

Advances in Static- and Fixed-Priority Real-Time Scheduling

(Written Preliminary Examination - II : Critical Review Report)

Arun G Chandrashekarapuram(arunc@seas.upenn.edu)
Department of Computer and Information Sciences
University of Pennsylvania

September 21, 2004

Abstract

Real-time periodic task scheduling has received wide attention from researchers in the scheduling area. An efficient and optimal algorithm which statically prioritizes the tasks for scheduling is the rate monotonic (RM) algorithm. Though RM algorithm provides a succinct linear time sufficient test for deciding schedulability of a given task set, several efforts have been made to improve the test, while trying to keep the computational complexity low. In this paper, we study a new test for deciding RM schedulability. RM can naturally be extended to schedule tasks on multiprocessors globally, although no good schedulability test is known. We study a new global multiprocessor scheduling algorithm based on RM, which provides a sufficient test. Periodic servers can also be used to schedule aperiodic tasks without jeopardizing periodic schedulability. Fixed-priority servers which add an additional task to the periodic task set to serve the aperiodic tasks is a well-known strategy for the same. We study some new findings regarding fixed-priority servers in this paper.

1 Introduction

The real-time scheduling of preemptive hard real-time periodic tasks has received wide attention from scheduling community. Periodic tasks are tasks which recur after a fixed time period. The scheduling algorithms for periodic tasks can be classified into two types : static priority algorithms which assign priorities to *tasks* and all the recurrences of a task have the same priority, and dynamic priority algorithms which assign priorities to *each recurrence* of a periodic task. The rate-monotonic (RM) scheduling algorithm is a well-known static priority algorithm. The RM algorithm assigns priorities to tasks in the inverse ratio of their periods : the task with the lowest period gets the highest priority. The RM algorithm was proved to be optimal among all static-priority scheduling algorithms by Liu and Layland[17], which means that the RM algorithm can schedule any set of tasks that is schedulable by any other static priority algorithm.

In the same paper[17], Liu and Layland gave a sufficient test for deciding schedulability of a periodic task set. Let $\tau = \{\tau_1, \dots, \tau_n\}$ be a set of n periodic tasks with task τ_i having a (worst-case) execution time requirement of C_i , a time period of T_i and a deadline D_i relative to the period. Then the *utilization* of τ_i is defined as $U_i = C_i/T_i$, and the utilization of τ is defined as $\sum_{i=1}^n U_i$. The schedulability test of Liu and Layland states that any task set satisfying

$$\sum_{i=1}^n U_i \leq n(2^{1/n} - 1) \quad (1)$$

is schedulable by the RM algorithm. This means that the test declares any τ with n tasks as schedulable by RM if its utilization is bounded from above by the RHS of inequality (1). For large n , the bound is > 0.69 , which means *any* task set whose utilization is ≤ 0.69 is accepted by this test as schedulable using RM. Note

that some task sets not satisfying this criteria might be schedulable by RM. Therefore, this is only a sufficient but not a necessary test. After this work has been done to get a better sufficient test, that is, a test which offers a higher bound. One such example of improvement is the Hyperbolic-Bound test (HB) proposed by Bini et al.[5]. Another direction for research has been to provide *exact*, that is, both necessary and sufficient conditions for deciding schedulability. Lehoczky et al. [16] provided an exact test for schedulability which requires pseudo-polynomial time. Tests equivalent to this test were also provided independently by Joseph et al.[15] and Audsley et al.[3]. In this review, we study another recently proposed algorithm for testing RM schedulability by Bini and Buttazzo[6]. The test is slightly novel in that it contains an exact schedulability test which does not perform any worse than the previous well-known exact tests for schedulability, and on the other hand is amenable to tuning via a parameter to yield sufficient tests for schedulability trading off acceptance ratio with computational complexity.

Research has also been done for preemptive scheduling on *multiprocessors*. The rate monotonic algorithm and some other uniprocessor scheduling algorithms can be naturally extended in the multiprocessor case. The algorithms for multiprocessors can be divided into two categories : partitioned and global[2]. While partitioned algorithms require all instantiations of a task to run on a predestined processor, a global algorithm places no such restrictions. Partitioned approaches can benefit from well-known uniprocessor algorithms, but Andersson et al.[2] say that it is not clear what good global scheduling strategies should be. Further, partitioned scheduling algorithms have been studied extensively[2]. Some results on global scheduling algorithms such as the multiprocessor RM algorithm have also been produced : for eg. it has been proved that the utilization bound for RM in the multiprocessor case is $1/m$ [2]. In this paper, we study an RM-based multiprocessor scheduling algorithm by Andersson et al.[2] that provides a sufficient utilization based test for schedulability. The proof strategies are quite general and we will derive utilization based bounds for two other algorithms : multiprocessor DM and multiprocessor EDF.

Another interesting area of research is fixed-priority aperiodic scheduling where hard real-time periodic tasks are scheduled along with soft aperiodic tasks. That is, without compromising the schedulability of periodic tasks, the algorithms seek to minimize the response time of the aperiodic tasks. Two well-known algorithms in this area include the deferrable server (DS) algorithm discussed by Lehoczky et al.[13], and the sporadic server (SS) algorithm. Both the algorithms have a special periodic task called the server task along with the other periodic tasks, with a time period and a maximum capacity. SS and DS chiefly differ in the way the server capacity is replenished. Since the SS provides a better utilization based bound for schedulability of periodic tasks than DS(see [7]), SS is usually preferred to DS. However, Bernat et al. [4] show that if the exact analysis with a new parameter selection technique is used, DS performance is similar to that of SS. Based on extensive simulation experiments, they contend that DS should also be considered while implementing fixed-priority servers.

The organization of the paper is as follows. In section 2, we define our model of periodic tasks, introduce RM scheduling, and give a brief overview of the known schedulability tests. We also define DM, EDF and PSS scheduling algorithms. We then introduce fixed-priority servers with examples. In section 3, we study the results of Bini and Buttazzo[6]. In section 4, we study the results of Andersson et al.[2], and extend their results to DM and EDF case. In section 5, we study the results of Bernat and Burns[4]. In section 6, we extend the test proposed by Bini and Buttazzo[6] for DM scheduling and deferrable servers. We also propose a few models for fixed-priority multiprocessor scheduling.

2 Background and Notation

In this section, we introduce relevant notation and review some of the well-known results about hard real-time *uniprocessor* scheduling. All algorithms are *preemptive* algorithms.

