Brink’s simulator...

Choose Mode:
    1) Normal Simulation
    2) Find Sporadic Server Size
    3) Generate Task Sets

1

Enter the number of sporadic servers:
0

Enter the number of aperiodic tasks:
1

Choose the type of aperiodic execution time distribution:
    1 = Exponential, 2 = Constant
1

Enter the aperiodic tasks as follows:
   mean.inter.arrival.time  mean.exec.time

For Example:  aperiodic task  1: 15 2

aperiodic task    1:
10 2

Enter the number of periodic tasks:
3

Enter the periodic tasks as follows:
     priority period exec.time initial.start.time

For Example:  periodic task  1: 1 10 4 7

periodic task    1:
1 10 2 0
periodic task    2:
2 15 3 0
periodic task    3:
3 50 15 0

Enter the length of the simulation:
15000


Simulation Starting...
  Sim Time =       2.00
  Sim Time =    1501.55
  Sim Time =    3002.00
  Sim Time =    4502.00
  Sim Time =    6002.00
  Sim Time =    7502.00
  Sim Time =    9002.00
  Sim Time =   10502.00
  Sim Time =   12002.00
  Sim Time =   13500.76
Simulation stopped at time =   15000.00
```

Figure II-3: Simulating Background Aperiodic Service

Figures II-8 and II-9 present the contents of the ss-sim.out file for the sporadic server simulation described above. The format for these data is the same as for the data from the background aperiodic service example presented in Figures II-4 and II-5. However, Figures II-8 and II-9 also provide the priority, period, execution time, size, and load for each sporadic server. The load for a sporadic server is the total utilization of the aperiodic tasks serviced by the sporadic server divided by the size of the sporadic server.

```
ss-sim.out


Aperiodic service algorithm: Background


Here is the Aperiodic Task Info:

          Mean-Inter
Task     Arrival-Time     Exec-Time  Utilization  SS.Priority
   1           10.00         2.00      .2000         0.


Total Aperiodic Utilization =      .2000

Using Exponential Distribution for Aperiodic Execution Time

Using Exponential Distribution for Aperiodic Interarrival Time


Here is the Periodic Task Info:

Task   Priority     Period   Exec. Time     Start   Utilization
   1       1.00      10.00       2.00         0.       .2000
   2       2.00      15.00       3.00         0.       .2000
   3       3.00      50.00      15.00         0.       .3000

Total Periodic Utilization =      .7000

Here are the priority levels:

        Priority      Period        Periodic-Time            SS-Time
           1.00       10.00               2.00               0.
           2.00       15.00               3.00               0.
           3.00       50.00              15.00               0.


Length of the simulation (time.v):   15000.00

Target simulation stop time:   15000.00
```

Figure II-4: Simulation Results for Background Aperiodic Service, Part I

II.1.4 Simulating Aperiodic Service Using Multiple Sporadic Servers

The fourth ss-sim example demonstrates how to perform an experiment using multiple sporadic servers. For this example, we replace the second periodic task in Figure II-1 with an aperiodic task and create an additional sporadic server to service the new aperiodic task. The dialog between the user and ss-sim is presented in Figures II-10 and II-11. The ss-sim.out file for this example is presented in Figures II-13 and II-13.

II.1.5 Generating Periodic Task Sets

For the experiments presented in this thesis, 10 periodic task sets were used. The parameters for these task sets are listed in Appendix I. The templates for creating these task sets were generated using the **Generate Task Sets** mode of ss-sim. In this mode, ss-sim generates random periodic tasks sets. The generated task sets are presented with the execution times of their periodic tasks scaled to the point of maximum schedulability. To generate a periodic task set at a specific periodic load (e.g., a 40%, 60%, or 80% total periodic load), it is necessary to scale down the execution time for each of the periodic tasks. The task sets generated all have hyperperiods of 2310. The periodic tasks have periods ranging from 55 units to 2310 units and the minimum periodic task period for all task sets created is 55 units.

```
All deadlines were met.

Total number of swap ins  =   6444
Total number of swap outs =   2150

Periodic Task Swap In/Out Data:


      -----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Ptask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |----------------------------------------------------------------|
      |    1         1501          1501        1500            0    |
      |    2         1001          1000        1000            0    |
      |    3          301          1100         300          800    |
      -----------------------------------------------------------------


Aperiodic Task Swap In/Out Data:


      -----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Atask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |----------------------------------------------------------------|
      |    1         1493          2239        1493          746    |
      -----------------------------------------------------------------


Atask Service Data:


  ID        1/mu     1/lambda     Min RT     Max RT     Mean RT     Sdev RT
  --     --------     --------  ---------  ---------   ---------   ---------
   1      2.0000      10.0000      .0250   110.0341     24.4462     19.9710
```

Figure II-5: Simulation Results for Background Aperiodic Service, Part II

An example of the dialog between the user and ss-sim to generate random periodic task sets is given in Figure II-14. The user enters 3 to specify the **Generate-Task-Sets** mode. Next, ss-sim prompts the user for a file name for the task set data. Next, ss-sim asks for the number of task sets to generate, the number of periodic tasks per set, and whether or not the tasks should be randomly phased. Next, ss-sim asks the user for a seed to randomize the task set generation. This concludes all the information that ss-sim needs to generate random periodic task sets. Ss-sim then selects the task set parameters, scales the task set to its breakdown utilization (i.e., its maximum schedulable utilization), and lists the task set number as each task set is created. Ss-sim writes the task set data into a file using the specified file name and exists.

Figure II-15 presents the output file generated by ss-sim for the dialog presented in Figure II-14. The first four lines presents the information entered by the user to create the periodic tasks sets. Next, the task ID, period, execution time, phase, and utilization of each task in Task Set 1 is presented. The last line indicates the total utilization of this task set.

```
Welcome to Brink’s simulator...

Choose Mode:
    1) Normal Simulation
    2) Find Sporadic Server Size
    3) Generate Task Sets

1

Enter the number of sporadic servers:
1

Choose a Sporadic Server Replenishment Policy:
    1 = Full Replenishment Policy
    2 = Simple Replenishment Policy
1

Note:
    The sporadic server priorities must be different from
    all periodic task priorities.  Also, each sporadic
    server must have a different priority.