A *periodic task set* τ is a set of n (≥ 1) tasks $\{\tau_1, \dots, \tau_n\}$ where each task τ_i is characterized by (C_i, T_i, D_i, J_i) , where $C_i \in \mathbb{N}(\geq 1)$ is the (worst-case) execution time of the task, and $T_i \in \mathbb{N}(\geq C_i)$ is the period. Without loss of generality, let $\forall i : T_i \leq T_{i+1}$. The j^{th} ($j \geq 1$) instance τ_{ij} of τ_i is an instantiation of τ_i at the beginning of the j^{th} period. $J_i \in \mathbb{N}(\geq 0)$ is the release time of the first instance of the task,

and the τ_{ij} is released at $r_{ij} = J_i + (j - 1)T_i$. An instance which has been released but has not completed execution is said to be active. $D_i \in \mathbb{N}$ is the relative deadline of an instance : the absolute deadline of τ_{ij} is $d_{ij} = J_i + (j - 1)T_i + D_i$. Unless otherwise mentioned, $D_i = T_i$ for all tasks. A task is given by (C_i, T_i) if the release time is 0 and the deadline is equal to the period. Let f_{ij} denote the time at which τ_{ij} finishes execution. The length of the time interval $[r_{ij}, f_{ij}]$ is the response time of the instance τ_{ij} . The *utilization* of τ_i is defined as $U_i = \frac{C_i}{T_i}$ and the utilization of τ is defined as

$$U = \sum_{i=1}^n U_i = \sum_{i=1}^n \frac{C_i}{T_i}. \quad (2)$$

τ_i is said to be *schedulable* by a scheduling algorithm A if $\forall j : f_{ij} \leq d_{ij}$. τ is said to be *schedulable* by A if τ_i is schedulable by A for all i . One standard method of scheduling real-time tasks is **priority driven** scheduling : each instance of a task τ_i is assigned a priority $p_i \in \mathbb{N}(\geq 1)$ and at any time t , the scheduler schedules the instance with the highest priority among all active instances.

A *static-priority* scheduling algorithm is an algorithm which assigns priorities to *tasks*, that is, all instances of the task have the same priority. On the other hand, a *dynamic priority* scheduling algorithm assigns priorities to *instances* of tasks and so different instances of the same task may have different priorities, and further, the priority assigned to an instance can also change with time. A scheduling algorithm is said to be *optimal* if it can schedule any task set that is schedulable by any other algorithm.

<i>Algorithm</i>	<i>Schedule</i>
RM	111+2+33111+2+33+111+32+3111+33+
EDF	111+2+33111+33+2+111+32+3111+33+

Table 1: Schedule produced by a static-priority and dynamic-priority assignment to the task set $\{(3, 6), (1, 8), (4, 12)\}$. i is used to indicate that an instance of task i executes in that unit-time slot, and i^+ indicates that it completes execution in that time slot.

For example, consider the (periodic) task set $\{(3, 6), (1, 8), (4, 12)\}$. All the tasks are released at time $t = 0$. The utilizations of the tasks are respectively $1/2, 1/8$ and $1/3$, and utilization of the task set is $23/24$. A static priority assignment to the tasks could be 1, 2, 3 respectively, that is τ_1 has the highest priority and τ_3 has the lowest (call this the rate-monotonic (RM) assignment). The schedule produced using this priority assignment from time $t = 0$ to time $t = 24$ is shown in table 1. Note that this schedule repeats after time $t = 24$, since 24 is the LCM of the periods. There is no idle time in this schedule. A dynamic priority assignment could be : schedule the instance which has the earliest deadline among all the active instances (call this the earliest-deadline-first assignment (EDF)). The schedule produced by this algorithm is shown in table 1. An example of an unschedulable task set is $\{(3, 6), (1, 8), (5, 12)\}$. The utilization is $25/24 > 1$; therefore, since the total the total computation requirement in the interval $[0, 24]$ is greater than 24, no algorithm can schedule this task set on a single processor.

A *sporadic* task set is same as a periodic task set except that the inter-arrival time of successive instances is *at least* T_i for every task τ_i .

2.1 Rate Monotonic (RM) Scheduling Algorithm

The *rate-monotonic (RM)* scheduling algorithm is a static-priority algorithm which assigns priorities to tasks in inverse ratio of their periods, ties broken arbitrarily. So the highest priority is assigned to the task with smallest period. Table 1 shows the schedule produced for a task set from time $t = 0$ to time $t = 24$.

A *critical instant* for τ_i is an instant such that an instance released at that instant takes the longest time to finish execution. Liu and Layland[17] showed that for tasks scheduled using RM, the critical instant

for any instance occurs when the instance is released simultaneously with all the higher priority instances. One consequence of this is that schedulability of τ_{ij} for any j can be decided by releasing τ_i simultaneously with all the higher priority tasks. The reason why this holds is that the interference produced by a higher priority task on a lower priority instance can always be increased by advancing the release time of the higher priority task to be in phase with the release time of the lower priority task.

It was shown that the *RM algorithm is optimal among all static-priority scheduling algorithms*. This can be proved by taking a feasible static priority assignment and swapping the priorities until the priorities are in rate-monotonic order, such that during each swap, no task misses its deadline.

2.1.1 Schedulability tests

Sufficient test for schedulability : We first note that *while $U \leq 1$ is necessary for schedulability, it is not enough*. For example, consider the task set $\{(2, 5), (7, 12)\}$. Its utilization is $59/60$, but the first instance of $(7, 12)$ misses its deadline. In fact, no static priority algorithm can schedule this, because optimal scheduling would require dynamic decisions : a task with a more immediate deadline should be executed before a task with a farther deadline. Here, the first instance of $(7, 12)$ has its deadline at 12, but the algorithm schedules the third instance of $(2, 5)$ at $t = 10$.

Liu and Layland[17] derived a sufficient test for schedulability based on the concept of *a task set fully-utilizing the processor*. Given a priority assignment, a task set is said to fully-utilize the processor if it is schedulable, and if increasing the computation time of *any* task renders the task set unschedulable. It was then proved that any task set which fully utilizes a processor must have a utilization at least $n(2^{1/n} - 1)$, where n is the number of tasks. Based on the property that the bound is a decreasing function of n , and that any schedulable task set can be converted into a task set that fully-utilizes the processor by increasing the computation time of the lowest priority task, it can be proved that any task set whose utilization is less than this bound is schedulable using RM[9]. That is, a task set is schedulable using RM if it satisfies the following condition :

$$\sum_{i=1}^n U_i \leq n(2^{1/n} - 1). \quad (3)$$

We refer to (3) as the LL-test for schedulability and the bound as the LL-bound. A scheduler using the LL-test can decide the schedulability in $O(n)$ time. For example, consider the task set $\{(1, 2), (1, 20), (1, 10)\}$. The utilization of this task set is 0.65 while the LL-bound for $n = 3$ is ≥ 0.779 . Thus, this task set is clearly schedulable by RM. Note that the example of table 1 is schedulable by RM although it would be declared as unschedulable by the LL-test. The task set $\{(2, 5), (7, 12)\}$ is declared as unschedulable by the LL-test, and is actually unschedulable by RM.

Now, the LL-bound has a minimum value of $\ln 2$ attained at $n = \infty$. This value is slightly greater than 0.69 : therefore, *any task set regardless of the number of tasks in the set is schedulable by RM, if its utilization is less than 0.69*.

Exact test for schedulability : To our knowledge, it is not known if there is a polynomial time exact test for schedulability, that is, the test concludes that a task set is schedulable if and only if it is actually schedulable. We discuss below two pseudo-polynomial exact tests for schedulability.

Lehoczky-Sha-Ding (LSD) test for schedulability (Lehoczky et al.[16]) : This test is based on the following fact.

Property : Let the first instance of τ_i be released along with all the higher priority tasks at time $t = 0$. Then, τ_{i1} is schedulable iff there exists an instant $t \leq T_i$ such that the computation time of all the higher priority tasks released before t plus C_i is less than or equal to t .