Enter the sporadic server information as follows:
    priority execution.time period

For Example:  sporadic server 1: 1 2 10

sporadic server    1:
0 2.59 10

Enter the number of aperiodic tasks:
1

Choose the type of aperiodic execution time distribution:
    1 = Exponential, 2 = Constant
1

Enter the aperiodic tasks as follows:
   mean.inter.arrival.time  mean.exec.time  SS.priority

For Example:  aperiodic task  1: 15 2 1

aperiodic task    1:
10 2 0
```

Figure II-6: Simulating Aperiodic Service with a Single Sporadic Server, Part I

```
Enter the number of periodic tasks:
3

Enter the periodic tasks as follows:
     priority period exec.time initial.start.time

For Example:  periodic task  1: 1 10 4 7

periodic task    1:
1 10 2 0
periodic task    2:
2 15 3 0
periodic task    3:
3 50 15 0

Enter the length of the simulation:
15000


Simulation Starting...
  Sim Time =       2.59
  Sim Time =    1501.55
  Sim Time =    3000.88
  Sim Time =    4501.24
  Sim Time =    6002.00
  Sim Time =    7500.00
  Sim Time =    9000.00
  Sim Time =   10501.40
  Sim Time =   12000.00
  Sim Time =   13500.76
Simulation stopped at time =   15000.00
```

Figure II-7: Simulating Aperiodic Service with a Sporadic Server, Part II

```
ss-sim.out


Aperiodic service algorithm: Sporadic Server


Here is the Sporadic Server Info:

    Priority    Period    Exec-Time    Size     Load
      0.        10.00       2.59       .2590     .7722

Sporadic Server Replenishment Policy: FULL


Here is the Aperiodic Task Info:

          Mean-Inter
Task     Arrival-Time     Exec-Time  Utilization  SS.Priority
   1           10.00         2.00      .2000         0.


Total Aperiodic Utilization =      .2000

Using Exponential Distribution for Aperiodic Execution Time

Using Exponential Distribution for Aperiodic Interarrival Time


Here is the Periodic Task Info:

Task   Priority     Period   Exec. Time     Start   Utilization
   1       1.00      10.00       2.00         0.       .2000
   2       2.00      15.00       3.00         0.       .2000
   3       3.00      50.00      15.00         0.       .3000

Total Periodic Utilization =      .7000

Here are the priority levels:

        Priority      Period        Periodic-Time            SS-Time
           0.         10.00              0.                   2.59
           1.00       10.00              2.00                 0.
           2.00       15.00              3.00                 0.
           3.00       50.00             15.00                 0.


Length of the simulation (time.v):   15000.00

Target simulation stop time:   15000.00
```

Figure II-8: Simulation Results for Aperiodic Service with a Sporadic Server, Part I

```
All deadlines were met.

Total number of swap ins  =   8951
Total number of swap outs =   4634

Periodic Task Swap In/Out Data:


      ----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Ptask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |--------------------------------------------------------------|
      |    1         1501         2236        1500           712   |
      |    2         1001         1361        1000           361   |
      |    3          301         1942         300          1642   |
      ----------------------------------------------------------------


Aperiodic Task Swap In/Out Data:


      ----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Atask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |--------------------------------------------------------------|
      |    1         1493         2808        1493          1315   |
      ----------------------------------------------------------------


Atask Service Data:


  ID        1/mu     1/lambda     Min RT     Max RT     Mean RT     Sdev RT
  --    --------     --------  ---------  ---------   ---------   ---------
   1      2.0000      10.0000      .0009    98.2641     14.3376     16.9750
```

Figure II-9: Simulation Results for Aperiodic Service with a Sporadic Server, Part II

```
Welcome to Brink’s simulator...

Choose Mode:
    1) Normal Simulation
    2) Find Sporadic Server Size
    3) Generate Task Sets

1

Enter the number of sporadic servers:
2

Choose a Sporadic Server Replenishment Policy:
    1 = Full Replenishment Policy
    2 = Simple Replenishment Policy
1

Note:
    The sporadic server priorities must be different from
    all periodic task priorities.  Also, each sporadic
    server must have a different priority.


Enter the sporadic server information as follows:
    priority execution.time period

For Example:  sporadic server 1: 1 2 10

sporadic server    1:
0 2.59 10
sporadic server    2:
2 3 15

Enter the number of aperiodic tasks:
2

Choose the type of aperiodic execution time distribution:
    1 = Exponential, 2 = Constant
1

Enter the aperiodic tasks as follows:
   mean.inter.arrival.time  mean.exec.time  SS.priority

For Example:  aperiodic task  1: 15 2 1

aperiodic task    1:
10 2 0
aperiodic task    2:
15 3 2
```

Figure II-10: Simulating Aperiodic Service with Multiple Sporadic Servers, Part I

```
Enter the number of periodic tasks:
2

Enter the periodic tasks as follows:
     priority period exec.time initial.start.time

For Example:  periodic task  1: 1 10 4 7

periodic task    1:
1 10 2 0
periodic task    2:
3 50 15 0

Enter the length of the simulation:
15000


Simulation Starting...
  Sim Time =       2.59
  Sim Time =    1500.35
  Sim Time =    3001.52
  Sim Time =    4502.00
  Sim Time =    6002.00
  Sim Time =    7502.00
  Sim Time =    9000.82
  Sim Time =   10502.00
  Sim Time =   12002.00
  Sim Time =   13500.00
Simulation stopped at time =   15000.00
```

Figure II-11: Simulating Aperiodic Service with Multiple Sporadic Servers, Part II

```
ss-sim.out


Aperiodic service algorithm: Sporadic Server


Here is the Sporadic Server Info:

    Priority    Period    Exec-Time    Size     Load
      0.        10.00       2.59       .2590     .7722
      2.00      15.00       3.00       .2000    1.0000

Sporadic Server Replenishment Policy: FULL


Here is the Aperiodic Task Info:

          Mean-Inter
Task     Arrival-Time     Exec-Time  Utilization  SS.Priority
   1          10.00         2.00      .2000         0.
   2          15.00         3.00      .2000         2.00


Total Aperiodic Utilization =      .4000

Using Exponential Distribution for Aperiodic Execution Time

Using Exponential Distribution for Aperiodic Interarrival Time


Here is the Periodic Task Info:

Task   Priority     Period   Exec. Time     Start   Utilization
   1       1.00      10.00       2.00         0.       .2000
   2       3.00      50.00      15.00         0.       .3000

Total Periodic Utilization =      .5000

Here are the priority levels:

        Priority      Period        Periodic-Time            SS-Time
           0.         10.00              0.                   2.59
           1.00       10.00              2.00                 0.
           2.00       15.00              0.                   3.00
           3.00       50.00             15.00                 0.