Proof : The sufficiency of the condition is obvious. For the necessary part, observe that if any higher priority task is released before τ_{i1} , then the scheduler preempts τ_{i1} and proceeds to execute the higher priority

instance to completion. Thus, if τ_{i1} completes at t , that means all higher priority instances released before t have been completed by t . \square

Now, further observe that if there is any such instant t , then any instant $t' > t$ such that there is no task released in the interval $[t, t')$ is also an instant with the above property. More specifically, observe that the first instant $t' \geq t$ such that a task is released at t' has the above property. Hence, *if the task is schedulable, then the above property holds at any instant $t' \leq f_i$ such that a higher priority task is released at t'* . The based based on this property is stated formally below.

Let

$$S_i = \left\{ r T_j \mid j = 1, \dots, i; r = 1, \dots, \left\lfloor \frac{T_i}{T_j} \right\rfloor \right\}. \quad (4)$$

A task set is schedulable if and only if it satisfies the following condition :

$$\forall i : L_i = \left(\min_{\{t \in S_i\}} \frac{\sum_{j=1}^i \left\lfloor \frac{t}{T_j} \right\rfloor C_j}{t} \right) \leq 1. \quad (5)$$

We refer to (5) as the *LSD-test* for schedulability. Clearly, the time complexity of this test is pseudo-polynomial in its input.

For example, consider the task set $\{(3, 6), (1, 8), (4, 12)\}$ discussed above which the LL-test decides as unschedulable. The S_i values for $i = 1, 2, 3$ are respectively $\{T_1\}, \{T_1, T_2\}, \{T_1, 2T_1, T_2, T_3\}$. The L_i values for $i = 1, 2, 3$ are respectively $1/2, 2/3, 7/12$, and each of them is less than 1.

Response-Time-Analysis Test for schedulability (Audsley et al.[8]) : This test uses the same principle as in the LSD-test, except that it explicitly calculates the response time of the first instance released at the critical instant, and then checks to see if $\forall i : f_i \leq D_i$. This test guesses the response time initially to be equal to C_i . It then corrects the guess by adding the computation times of all the tasks released before the current guess to C_i . It repeats this process until the sum is exactly equal to the current guess.

Formally, the response time R_i is calculated using an iterative approach, as a fixed-point solution to the following equation :

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{R_i^k}{T_j} \right\rfloor C_j \quad (6)$$

where $R_i^0 = C_i$. The fixed point is reached when $R_i^{k+1} = R_i^k$. We refer to (6) as the *RTA (Response-Time-Analysis) test* for schedulability.

For example, for the task $\tau_3 = (4, 12)$ in the above task set, $R_2^0 = 4, R_2^1 = 8, R_2^2 = 11, R_2^3 = 12, R_2^4 = 12$. Thus, after four iterations, f_{31} is rightly calculated to be 12, as can be seen in table 1.

We may note here that if J_i is not zero for all tasks, then RM is provably not optimal[10].

2.2 Deadline Monotonic Scheduling (DM)

Deadline monotonic scheduling algorithm is a static priority scheduling algorithm, which is an extension of RM to deal with task sets where $D_i \leq T_i$. It schedules tasks in the inverse ratio of their *relative deadlines* D_i , regardless of their computation times and periods.

As an example, consider the task set $\{(3, 8, 6), (1, 10, 4), (4, 16, 12)\}$, Here, τ_2 has a shorter relative deadline than τ_1 . The schedule for this task from $t = 0$ to $t = 24$ is $2^+111^+3333^+112^+1^+xxxx111^+2^+333$ (the notation is explained in table 1; x means the processor is idle in that slot).

The DM algorithm is optimal among all static priority algorithms where $D_i \leq T_i$. The schedulability tests for DM algorithm are similar to those of RM.

We may note here that if J_i is not zero for all tasks, then determining an optimal priority assignment is NP-hard in the strong sense, that is, even a pseudo-polynomial time algorithm does not exist unless P = NP.

2.3 Earliest Deadline First (EDF) Scheduling Algorithm

The *earliest-deadline-first (EDF)* scheduling algorithm is a dynamic priority algorithm. When either an instance finishes execution or a new instance arrives, the EDF algorithm schedules that instance whose *absolute* deadline is the earliest among all the active instances. Liu and Layland[17] showed that EDF is optimal among all dynamic priority scheduling algorithms.

Exact test for schedulability : Liu and Layland[17] also derived a necessary and sufficient test for schedulability : τ is schedulable iff

$$\sum_{i=1}^n U_i \leq 1. \quad (7)$$

This test runs in time $O(n)$. Table 1 shows the EDF schedule for a task set. It may be interesting to note here that *the earliest deadline first algorithm is optimal even in the non-preemptive case*[12].

2.4 Processor Sharing Schedule (PSS)

The *processor sharing schedule* (see [18]) is an idealized scheduling algorithm which during any instant assigns a fraction U_i to task τ_i . Clearly, any task set satisfying $U \leq 1$ is schedulable by this algorithm, although, it might be almost impossible to implement in practice. This process model could be useful for theoretical results.

2.5 Scheduling Aperiodic Tasks along with Periodic Tasks

Algorithms for scheduling aperiodic tasks along with periodic tasks are well-known. They are real-time-scheduling algorithms which schedule a given *periodic* task set using some *static* priority algorithm (RM in this paper) and also schedule *aperiodic* tasks in a soft manner, that is, they attempt to minimize the response time of the aperiodic requests. An *aperiodic* task τ_a is characterized by (C_a, D_a, J_a) , where C_a is the execution time, D_a is the absolute deadline, and J_a is the release time of the task. For such algorithms, the characteristics of the periodic task set are fully known, whereas those of the aperiodic task set are unknown. All aperiodic requests are queued in FIFO order for processing. Implementation complexity and aperiodic responsiveness are two important measures to evaluate the effectiveness of these algorithms.

We first mention two natural approaches to scheduling aperiodic tasks softly. The first approach called *background scheduling* schedules the aperiodic tasks whenever there is no periodic task running. This naive approach, though it can utilize all the free time available, suffers from bad aperiodic responsiveness. The second approach is called the *slack-stealing* algorithm which utilizes the fact that a hard task instance can be completed anytime before its deadline; in particular, there is no advantage in finishing it early and so its execution can be delayed as long as it is completed before its deadline. This approach utilizes all free time and has very good aperiodic responsiveness, but is clearly difficult to implement since whenever an aperiodic task arrives, the scheduler would need to calculate the required slack-intervals, and this could entail forming the exact periodic schedule for some interval of time.

Bandwidth preserving server is an approach to have reasonably good aperiodic responsiveness, and also low implementation complexity. In this approach, there is an additional periodic task $\tau_s = (C_s, T_s, J_s)$ called the *server task*, where C_s is the maximum available *bandwidth* or *capacity*, T_s is the time period and J_s is the release time of the server. While the periodic tasks are scheduled in the rate-monotonic order, generally, the server has the highest priority regardless of its period. These are also known as **fixed-priority aperiodic servers**, since the priority of the periodic tasks is statically determined (RM). Different algorithms differ in the rules governing consumption and replenishment of the bandwidth.