Length of the simulation (time.v):   15000.00

Target simulation stop time:   15000.00
```

Figure II-12: Simulation Results for Aperiodic Service with Multiple Sporadic Servers, Part I

```
All deadlines were met.

Total number of swap ins  =   8945
Total number of swap outs =   4640

Periodic Task Swap In/Out Data:


      -----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Ptask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |---------------------------------------------------------------|
      |    1         1501          2050        1500           520    |
      |    2          301          1815         300          1515    |
      -----------------------------------------------------------------


Aperiodic Task Swap In/Out Data:


      -----------------------------------------------------------------
      |          |   # Times  | # Times     |   # Times  |   # Times  |
      | Atask ID |  Activated | Swapped In |  Completed | Swapped Out|
      |---------------------------------------------------------------|
      |    1         1476          2420        1476           944    |
      |    2          997          2177         997          1178    |
      -----------------------------------------------------------------


Atask Service Data:


  ID        1/mu     1/lambda     Min RT     Max RT     Mean RT     Sdev RT
  --    --------     --------  ---------  ---------   ---------   ---------
   1      2.0000      10.0000      .0013    95.1209     10.7225     14.7698
   2      3.0000      15.0000      .0094   161.5009     30.2760     29.2302
```

Figure II-13: Simulation Results for Aperiodic Service with Multiple Sporadic Servers, Part II

```
Welcome to Brink’s simulator...

Choose Mode:
    1) Normal Simulation
    2) Find Sporadic Server Size
    3) Generate Task Sets

3

Note:
    All of the task sets generated will have a minimum task
    period of 55 units.


Enter file name for task set information.
new-task-sets

Enter the number of periodic task sets to generate.
1

Enter the number of periodic tasks per set.
10

Random phasing for periodic tasks? (y/n)
y

Enter an integer seed to randomize task set generation.
100


Selecting task set parameters...
Scaling task set to breakdown utilization...
scale.now =  1.000
scale.now =   .500
scale.now =   .750
scale.now =   .875
scale.now =   .938
scale.now =   .906
scale.now =   .922
scale.now =   .914
scale.now =   .910
final scale =   .914
Completed Task Set:   1


Finished.
```

Figure II-14: Generating Random Periodic Task Sets

```
             1 Number of Periodic Task Sets
            10 Number of Periodic Tasks Per Set
             1 Random Phasing (1 = yes)
           100 Integer Seed




    Task set    1

        task         period      exec-time    start-time   utilization
        ----         ------      ---------    ----------   -----------
           1        55.0000         8.1219      25.17         .1477
           2        66.0000         7.5141      43.65         .1138
           3        77.0000          .8673      62.87         .0113
           4        85.5556        14.6057       3.85         .1707
           5       105.0000        14.8256      14.52         .1412
           6       115.5000         6.3938      72.17         .0554
           7       154.0000         8.2625     103.30         .0537
           8       177.6923        25.4867     133.05         .1434
           9       330.0000         2.4540     164.20         .0074
          10       385.0000        26.4003     220.74         .0686