Two well-known algorithms are the *deferrable server* introduced by Strosnider, Lehoczky, and Sha[14][13], and the *sporadic server* introduced by Sprunt, Sha, and Lehoczky [23].

Let $U_p = \sum_{i=1}^n U_i$ and $U_s = \frac{C_s}{T_s}$. We demonstrate below the functioning of the deferrable server and the sporadic server with the help of an example.

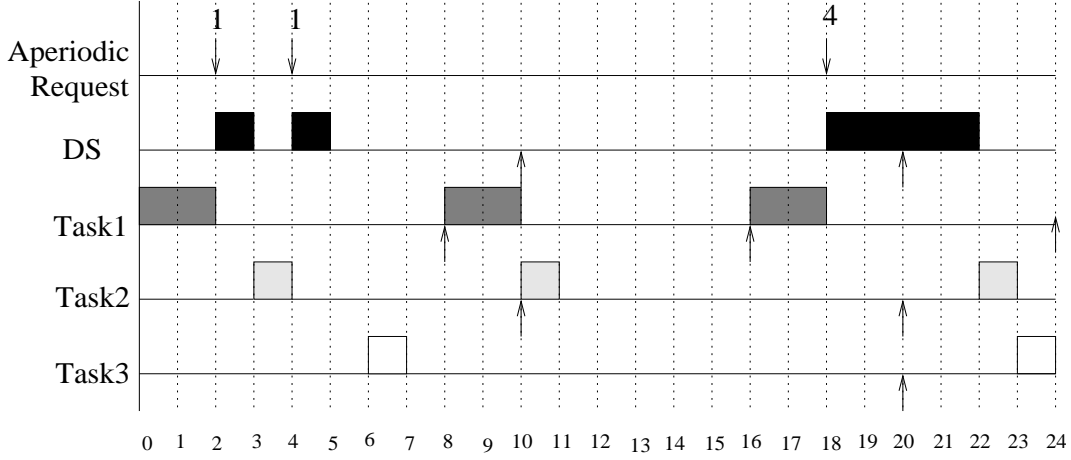


Figure 1: Deferrable server operation

2.5.1 Deferrable server

The operation of the server task in the **deferrable server** obeys the following consumption and replenishment rules :

- After its release, in any period, as long as its capacity is non-zero, τ_s executes aperiodic tasks, if any. The server cannot execute any aperiodic task if its capacity is zero.
- Regardless of the capacity used in a period by aperiodic tasks, the capacity of τ_s is increased to be C_s at the beginning of the next period.

For example, consider the periodic task set $\{(2, 8), (1, 10), (1, 20)\}$, where the server $\tau_s = (2, 10)$. Figure 1 shows a series of aperiodic requests, and the operation of deferrable server (DS). The arrows indicate the beginning of a period or the arrival of an aperiodic task. Since there is no aperiodic request at time $t = 0$, DS is idle, and lets τ_1 execute. The first aperiodic request arrives at time $t = 2$ with execution time 1, and the DS serves it for one time unit and then becomes idle. At time $t = 4$, another similar aperiodic request arrives, and DS serves the task. After that, DS capacity is extinguished for the first time period. Again at time $t = 10$, the DS capacity is replenished to 2 units, but DS becomes idle since there is no aperiodic request. At time $t = 18$ an aperiodic request of 4 units of execution arrives, and DS executes from $t = 18$ to $t = 20$ for 2 time units. At time $t = 20$ its capacity is replenished to 2 units, and DS continues to execute till time $t = 22$.

Some of the key properties discussed by Buttazzo[7] are :

- The server can execute consecutively for $2C_s$ time units. This can be seen in figure 1 where the DS executes from $t = 18$ to $t = 22$. Therefore, the critical instant for a periodic task w.r.t the deferrable server occurs when the task is released at the start of a double-hit, and then the DS executes for its fully capacity at the beginning of every period.
- A sufficient test, derived using the above fact, for determining the server utilization is

$$U_s \leq \frac{2 - e^{U_p}}{2e^{U_p} - 1} \quad (8)$$

For eg., in the above task set, $U_p = 0.4$. The corresponding bound for U_s is 0.256.

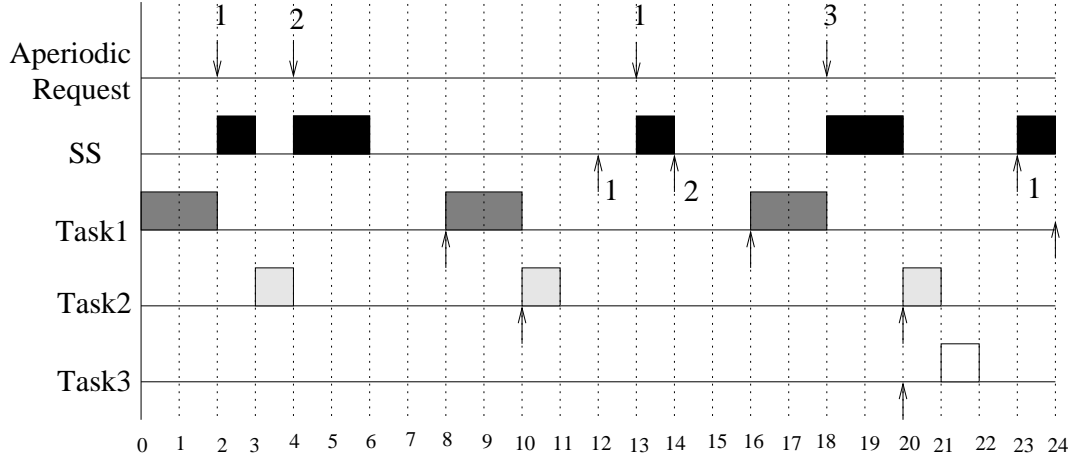


Figure 2: Sporadic server operation

2.5.2 Sporadic server

The operation of the server task in the **sporadic server** is given by the following consumption and replenishment rules :

- After its release, τ_s is *idle* if its capacity is zero, *or* it has a non-zero capacity, but there are no aperiodic requests.
- If it is idle with non-zero capacity, and an aperiodic task arrives at time t , then the server becomes active at t , and the *next replenishment time* of the server is decided to be $t + T_s$.
- After becoming active at time t , suppose the server executes continuously from t to t' . Then the amount of replenishment at time $t + T_s$ is $t' - t$. Note that the maximum value of $t' - t$ is C_s .
- If the server is idle with zero capacity, and an aperiodic task arrives, the server has to wait till the next replenishment time before it starts serving the task.

Thus, the sporadic server can have several points at which its capacity is replenished and the amounts can vary. Note that the server capacity is never more than C_s at any point in time.

For example, consider the same task set $\{(2, 8), (1, 10), (1, 20)\}$ with server task $\tau_s = (3, 10)$, whose operation for a series of aperiodic requests is shown in figure 2. There is no aperiodic request at time $t = 0$ and so the SS becomes idle, while allowing τ_1 to execute. At $t = 2$, a request arrives with 1 unit requirement. SS serves the request, and a replenishment instant at time $t = 2 + 10 = 12$ is scheduled, with 1 unit replenishment. Similarly, at time $t = 4$ a task arrives with 2 units requirement arrives, and a replenishment of 2 unit is scheduled for time $t = 4 + 10 = 14$. The fourth aperiodic request of 3 units at $t = 18$ can only be served for 2 units because the sporadic server does not have enough capacity. It must wait till $t = 23$ to get a replenishment of 1 unit. It then completes the task at $t = 24$.