            Total Utilization =  .9131
```

Figure II-15: A Randomly Generated Periodic Task Set

III. Applying the Rate Monotonic Scheduling Approach

In this appendix, we briefly outline the steps a real-time system designer would follow to apply the rate monotonic approach to schedule a specific real-time application. This discussion is broken into two sections. The first section presents a flow diagram of the rate monotonic approach and discusses the type of information needed to apply the algorithms. It also discusses the rate monotonic schedulability analysis for hard-deadline periodic and aperiodic tasks that can share data. The first section closes with a discussion of the design of sporadic servers to meet the requirements of the soft-deadline aperiodic tasks. The second section presents an example of applying the rate monotonic approach to a specific real-time scheduling problem.

III.1 Scheduling A Real-Time Application

A high-level flow diagram of the steps taken in applying the rate monotonic scheduling theory to a real-time application is presented in Figure III-1. A real-time designer begins with the application requirements and creates the task set with its execution and timing requirements. Next, the designer performs a schedulability analysis for the hard-deadline tasks. Depending upon the outcome of this analysis, the designer may reconsider the assumptions made concerning the execution time, request rate, or deadlines and repeat the analysis searching for a schedulable solution. If no solution can be found, the rate monotonic approach will not work for this task set. If a schedulable solution is found, the designer then creates sporadic servers to service the soft-deadline aperiodic tasks and uses equations to predict average response time where possible and simulation to predict average response time for the cases where the equations do not apply. As with the periodic tasks, if the response time goals cannot be met, the designer may reconsider the assumptions for the soft-deadline aperiodic tasks in order to find a workable solution. If the response time goals for the soft-deadline aperiodic tasks can be met, then a solution has been obtained. Otherwise, the rate monotonic approach is not able to meet the response time requirements of the soft-deadline aperiodic tasks.

The task set specification created from the application requirements should be composed of one or more of the following types of tasks:

- **Hard-Deadline Periodic Tasks**. For each of these tasks, the period, worst-case execution time, and deadline must be known. The rate monotonic algorithm makes the assumption that the deadline for a periodic task coincides with the end of its period. However, deadlines shorter or longer than the period of the periodic task may still be schedulable with either a deadline or rate monotonic priority assignment as is described next in Section III.1.1.
- **Hard-Deadline Aperiodic Tasks**. For each of these tasks, the minimum interarrival time, worst-case execution time, and deadline must be known. Each of these tasks will be serviced with its own sporadic server. The schedulability analysis approach employed for periodic tasks is also used to schedule these tasks.
- **Soft-Deadline Aperiodic Tasks**. For each class of soft-deadline aperiodic tasks, the mean interarrival time, mean execution time, and desired average response time must be known.
- **Background Tasks**. These tasks have no timing requirements and are assigned the lowest priority in the system.

The specification of the task set should also note all data that must be shared between tasks in an exclusive manner (e.g., using semaphores) and the maximum execution time each task may require to modify and release the lock on the shared data.

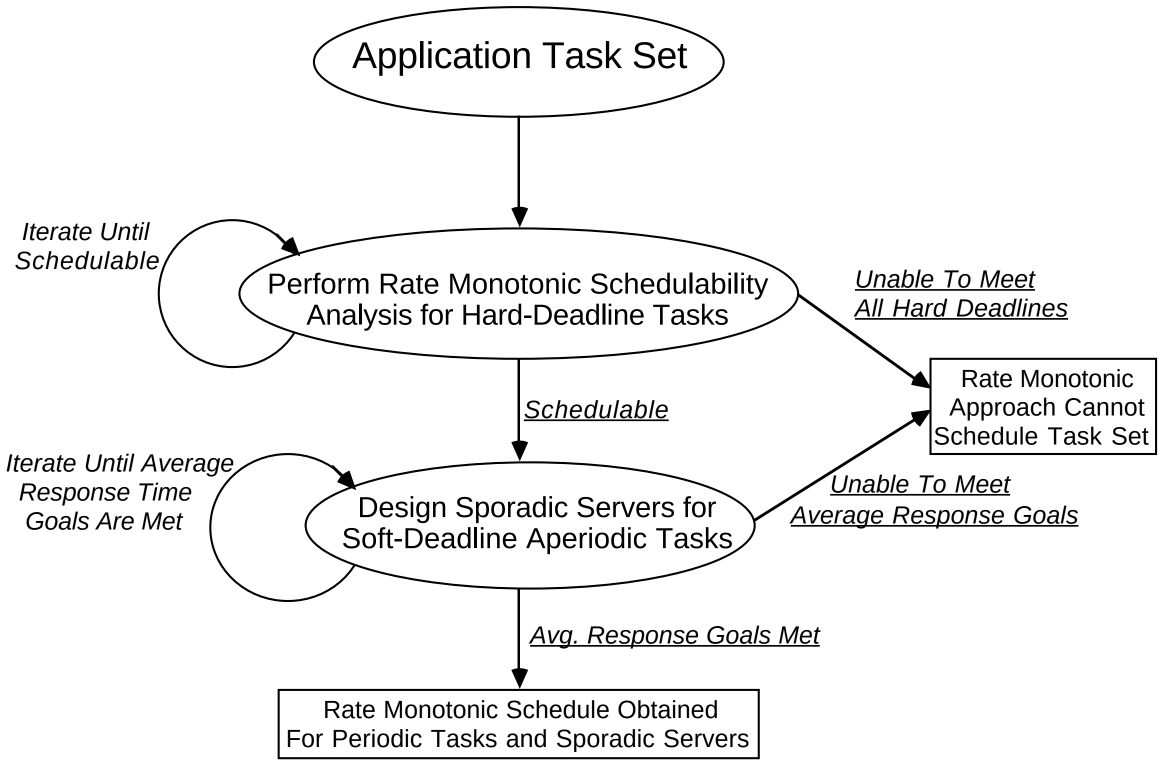

Application Task Set
Iterate Until Schedulable
Perform Rate Monotonic Schedulability Analysis for Hard-Deadline Tasks
Unable To Meet All Hard Deadlines
Rate Monotonic Approach Cannot Schedule Task Set
Schedulable
Iterate Until Average Response Time Goals Are Met
Design Sporadic Servers for Soft-Deadline Aperiodic Tasks
Unable To Meet Average Response Goals
Avg. Response Goals Met
Rate Monotonic Schedule Obtained For Periodic Tasks and Sporadic Servers

Figure III-1: Flow Diagram for Applying the Rate Monotonic Scheduling Approach

III.1.1 Scheduling Hard-Deadline Periodic and Aperiodic Tasks

Designing a priority assignment for the hard-deadline tasks and testing its schedulability requires the following steps:

1. Design a sporadic server for each hard-deadline aperiodic task. A sporadic server's period should be equal to the minimum interarrival time of its aperiodic task and its execution time should be equal to the worst-case execution time of its aperiodic task (as is discussed in Section 2.4).
2. Assign priorities to the periodic tasks and sporadic servers. Rate monotonic priority assignments should be used for tasks whose deadlines are equal to or greater than their periods or minimum interarrival times. Otherwise, deadline monotonic priority assignments should be used (as discussed in Section 2.4).
3. Determine the worst case blocking time each hard-deadline task may experience due to the sharing of data using the priority ceiling protocols (as discussed in Section 2.5 and [Sha 87]) or due to a task whose assigned priority is greater than its rate monotonic priority (this is necessary for short-deadline tasks as is discussed in Section 2.4).
4. In priority order, the schedulability of each task should be tested using the Equations 3 or 4 as discussed in Sections 2.4 and 2.5. If the schedulability analysis indicates that all the hard timing requirements have been met, then the problem of scheduling the soft-deadline aperiodic tasks may be addressed. However, if one or more of the timing requirements are not met, the designer may consider altering the assumptions of worst-case request rates and

execution times for one or more of the tasks and repeat these steps in search of a schedulable solution.