Some of the key properties of the sporadic server discussed by Buttazzo [7] are :

- The sporadic server can be treated as a regular periodic task for schedulability purposes. This is because the worst case interference that a sporadic task can produce is when it is invoked as quickly as possible, and to its full capacity.
- A sufficient schedulability test for the periodic tasks is

$$U_s \leq \frac{2}{e^{U_p}} - 1 \quad (9)$$

For eg., for the above task set, the corresponding bound for U_s is 0.34.

3 The Space of Rate-Monotonic Schedulability

In this section, we study the results derived by Bini and Buttazzo [6], and discuss some of their implications for deciding real-time schedulability. The paper gives a new exact schedulability test for the RM algorithm. This new test explores only a subset of S_i defined in (4) for deciding schedulability of τ_i , and the number of points explored is independent of (C_i, T_i, J_i) for all i - however, the test is exponential in n . Furthermore, the test can also be tuned via a parameter to derive a sufficient test for schedulability similar to the LL-test, but whose bound on the RHS of (3) can be greater than $n(2^{1/n} - 1)$. The bound depends on the tuning parameter used, and the higher the bound that is desired, the greater the computational complexity of the test.

In the following discussion, all tasks are assumed to be released at their critical instants, that is, $\forall i : J_i = 0$. Further, let τ_{ij} be active in (r_{ij}, f_{ij}) instead of $[r_{ij}, f_{ij}]$ as it makes the discussion easier. Let $\Gamma_i = \{\tau_1, \dots, \tau_i\}$ be the set of the first i tasks. We will be using the task set $\tau_{eg} = \{(1, 5), (2, 10), (5, 25), (29, 80)\}$ as a running example for all illustrations in this section. This task set is schedulable using RM, but its utilization 0.9625 is greater than 0.7568, the schedulability bound for 4 tasks using LL-test.

We say the processor is *i-active* at time t if any task of Γ_i is active at t , that is, has been released but has not yet completed execution. For eg., first instance of $(29, 80)$ finishes at $t = 75$; so the processor is 4-active in $(0, 75)$ but not 4-active in $[75, 80]$. Let $W_i(b)$ denote the *total amount of time the processor is i-active in the interval $[0, b]$* . For eg., $W_4(80) = 75$. Similarly, it can be verified that $W_3(10) = 9$. By definition, $0 \leq W_i(b) \leq b$.

Let $\psi_i(b)$ denote the **last instant in $[0, b]$ at which the processor is **not** *i-active***, that is, all the tasks released before $\psi_i(b)$ have been executed by $\psi_i(b)$, and the processor is *i-active* throughout the interval $(\psi_i(b), b]$. For example, the processor becomes free for the first time after $t = 0$ at $t = 75$. Therefore, for all $0 < b < 75$, the processor is 4-active, and so $\psi_4(b)$ for such b is $t = 0$. For $b = 75$, it is 75 since the processor is idle at 75. For $b = 76$, it is 75. Since the intervals during which the processor is busy are all open intervals, $\psi_i(b)$ is well-defined.

We first observe that

$$\forall t \in [0, b] : W_i(b) \leq \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t) \quad (10)$$

The above equation holds because the total amount of *i-active* time in $[t, b]$ is upper bounded by the interval length $(b - t)$, and the *i-active* time in $[0, t]$ can't exceed the sum of the computation times of the tasks released before t . Now, further observe that

$$W_i(b) = \sum_{j=1}^i \left\lceil \frac{\psi_i(b)}{T_j} \right\rceil C_j + (b - \psi_i(b)) \quad (11)$$

Equation (11) holds because $\psi_i(b) \in [0, b]$, the amount of time processor is *i-active* after $\psi_i(b)$ is exactly $(b - \psi_i(b))$ by definition of $\psi_i(b)$, and the amount of time processor is *i-active* before $\psi_i(b)$ is exactly the sum of the computation times of all tasks released before $\psi_i(b)$ because otherwise the processor cannot be idle at $\psi_i(b)$.

Now, τ is schedulable if and only if Γ_i is schedulable for all i , and Γ_i is schedulable if and only if $C_i + W_{i-1}(T_i) \leq T_i$ for all i . This is just rephrasing the RTA-test(6). From (11), all that is required now is to find $\psi_i(b)$. The following lemma helps us to find $\psi_i(b)$:

Lemma 3.1 *Let Γ_i be schedulable by RM and let $\mathcal{P}_i(b)$ be defined as follows :*

$$\begin{cases} \mathcal{P}_0(b) = \{b\} \\ \mathcal{P}_i(b) = \mathcal{P}_{i-1} \left(\left\lfloor \frac{b}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(b) \end{cases}$$

Then

$$\psi_i(b) \in \mathcal{P}_i(b).$$

Proof (from [6]): We prove this by induction on i .

Initial Step : We have to prove that $\psi_1(b) \in \mathcal{P}_1(b)$ for all b . Now,

$$\mathcal{P}_1(b) = \mathcal{P}_0 \left(\left\lfloor \frac{b}{T_1} \right\rfloor T_1 \right) \cup \mathcal{P}_0(b) = \left\{ \left\lfloor \frac{b}{T_1} \right\rfloor T_1, b \right\}.$$

Since τ_1 is schedulable, $\psi_1(b)$ is either

- $\left\lfloor \frac{b}{T_1} \right\rfloor T_1$, if the last instance of τ_1 in $[0, b]$ is active at b , or
- b .

Thus, $\psi_1(b) \in \mathcal{P}_1(b)$.

Induction Step : If $\psi_i(b) \in \mathcal{P}_i(b)$ for all b , we have to prove that, given a schedulable set Γ_{i+1} , $\psi_{i+1}(b) \in \mathcal{P}_{i+1}(b)$ for all b .

Consider the time interval $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$. In this interval, two things can happen :

1. *The processor is $(i+1)$ -active throughout the interval :* Then $\psi_{i+1}(b) = \psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1})$ because the processor is $(i+1)$ -active in $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$. Now, for any x , $\psi_{i+1}(x) \leq \psi_i(x)$, because Γ_{i+1} has one additional task compared to Γ_i . Further, it must be the case that $\psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1}) = \psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$. Suppose, on the contrary, it is a strict inequality. Then, the last instance of τ_{i+1} would miss its deadline at $\lfloor b/T_{i+1} \rfloor T_{i+1}$ contradicting the fact that τ_{i+1} is schedulable.
2. *There exists an instant of time at which the processor is not $(i+1)$ -active :* Let $x \in [\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$ be an instant of time where no tasks in Γ_{i+1} are active. Since at time x the $(\lfloor b/T_{i+1} \rfloor + 1)^{th}$ job of τ_{i+1} has finished execution, τ_{i+1} is never active in $[x, b]$. This implies that $\psi_{i+1}(b) = \psi_i(b)$.

Since by induction hypothesis, $\psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$ and $\psi_i(b) \in \mathcal{P}_i(b)$,

$$\psi_{i+1}(b) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \cup \mathcal{P}_i(b) = \mathcal{P}_{i+1}(b) \text{ by definition.}$$

□

From lemma 6.1, it is easy to see that the exact-test for schedulability (5) can now be written as :

$$\forall i : L_i = \left(\min_{\{t \in \mathcal{P}_i(T_i)\}} \frac{\sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \right) \leq 1. \quad (12)$$

The authors call this the HET-test. It follows from the definition of $\mathcal{P}_i(b)$ that $\mathcal{P}_i(T_i) \subseteq S_i$. Further, it can be verified that the total number of points in $\mathcal{P}_i(T_i)$ is $\leq 2^{i-1}$. Thus, verifying (12) would cost at most $O(2^n)$. Therefore, the time complexity of deciding schedulability is changed from pseudo-polynomial to exponential. Clearly, while the HET test would always perform better than the traditional exact test (or the RTA-test), the difference between the two tests would be more visible in task sets where $\lfloor T_i/T_j \rfloor$ is large for several pairs (i, j) . For example, consider the task set in table 2. The table shows that the number of points explored by (5) is 40, whereas that by the HET test is only 15.

Test used	Number of points explored	Schedulability result
LSD-exact test	$\sum_{i=1}^{i=4} S_i = 40$	Schedulable
HET	$\sum_{i=1}^{i=4} P_i(T_i) = 15$	Schedulable
0.2-HET	$\sum_{i=1}^{i=4} P_i(T_i, \delta = 0.2) = 7$	Not schedulable
0.3-HET	$\sum_{i=1}^{i=4} P_i(T_i, \delta = 0.3) = 8$	Not schedulable
0.39-HET	$\sum_{i=1}^{i=4} P_i(T_i, \delta = 0.39) = 12$	Schedulable

Table 2: Number of points explored and schedulability results for $\{(1, 4), (2, 10), (5, 25), (29, 80)\}$. The utilization of the task set of the task set is **0.9625.**

3.1 The δ -HET Test

Bini and Buttazzo tune the HET-test using a parameter $0 < \delta \leq 1$ as follows :

$$\begin{cases} \mathcal{P}_0(b, \delta) = b & \forall \delta \\ \mathcal{P}_i(b, \delta) = \begin{cases} \mathcal{P}_{i-1} \left(\left\lfloor \frac{b}{T_i} \right\rfloor T_i, \delta \right) \cup \mathcal{P}_{i-1}(b, \delta) & \text{if } b\delta \geq T_i \\ \mathcal{P}_{i-1} \left(\left\lfloor \frac{b}{T_i} \right\rfloor T_i, \delta \right) & \text{otherwise} \end{cases} \end{cases}$$

The authors call this test the δ -HET test. This test throws away the search points for $T_i > b\delta$, and retains them if $b\delta \geq T_i$. We note from (12) that each $t \in P_i(b)$ represents a region in an n -dimensional space, namely the region given by $\sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t$, where the n dimensions are the n possible C_j values. Clearly, since we want at least one inequality to be satisfied, the space of valid C_j tuples is represented by the union of those regions. If we discard some of the regions, then we get a sufficient schedulability test. The above test discards some of the regions based on δ .

The following properties can be easily observed :

- $\delta_1 \leq \delta_2 \iff \mathcal{P}_i(b, \delta_1) \subseteq \mathcal{P}_i(b, \delta_2)$. Thus, by choosing a larger δ , we can increase the set of points to be explored.
- Define

$$W_i^{(\delta)}(b) = \min_{t \in \mathcal{P}_i(b, \delta)} \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t). \quad (13)$$

From (10), and from the above observation, it is clear that $W_i^{(\delta_1)}(b) \geq W_i^{(\delta_2)}(b) \iff \delta_1 \leq \delta_2$ and $\forall \delta : W_i^{(1)}(b) = W_i(b) \leq W_i^{(\delta)}(b)$.

The δ -HET test is designed to guarantee more tasks as schedulable if a larger δ is chosen. If $\delta = 1$, then δ -HET test is the HET-test. Note that the tuning is based on the actual time period of the tasks, and thus will vary for different task sets having the same number of tasks.

For example, table 2 shows the number of points explored for different values of δ and the corresponding schedulability results. As delta is increased, it can be seen that the number of points increases. While the test declares the task set to be unschedulable for $\delta < 0.3$, it rightly declares it schedulable for $\delta = 0.39$ though exploring smaller number of points than the HET test.

3.2 Performance of HET and δ -HET

We discuss three experiments used to evaluate the performance of HET and δ -HET.

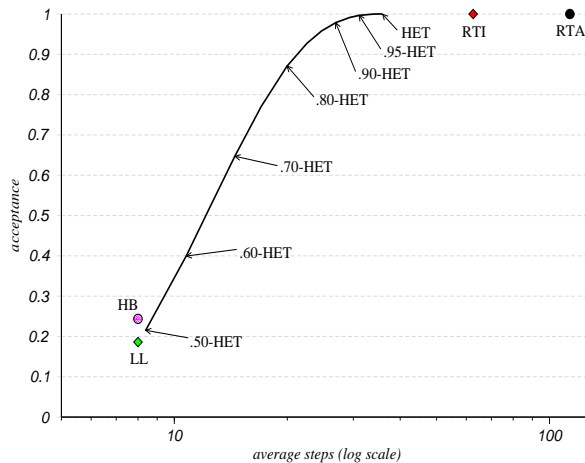


Figure 3: Acceptance as a function of average number of steps taken by δ -HET (log scale)

1. The first test evaluates HET against two other exact-schedulability tests : the RTA-test and an improved version of the RTA-test which they call the RTI-test[21]. Simulations are performed using 10^8 task sets, each composed of 8 tasks. The periods of the tasks are chosen uniformly from $[0, 1000000]$ and the computation times C_i are chosen uniformly from $[0, T_i]$. The results are displayed as a probability density function, of the number of steps.
2. The second test measures the average number of steps taken by RTA, RTI and HET, as the number of tasks in the set was varied. While the complexity of HET grows exponentially as the number of tasks is increased, its growth rate is much smaller than that of RTA and RTI. This is because the HET test is independent of the actual periods, while the RTA test is.
3. The third test plots the acceptance ratio as a function of average number of steps taken by δ -HET. The acceptance ratio of a test is

$$\frac{\text{No. of tasks decided as schedulable by the test}}{\text{No. of tasks decided as schedulable by an exact test}}$$

The plot, taken from the paper itself[6], is shown in figure 3¹.

The δ -HET test performs better than the LL-test for $\delta \geq 0.5$. As δ increases, the acceptance also increases steadily, and the acceptance for $\delta = 1$ is 1 as expected. As can be seen in the figure, a wide-range of acceptance can be achieved by varying δ .

4 Static-Priority Scheduling on Multiprocessors

In this section, we study the results derived by Andersson, Baruah and Jonsson[2]. We first give some background of multiprocessor scheduling, state two preliminary results that we will use in the proof for schedulability, and then study a static-priority scheduling algorithm DM-US $[m/(3m-2)]$, which is a slightly adapted version of the RM-US $[m/(3m-2)]$ algorithm[2]. We then show that the algorithm is also easily adaptable to derive a dynamic priority scheduling algorithm EDF-US.