If the schedulability test in step 4 fails, the designer has a number of possible techniques he can use to attain a schedulable solution, a few of which are as follows. The schedulability analysis will identify the task or tasks that miss their deadlines. The designer may then consider reducing the execution time needed by these tasks or the higher priority tasks that preempt these tasks. The designer may also consider using a longer period or minimum interarrival time for these tasks. The blocking duration experienced by the tasks that miss their deadlines due to the sharing of data with lower priority tasks is another candidate for reduction. However, if no such modification is workable within the constraints of the application, then the rate monotonic approach cannot schedule this task set.

III.1.2 Scheduling Soft-Deadline Aperiodic Tasks

Scheduling the soft-deadline aperiodic tasks requires the following steps:

1. The soft-deadline aperiodic tasks must be placed into different classes based upon their mean interarrival time, their mean execution time, and their desired average response time. Each class should be described with one mean interarrival time, one mean execution time, and a desired average response time.
2. For each class of soft-deadline aperiodic tasks, a sporadic server should be created. In general, the server's size should larger than the load of the aperiodic tasks for which it is providing service and should have a priority high enough to meet its average response time requirements. However, a general solution for determining the best period, execution time, and priority of these sporadic servers to provide the desired response time remains an unsolved problem. The only case for which this problem has been solved is for a sporadic server that executes at the highest priority level in the system and services a low to moderate load of aperiodic tasks. For this case, the guidelines outlined in Section 3.9 and the equations developed in Section 4.2.1 can be used to design the sporadic server and predict the average response time for the aperiodic tasks it services. For all other cases, the selection of the sporadic server's period, execution time, and priority is a trial an error process for which simulation is necessary to predict the average response time performance.

If the equations or simulations for step 2 above indicate that the average response time goals for the aperiodic tasks will not be met, the designer may re-evaluate the response time goals and execution time assumptions and repeat steps 1 and 2 above to search for a schedulable solution. The techniques that the designer may apply are similar to those outlined above for the periodic tasks. If the blocking duration due to the ability of the aperiodic tasks to execute at lowest priority is a problem, the designer may consider disallowing low priority service for the aperiodic tasks and requiring that they only be serviced by their sporadic servers (as discussed in Sections 2.5 and 5.3).

III.2 An Example of the Rate Monotonic Scheduling Approach

In this section we present an example of how to use the rate monotonic approach and the techniques presented in this thesis to schedule a real-time task set. This example is designed to illustrate many of the scheduling problems that can be encountered in an actual real-time system. Figure III-2 presents the periodic and aperiodic tasks for this example. In Figure III-2, **ID** provides the identifier for a task or semaphore (periodic task IDs start with **P**, aperiodic task IDs start with **A**, and semaphore IDs start with

S), **C** indicates worst-case execution time, **T** indicates the period for periodic tasks and the minimum or average interarrival time for the aperiodic tasks, and **D** indicates the deadline for a given task.

Each of the tasks in Figure III-2 illustrates a different type of scheduling problem:

- **P1:** A short-deadline periodic task (**D** < **T**).
- **P2:** A "normal" periodic task (**D** = **T**).
- **P3:** A periodic task that shares data with a hard-deadline aperiodic task (**P3** and **A2** share data using semaphore **S1**).
- **P4:** A periodic task that shares data with a soft-deadline aperiodic task that will be serviced using a sporadic server (**P4** and **A3** share data using semaphore **S2**).
- **A1:** A "normal" hard-deadline aperiodic task (**D** = **T**).
- **A2:** A short-deadline aperiodic task (**D** < **T**) that shares data with **P3**.
- **A3:** A soft-deadline aperiodic task whose average response time goal can probably be met with a sporadic server. This aperiodic task shares data with **P4**.
- **A4:** A soft-deadline aperiodic task which will probably be serviced at low priority without a sporadic server. As yet, no equations are available to accurately predict the average response time performance for this aperiodic task.

To schedule the task set in Figure III-2, we follow the steps outlined in Section III.1 and in Figure III-1. We first consider the hard-deadline tasks and then consider the soft-deadline aperiodic tasks.

To schedule the hard-deadline tasks, we begin by creating sporadic servers for the hard-deadline aperiodic tasks (**A1** and **A2**). The execution time and periods of these sporadic servers are set equal to the execution time and minimum interarrival times for **A1** and **A2**. Next, priorities are assigned to the hard-deadline tasks (**P1**, **P2**, **P3**, **P4**, **A1**, **A2**). Rate monotonic assignments are used when **D** = **T** and deadline monotonic assignments when **D** < **T**. These priority assignments are shown in Figure III-3 where the hard-deadline tasks are listed in priority order with lower numbers indicating higher priority (the sporadic servers for **A1** and **A2** are simply listed as **A1** and **A2**).