4.1 Multiprocessor Scheduling : Background

The system consists of $m (\geq 2)$ identical multiprocessors, and a set τ of n periodic tasks. Instances can be preempted, as in the case of uniprocessor systems. It is assumed that there is no interaction between

¹The HB-test (Hyperbolic Bound test) is another sufficient schedulability test[5].

instances on different processors. A *run-time* multiprocessor scheduler, at each instant, assigns a priority to all the active instances, and allocates the available processors to the instances having the highest priority. Run-time scheduling algorithms can be classified into two types : *partitioned* scheduling algorithms and *global* scheduling algorithms. A partitioned scheduling algorithm assigns tasks to processors, and all instances of a task should be executed on the assigned processor. In contrast, a global scheduling algorithm can schedule an instance on any available processor. However, instances are not allowed to be parallelized.

A *static* priority run-time algorithm assigns priorities to *tasks*, and all instances of a task have the same priority as the task. A *dynamic* priority algorithm assigns priorities to *instances* of tasks. An example of a global static-priority scheduling algorithm is the multiprocessor RM algorithm which assigns priorities to tasks rate-monotonically, and at each instant executes the m highest priority instances on each of the m processors. An example of a global dynamic-priority scheduling algorithm is the multiprocessor EDF algorithm, which, at any instant, schedules m instances having the earliest deadline among all instances, on each of the m processors².

4.2 Two Preliminary Results

We briefly discuss two results used by the authors in their paper.

4.2.1 Resource augmentation

Let $I = \{I_j \mid j > 0\}$ denote any (not necessarily finite) set of instances. For any algorithm A and time instant $t \geq 0$, let $W(A, m, s, I_j, t)$ denote the amount of computation that has been performed by A on instance I_j in the time interval $[0, t]$ on m processors each running at a speed of s per time unit. Clearly, $0 \leq W(A, m, s, I_j, t) \leq C_j$. Let $W(A, m, s, I, t) = \sum_j W(A, m, s, I_j, t)$. Define a **work-conserving** algorithm to be an algorithm which never idles a processor if an instance is active. Then the following theorem, first discussed by Phillips et al.[20] and slightly adapted by the Andersson et al. [2], holds:

Theorem 4.1 *For any set of instances I , any time instant $t \geq 0$, any work-conserving algorithm A , and any algorithm (in particular, whether work-conserving or not) A' , it is the case that*

$$W(A, m, (2 - \frac{1}{m}) \cdot s, I, t) \geq W(A', m, s, I, t).$$

Discussion : First note that two work-conserving algorithms need not do the same amount of work by time t . For example, consider a set of 3 tasks, all released at time $t = 0$ with computation times 1, 2 and 3 respectively, to be executed on two processors. One algorithm assigns priorities as 1, 2 and 3 respectively, while another algorithm assigns them priorities as 1, 3 and 2 respectively. While the work done by the second algorithm by $t = 3$ is 6, the work done by the first algorithm is 5. This means all work-conserving algorithms need not do the same work by time t . However, if a work-conserving algorithm is provided with processors $(2 - 1/m)$ times faster processors, then regardless of the scheduling strategy used, it can beat any other work-conserving algorithm. Since it holds for any set of instances, it also holds for periodic instances.

The proof of this theorem can be found in a technical report[1] by Andersson et al. It basically relies on the restriction that parallelization of tasks is disallowed.

4.2.2 Predictability of a scheduling algorithm

Let $I = \{I_j \mid j > 0, I_j = (C_j, D_j, J_j)\}$ be a set of instances with execution time **exactly** C_j , deadline D_j and release time R_j . Let A being an algorithm scheduling I , and let the finishing time of I_j be f_j . Let $I' = \{I_j \mid j > 0, I_j = (C'_j, D_j, J_j)\}$ be a set of instances derived from I such that $\forall j : C'_j \leq C_j$. Then A is said to be **predictable** if $\forall j : f'_j \leq f_j$. In other words, if a task executes for a time less than its worst case execution time, then its finishing time will be no later than its finishing time had it executed for its worst-case duration.

²If there are $l < m$ instances active, then there is no priority assignment necessary

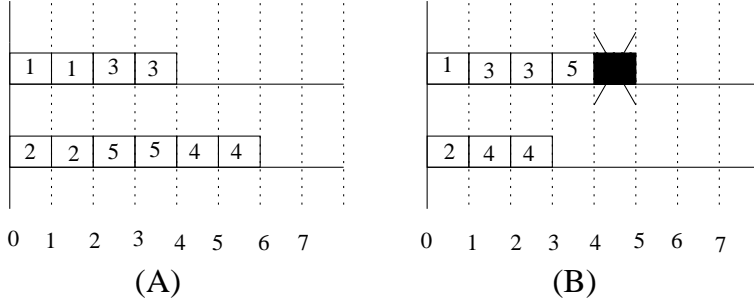


Figure 4: Predictability does not hold when preemption is disallowed

An algorithm A is said to be **priority-driven** if it satisfies the condition that for any two instances I_j, I_k , if at *some* point of time I_j has a higher priority than I_k , then I_j has a higher priority than I_k at *any* point of time. In other words, once the *relative* priority of an instance is decided, it is never changed during execution. For example, RM and DM are priority-driven static-priority algorithms, whereas EDF is a priority-driven dynamic-priority algorithm. An example of an algorithm that is *not* priority-driven is the *least-slack-time-first* algorithm[19].

The following theorem by Ha and Liu[11] holds:

Theorem 4.2 *Any priority-driven scheduling algorithm is predictable.*

Discussion : Note that the theorem does not hold if preemption were disallowed. For example, consider the task set $\{(2, 3, 3, 0), (2, 3, 3, 0), (2, 4, 4, 0), (2, 6, 1), (2, 4, 2)\}$ scheduling using EDF. The task set is schedulable. However, if τ_1 and τ_2 execute for only 1 time unit instead of 2, τ_5 misses its deadline at $t = 4$. This is shown in figure 4.

4.3 Algorithm DM-US $[m/(3m-2)]$

Andersson et al.[2] present an algorithm called RM-US $[m/(3m - 2)]$, which can schedule any τ such that $U \leq m^2/(3m - 2)$. We now present the algorithm DM-US $[m/(3m - 2)]$, which is a slightly adapted version of RM-US $[m/(3m - 2)]$ and derive a utilization based test which guarantees schedulability of any task set τ satisfying $U^d \leq m^2/(3m - 2)$. Our proof strategy and presentations will parallel those by Andersson et al.[2]. We will first prove that a special category of the above task sets, which we call “light sets”, are schedulable by multiprocessor DM algorithm, which is same as the multiprocessor RM algorithm, except that it prioritizes the tasks by their relative deadlines D_i . We then define the DM-US $[m/(3m - 2)]$ algorithm, and proceed to derive the utilization based test.