The remaining information needed before a schedulability analysis can be performed is the worst-case blocking time for each of the tasks. For this task set, blocking occurs due to either a deadline monotonic priority assignment or due to the sharing of data. Figure III-3 also lists this information for each hard-deadline task. Blocking due to deadline monotonic priority assignments can occur for tasks **A2**, **P2**, **P3**, **A1**, and **P4** because tasks **P1** and **A2** have been given priorities higher than a rate monotonic priority assignment. The deadline monotonic blocking for **A2** is 1 unit from **P1**. The deadline monotonic blocking for **P2**, **P3**, **A1**, and **P4** is 1 unit from **P1** plus 1 unit from **A2**. The blocking due to the sharing of data affects tasks **A2**, **P2**, and **P4**. Tasks **A2** and **P3** share data using semaphore **S1**. Using the priority ceiling protocols to share data, **P3** may block the execution of all tasks with priorities greater than **P3**'s priority and equal to or lower than **A2**'s priority. Thus, tasks **A2** and **P2** can be blocked for 1 unit of time due to the sharing of data using **S1**. Tasks **P4** and **A3** share data using semaphore **S2**. For this stage of the analysis, we are assuming that all the soft-deadline aperiodic tasks execute at the lowest system priority. Therefore, the only task that **A3** can block is **P4**. This gives a maximum blocking time of 1 unit for **P4** due to the sharing of data using **S2**.

Periodic Tasks:

ID	C	T	D	% Utilization
P1	1	25	8	4.0
P2	1	10	10	10.0
P3	2	15	15	13.3
P4	2	20	20	10.0

Aperiodic Tasks:

Hard-Deadline:

ID	C	T	D	% Utilization
A1	1	16	16	6.25
A2	1	30	9	3.33

Soft-Deadline:

ID	Mean Service Time	Mean Interarrival Time	Desired Average Response Time	% Utilization
A3	2	20	2.5	10.0
A4	5	100	50	5.0

Shared Data:

Semaphore ID	Sharing Tasks	C
S1	P3, A2	1
S2	P4, A3	1

Figure III-2: Real-Time Task Set for Rate Monotonic Scheduling Example

Hard-Deadline Tasks:

Priority	ID	C	T	D	Blocking: Deadline Monotonic	Blocking: Sharing
1	P1	1	25	8	0	0
2	A2	1	30	9	1(P1)	1(S1)
3	P2	1	10	10	1(P1) + 1(A2)	1(S1)
4	P3	2	15	15	1(P1) + 1(A2)	0
5	A1	1	16	16	1(P1) + 1(A2)	0
6	P4	2	20	20	1(P1) + 1(A2)	1(S2)

Figure III-3: Priority Assignment and Blocking Durations for the Hard-Deadline Tasks

To perform a schedulability analysis for the tasks in Figure III-3, the following equations are used (these equations are presented in [Sha 87] and discussed in Sections 2.4 and 2.5):

$$\forall\, i,\ 1 \le i \le n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1) \tag{52}$$

$$\forall\, i,\ 1 \le i \le n, \quad \min_{(k,\,l) \in R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1 \tag{53}$$

where C_i and T_i are respectively the execution time and period of task τ_i, $U_i = C_i/T_i$ is the utilization of task τ_i, and $R_i = \{\, (k, l) \mid 1 \le k \le i,\ l = 1, \cdots, \lfloor T_i/T_k \rfloor \,\}$. The term B_i is the worst-case blocking time for task τ_i. Both equations 52 and 53 test sufficient conditions under which a task set is schedulable. Equation 52 is simpler and easier to apply than Equation 53. However, Equation 52 is more pessimistic than Equation 53. Thus, whenever Equation 52 indicates that a task is not schedulable, Equation 53 should then be used.

Figure III-4 presents the results of applying Equation 52 to all the tasks presented in Figure III-3. For the short-deadline tasks (i.e., where **D** < **T**), these equations assume that the period of the task is equal to its deadline (note the denominator of 8 on the left-hand side of the equation for **P1** instead of 25). Since the inequality generated for each task by Equation 52 holds we know that these tasks are schedulable and Equation 53 is not needed. The sporadic servers, the priority assignments, and the priority ceiling protocols can guarantee the hard deadlines for the tasks presented in Figure III-3 and, therefore, the hard-deadline requirements for this task set can be met with the rate monotonic approach.

We now consider the soft-deadline aperiodic tasks. To service **A3** we want to create a high-priority sporadic server. The total utilization of the hard-deadline tasks is 46.88% which should leave plenty of utilization left to service the aperiodic tasks which require only 15% of the resource utilization. From the simulation data and guidelines presented in Chapter 4, we know that the larger the sporadic server's execution time the better average aperiodic response time it will provide. Referring to the task data presented in Figure III-3, we want to estimate the maximum preemption or blocking time these tasks can

P1: $\frac{1}{8} + \frac{0}{8} \leq 1(2^{1/1} - 1)$

$0.1250 \leq 1$

A2: $\frac{1}{9} + \frac{1}{9} + \frac{1}{9} \leq 1(2^{1/1} - 1)$

$0.3333 \leq 1$

P2: $\frac{1}{10} + \frac{3}{10} \leq 1(2^{1/1} - 1)$

$0.40 \leq 1$

P3: $\frac{1}{10} + \frac{2}{15} + \frac{2}{15} \leq 2(2^{1/2} - 1)$

$0.3667 \leq 0.8284$

A1: $\frac{1}{10} + \frac{2}{15} + \frac{1}{16} + \frac{2}{16} \leq 3(2^{1/3} - 1)$

$0.4208 \leq 0.7798$

P4: $\frac{1}{10} + \frac{2}{15} + \frac{1}{16} + \frac{2}{20} + \frac{3}{20} \leq 4(2^{1/4} - 1)$

$0.5458 \leq 0.7568$

Figure III-4: Scheduling Equations for the Hard-Deadline Tasks

withstand without missing any deadlines. **P1** has a deadline of 8 units and a computation time of 1 unit. **P1** may also be blocked for 1 unit of time by the sharing of data between **P4** and **A3** when **A3** is serviced by a high-priority sporadic server. So, **P1** can withstand an additional blocking time of 6 units without missing its deadline of 8 units (1 + 1 + 6 = 8). **A2** must be able to tolerate 1 unit of preemption from **P1**, its own execution time of 1 unit, and the blocking time of 1 unit due to the sharing of data with **P3** and still meet its deadline of 9 units. This also leaves a maximum of 6 additional units that **A2** can be blocked and still meet its deadline (1 + 1 + 1 + 6 = 9). **P2** must be able to withstand the blocking of 3 units due to **P1**, **A2**, and **S1** and its own execution time of 1 unit. Again, this leaves a maximum of 6 additional units that **P2** can be blocked without missing its deadline (1 + 3 + 6 = 10). An additional blocking time of greater than 6 units may cause tasks **P1**, **A2**, and **P2** to miss their deadlines. Therefore, we choose 6 units of execution time for the high-priority sporadic server. We now need to choose a period for the sporadic server. A period of 10 units is not good because the total utilization of the task set and the sporadic server would then be over 100%. However, a period of 20 units dedicates 30% of the resource to the sporadic server and would require a total scheduled resource utilization of 76.88% which is probably schedulable by the rate monotonic approach given a guaranteed schedulability bound of 69% and an average bound of 88% [Lehoczky 89]. An exact analysis is necessary (and will be done later), but this estimate is enough to get a rough idea of the execution time and period of the sporadic server.

Before performing the analysis to see if the tasks in Figure III-3 and a sporadic server with priority 0 (the highest priority), **C** = 6, and **T** = 20 are schedulable, we should use the equations developed in Chapter 4 to see if the average response time requirement of the soft-deadline aperiodic task can be met. We will use the high-priority sporadic server to service **A3** since **A3** has a tighter timing requirement than **A4** does. We are assuming a Poisson arrival process and a constant service time for these aperiodic tasks. The appropriate equations are:

M/D/1:

$$L = \lfloor \mu C \rfloor + 1 \tag{54}$$

$$\rho_{over} = \left[\frac{2.33\sqrt{T} - \sqrt{(2.33)^2 T + 4T(L - \frac{1}{2})}}{-2T\sqrt{\mu}} \right]^2 \tag{55}$$

$$\text{for } \rho \leq \rho_{over}, \; W \approx \frac{\rho}{2\mu(1-\rho)} + \frac{1}{\mu} \tag{56}$$

Using **C** = 6, **T** = 20, $\mu = 1/2$ in Equations 54 and 55 yields $\rho_{over} = 0.1079$. Since the load for **A3** (10%) is less than ρ_{over} (10.79%), Equation 56 can be used to predict the average response time for these aperiodic jobs. Equation 56 predicts an average response time of **W** = 2.11 which is better than the desired average response time of 2.5 units. Therefore, if the hard-deadline tasks are schedulable with this sporadic server, then all of the hard deadlines will always be met and the desired response time for **A3** will also be met.

Now, we must perform a schedulability analysis for the task set presented in Figure III-5 which includes the high priority sporadic server (**SS**) described above. **SS** causes several changes from the original hard-deadline task set presented in Figure III-3. First of all, **SS** is now the highest priority task and we treat it as if it were a short-deadline periodic task with a deadline of 6 units. Since we are giving **SS** a deadline monotonic priority, this creates an additional blocking term for tasks **P1**, **A2**, **P2**, **P3**, **A1**, and **P4**. **SS** services **A3** which shares data with **P4**. Recall from Section 2.5 that this allows **A3** to both preempt and block **P4** (hence 1 unit of blocking due to sharing for **P4**). Also, note that **P4** can now block **A3** if **SS** has capacity and **P4** has locked **S2**. This creates another possibility for 1 unit of blocking due to the sharing of data between **P4** and **A3** via **S2** for all tasks with priorities higher than **P4** and equal to or lower than the sporadic server (i.e., **SS**, **P1**, **A2**, **P2**, **P3**, **A1**).

The application of Equation 52 to the task set in Figure III-5 is presented in Figure III-6. As before, we are treating the tasks with deadline monotonic priorities as if their periods were equal to their deadlines (i.e., in the equations for **SS**, **P1**, and **A2**). Note that the inequalities hold for **SS**, **P1**, **A2**, and **P2**

Sporadic Server with Hard-Deadline Tasks:

					Blocking	
Priority	ID	C	T	D	Deadline Monotonic	Sharing
0	SS	6	20	6	0	1(S2)
1	P1	1	25	8	6(SS)	1(S2)
2	A2	1	30	9	6(SS) + 1(P1)	max[1(S1), 1(S2)]
3	P2	1	10	10	6(SS) + 1(P1) + 1(A2)	max[1(S1), 1(S2)]
4	P3	2	15	15	6(SS) + 1(P1) + 1(A2)	1(S2)
5	A1	1	16	16	6(SS) + 1(P1) + 1(A2)	1(S2)
6	P4	2	20	20	6(SS) + 1(P1) + 1(A2)	1(S2)

Figure III-5: Priority Assignment and Blocking Durations for the Sporadic Server and the Hard-Deadline Tasks

indicating that these tasks are schedulable. However, the inequalities do not hold for **P3**, **A1**, and **P4** indicating that these tasks may not meet their deadlines. It is now necessary to apply Equation 53 to test the schedulability of tasks **P3**, **A1**, and **P4**. These equations are presented in Figure III-7. As can be seen from the equations for each task in Figure III-7, one or more of the inequalities generated by Equation 53 holds indicating that tasks **P3**, **A1**, and **P4** will also meet their deadlines.

At this point we have seen that all of the hard-deadlines for the task set presented in Figure III-2 can be met and the average response time requirements for **A3** can also be met using a high-priority sporadic server. This leaves us with the remaining timing requirement for **A4**. One could service **A4** in background or with another sporadic server. However, at the time of this writing, we have no analytical approach or equations to predict the average response time for **A4**.

SS: $\frac{6}{6} + \frac{0}{6} \leq 1(2^{1/1} - 1)$

$1 \leq 1$

P1: $\frac{1}{8} + \frac{6}{8} + \frac{1}{8} \leq 1(2^{1/1} - 1)$

$1 \leq 1$

A2: $\frac{1}{9} + \frac{7}{9} + \frac{1}{9} \leq 1(2^{1/1} - 1)$

$1 \leq 1$

P2: $\frac{1}{10} + \frac{8}{10} + \frac{1}{10} \leq 1(2^{1/1} - 1)$

$1 \leq 1$

P3: $\frac{1}{10} + \frac{2}{15} + \frac{8}{15} + \frac{1}{15} \leq 2(2^{1/2} - 1)$

$0.8333 > 0.8284$

A1: $\frac{1}{10} + \frac{2}{15} + \frac{1}{16} + \frac{8}{16} + \frac{1}{16} \leq 3(2^{1/3} - 1)$

$0.8583 > 0.7798$

P4: $\frac{1}{10} + \frac{2}{15} + \frac{1}{16} + \frac{2}{20} + \frac{8}{20} + \frac{1}{20} \leq 4(2^{1/4} - 1)$

$0.8458 > 0.7568$

Figure III-6: Scheduling Equations for the Sporadic Server and the Hard-Deadline Tasks, Part I

P3: i = 2, Check if any of the following inequalities hold:

(k,l)		
(1,1)	$C_{P2} + C_{P3} + B_{P3} \leq T_{P2}$	$1 + 2 + 9 > 10$
(2,1)	$2C_{P2} + C_{P3} + B_{P3} \leq T_{P3}$	$2 + 2 + 9 \leq 13$

A1: i = 3, Check if any of the following inequalities hold:

(k,l)		
(1,1)	$C_{P2} + 2C_{P3} + C_{A1} + B_{A1} \leq T_{P2}$	$1 + 4 + 1 + 9 > 10$
(2,1)	$2C_{P2} + C_{P3} + C_{A1} + B_{A1} \leq T_{P3}$	$2 + 2 + 1 + 9 \leq 15$
(3,1)	$2C_{P2} + 2C_{P3} + C_{A1} + B_{A1} \leq T_{A1}$	$2 + 4 + 1 + 9 \leq 16$

P4: i = 4, Check if any of the following inequalities hold:

(k,l)		
(1,1)	$C_{P2} + C_{P3} + C_{A1} + C_{P4} + B_{P4} \leq T_{P2}$	$1 + 2 + 1 + 2 + 9 > 10$
(1,2)	$2C_{P2} + 2C_{P3} + 2C_{A1} + C_{P4} + B_{P4} \leq 2T_{P2}$	$2 + 4 + 2 + 2 + 9 \leq 20$
(2,1)	$2C_{P2} + C_{P3} + C_{A1} + C_{P4} + B_{P4} \leq T_{P3}$	$2 + 2 + 1 + 2 + 9 > 15$
(3,1)	$2C_{P2} + 2C_{P3} + C_{A1} + C_{P4} + B_{P4} \leq T_{A1}$	$2 + 4 + 1 + 2 + 9 > 16$

Figure III-7: Scheduling Equations for the Sporadic Server and the Hard-Deadline Tasks, Part II

References

[Ada 83] *Reference Manual for the Ada Programming Language*
U.S. Department of Defense, Washington, D.C., 1983.

[Allen 78] Arnold O. Allen.
Probability, Statistics, and Queueing Theory.
Academic Press, 1978.

[Borger 87] M. W. Borger.
VAXELN Experimentation: Programming a Real-time Periodic Task Dispatcher using VAXELN Ada 1.1.
Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, September, 1987.

[Borger 89] 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.

[Davari 86] 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.

[Dhall 78] S. K. Dhall and C. L. Liu.
On a Real-Time Scheduling Problem.
Operations Research 26(1):127-140, February, 1978.

[Everitt 84] D. E. Everitt and T. Downs.
The Output of the M/M/s Queue.
Operations Research 32(4):796-808, July-August, 1984.

[FutureBus 89] *Futurebus P896.2 Specification*
IEEE, 1989.

[Gross 84] Donald Gross and Douglas R. Miller.
The Randomization Technique as a Modeling Tool and a Solution Procedure for Transient Markov Processes.
Operations Research 32(2):343-361, March-April, 1984.

[Hood 86] Philip Hood and Vinod Grover.
Designing Real Time Systems in Ada, Final Report.
Technical Report 1123-1, SofTech, Inc. 460 Totten Pond Rd, Waltham, MA 02254-9197, January 8, 1986.

[Kelton 85] W. David Kelton and Averill M. Law.
The Transient Behavior of the M/M/s Queue, with Implications for Steady-State Simulation.
Operations Research 33(2):378-396, March-April, 1985.

[Kleinrock 75] Leonard Kleinrock.
Queueing Systems, Volume I: Theory.
John Wiley & Sons, 1975.

[Kleinrock 76] Leonard Kleinrock.
Queueing Systems, Volume II: Computer Applications.
John Wiley & Sons, 1976.

[Lawler 81] Eugene L. Lawler and Charles U. Martel.
Scheduling Periodically Occurring Tasks on Multiprocessors.
Information Processing Letters 12(1):9 - 12, February, 1981.

[Lehoczky 87] 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.

[Lehoczky 89] J. P. Lehoczky, L. Sha, and Y. Ding.
The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.
In *Proceedings of the Real-Time Systems Symposium*, pages 166-171. IEEE, Santa Monica, CA, December, 1989.

[Leung 80] Joseph Y.-T. Leung and M. L. Merrill.
A Note on Preemptive Scheduling of Periodic, Real Time Tasks.
Information Processing Letters 11(3):115 - 118, November, 1980.

[Leung 82] Joseph Y.-T. Leung and Jennifer Whitehead.
On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks.
Performance Evaluation 2:237-250, 1982.

[Little 61] J. D. C. Little.
A proof of the queueing formula: $L = \lambda W$.
Operations Research 9(3):383-387, 1961.

[Liu 73] 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.

[Liu 82] C. L. Liu, Jane W. S. Liu, and Arthur L. Liestman.
Scheduling with Slack-Time.
Acta Informatica 17:31-41, 1982.

[Liu 87] Jane W. S. Liu, Kwei-Jay Lin, and Swaminathan Natarajan.
Scheduling Real-Time, Periodic Jobs Using Imprecise Results.
In *Proceedings of the 8th Real-Time Systems Symposium*, pages 252-260. IEEE, San Jose, California, December, 1987.

[Mok 78] A. K. Mok and M. L. Dertouzos.
Multiprocessor Scheduling in a Hard Real-Time Environment.
In *Proceedings of the 7th Texas Conference on Computer Systems*, pages 5.1 - 5.12. IEEE/ACM, Houston, Texas, November, 1978.

[Mok 83] A.K. Mok.
Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.
PhD thesis, M.I.T., 1983.

[Odoni 83] Amedeo R. Odoni and Emily Roth.
An Empirical Investigation of the Transient Behavior of Stationary Queueing Systems.
Operations Research 31(3):432-455, May-June, 1983.

[Pegden 82] Claude Dennis Pegden and Matthew Rosenshine.
Some New Results for the M/M/1 Queue.
Management Science 28(7):821-828, July, 1982.

[Rajkumar 89] Ragunathan Rajkumar.
Task Synchronization in Real-Time Systems.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August, 1989.

[Ramamritham 84]
Krithivasan Ramamritham and John A. Stankovic.
Dynamic Task Scheduling in Hard Real-Time Distributed Systems.
IEEE Software 1(3):65-74, July, 1984.

[Rider 76] Kenneth Lloyd Rider.
A Simple Approximation to the Average Queue Size in the Time-Dependent M/M/1 Queue.
Journal of the ACM 23(2):361-367, April, 1976.

[Sha 86] 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.

[Sha 87] 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.

[Sha 89a] 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.

[Sha 89b] 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.

[Simscript 88] *Simscript II.5 Programming Language*
CACI Products Company, 3344 North Torrey Pines Court, La Jolla, CA 92037, 1988.

[Sprunt 88] Brinkley Sprunt, John P. Lehoczky, and Lui Sha.
Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm.
In *Proceedings of the 9th Real-Time Systems Symposium*, pages 251-258. IEEE, Huntsville, AL, December, 1988.

[Sprunt 89a] 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.

[Sprunt 89b] 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.

[Stankovic 88] John A. Stankovic.
A Serious Problem for Next-Generation Systems.
Computer 21(10):10-19, October, 1988.

[Strosnider 88] Jay Kurt Strosnider.
Highly Responsive Real-Time Token Rings.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August, 1988.

[Tokuda 87] Hideyuki Tokuda, James W. Wendorf, and Huay Yong Wang.
Implementation of a Time-Driven Scheduler for Real-Time Operating Systems.
In *Proceedings of the 8th Real-Time Systems Symposium*, pages 271-280. IEEE, San Jose, California, December, 1987.

[Tokuda 88] H. Tokuda and M. Kotera.
Scheduler 1-2-3: An Interactive Schedulability Analyzer for Real-Time Systems.
In *Proceedings of IEEE Compsac '88*. IEEE, October, 1988.

[Vanas 86] Harmen R Van As.
Transient Analysis of Markovian Queueing SYstems and Its Application to Congestion-Control Modeling.
IEEE Journal on Selected Areas in Communications SAC-4(6):891-904, September, 1986.

[Zhao 87a] W. Zhao, K. Ramamritham, and J. A. Stankovic.
Scheduling Tasks with Resource Requirements in Hard Real-Time Systems.
IEEE Transactions on Software Engineering SE-13(5), May, 1987.

[Zhao 87b] W. Zhao, K. Ramamritham, and J. Stankovic.
Preemptive Scheduling Under Time and Resource Constraints.
IEEE Transactions on Computers C-36(8), August, 1987.

Table of Contents

List of Figures