4.3.1 Light sets

Definition 4.1 Light set : *A task set τ is said to be a “light set” if it satisfies the following properties:*

Property 1 : $U_i^d \leq \frac{m}{3m-2}$

Property 2 : $U^d \leq \frac{m^2}{3m-2}$

Let τ be a “light set” of tasks, and let $\tau^{(k)} = \{\tau_1, \dots, \tau_k\}$ be the set of the k highest priority tasks of τ . Clearly, $\tau^{(k)}$ is a “light set”. We will prove that $\tau^{(k)}$ is schedulable for any k , and therefore τ is schedulable.

We first prove a lemma, which would help us in proving the schedulability of $\tau^{(k)}$ by algorithm DM.

Lemma 4.1 *The task set $\tau^{(k)}$ is schedulable on a set of m processors each of computing capacity $\left(\frac{m}{2m-1}\right)$.*

Proof : First note that since $3m - 2 > 2m - 1$, and by property 1,

$$U_i^d \leq \frac{m}{2m - 1}$$

and therefore, since $U_i \geq U_i^d$,

$$U_i \leq \frac{m}{2m - 1} \tag{14}$$

Further, note that by property 2,

$$\sum_{\tau_i \in \tau^{(k)}} U_i^d \leq \frac{m^2}{2m - 1}$$

and therefore

$$\sum_{\tau_i \in \tau^{(k)}} U_i \leq \frac{m^2}{2m - 1}. \tag{15}$$

Since a processor-sharing schedule (PSS) can schedule any set of instances, each of whose utilization U_i is ≤ 1 and the sum of whose utilizations U is $\leq m$, on a set of m unit-speed processors, by (14) and (15), τ can be scheduled on $m \left(\frac{m}{2m-1}\right)$ -speed processor by using PSS. \square

Now, since the algorithm DM is work-conserving,

$$W(\text{DM}, m, (2 - 1/m), \tau^{(k)}, t) \geq W(\text{PSS}, m, 1, \tau^{(k)}, t).$$

In other words

$$W(\text{DM}, m, 1, \tau^{(k)}, t) \geq W(\text{PSS}, m, \frac{m}{2m - 1}, \tau^{(k)}, t) \tag{16}$$

for all $t \geq 0$. Thus, *algorithm DM, on m unit-speed processors, performs at least as much work on $\tau^{(k)}$ as the work done by PSS on $m \frac{m}{2m-1}$ -speed processors.*

We now prove the key lemma of the paper :

Lemma 4.2 $\tau^{(k)}$ is schedulable by DM on m unit speed processors.

Proof : The proof here is a replica of the proof by Andersson et al.[2], adapted for the DM case.

Let us assume that the first $(l - 1)$ instances of τ_k have met their deadlines under the algorithm DM; we will prove that the l^{th} instance of τ_k also meets its deadline. The lemma will then follow by induction starting from $l = 1$.

The l^{th} instance of τ_k arrives at time-instant $(l - 1)T_k$, has a deadline at time-instant $(l - 1)T_k + D_k$, and needs C_k units of execution. The **key idea** is to use the resource augmentation theorem to establish a lower bound on the amount of work done on the higher priority tasks till time $(l - 1)T_k$, and thus derive an upper bound on the amount of work to be done in the l^{th} interval. This leads a lower bound on the time available to process the l^{th} instance of τ_k . It is sufficient if the lower bound is at least C_k .

From (16), and the fact that PSS schedules each task τ_j for $(l - 1)T_k \cdot U_j$ units over the interval $[0, (l - 1)T_k)$, we have

$$W(\text{DM}, m, 1, \tau^{(k)}, (l - 1)T_k) \geq (l - 1)T_k \left(\sum_{j=1}^k U_j \right). \tag{17}$$

Also at least $(l-1) \cdot T_k \cdot \left(\sum_{j=1}^{k-1} U_j \right)$ units of this execution by Algorithm DM was of tasks in $\tau^{(k-1)}$ because *exactly* $(l-1)T_k U_k$ units of work was generated by τ_k prior to $(l-1)T_k$ - so the remainder of the work must have been generated by $\tau^{(k-1)}$.

Now, the cumulative execution requirement of all instances generated by $\tau^{(k-1)}$ that arrive prior to $(l-1)T_k + D_k$, the deadline of τ_k 's l^{th} instance, is bounded from above by

$$\begin{aligned}
& \sum_{j=1}^{k-1} \left\lceil \frac{(l-1)T_k + D_k}{T_j} \right\rceil C_j \\
& \leq \sum_{j=1}^{k-1} \left\lceil \frac{lT_k}{T_j} \right\rceil C_j \quad (\text{since } D_k \leq T_k) \\
& \leq \sum_{j=1}^{k-1} \left(\frac{lT_k}{T_j} + 1 \right) C_j \\
& = lT_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \tag{18}
\end{aligned}$$

Since at least $(l-1)T_k \sum_{j=1}^{k-1} U_j$ amount of work gets done prior to $(l-1)T_k$, the amount of work to be done in $[(l-1)T_k, (l-1)T_k + D_k)$ on tasks in $\tau^{(k-1)}$ is at most

$$\left(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \right) \tag{19}$$

$$\leq \left(T_k \sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} C_j \right). \tag{20}$$

The last inequality holds because $U_j^d \geq U_j$. Therefore, the total processor capacity left unused by $\tau^{(k-1)}$ during the interval $[(l-1)T_k, (l-1)T_k + D_k)$ is at least

$$m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} C_j \right) \tag{21}$$

Since there are m processors available, the cumulative length of the intervals over $[(l-1)T_k, (l-1)T_k + D_k)$ during which $\tau^{(k-1)}$ leave at least one processor idle is minimized if the different processors tend to idle simultaneously (in parallel); hence, a lower bound on this cumulative length of the intervals over $[(l-1)T_k, (l-1)T_k + D_k)$ during which $\tau^{(k-1)}$ leave at least one processor idle is given by $\left(m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} C_j \right) \right) / m$, which equals

$$T_k - \frac{1}{m} \left(T_k \sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} C_j \right) \tag{22}$$

For the l^{th} instance of τ_k to meet its deadline, it suffices that this cumulative interval length be at least as large as τ_k 's execution requirement; that is,

$$T_k - \frac{1}{m} \left(T_k \sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} C_j \right) \geq C_k$$

$$\Leftrightarrow \frac{C_k}{T_k} + \frac{1}{m} \left(\sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} \frac{C_j}{T_k} \right) \leq 1,$$

and since $\forall j < k : T_k \geq T_j$, it is enough if

$$U_k + \frac{1}{m} \left(\sum_{j=1}^{k-1} U_j^d + \sum_{j=1}^{k-1} U_j \right) \leq 1.$$

Further, since $\forall j : U_j \leq U_j^d$, it is enough if

$$U_k^d + \frac{1}{m} \left(2 \sum_{j=1}^{k-1} U_j^d \right) \leq 1. \quad (23)$$

Now,

$$\begin{aligned} & U_k^d + \frac{1}{m} \left(2 \sum_{j=1}^{k-1} U_j^d \right) \\ &= U_k^d + \frac{1}{m} \left(2 \sum_{j=1}^k U_j^d - 2U_k^d \right) \\ &\leq (\text{By property 2}) \\ & U_k^d \left(1 - \frac{2}{m} \right) + \frac{2m}{3m-2} \\ &\leq (\text{By property 1}) \\ & \frac{m}{3m-2} \left(1 - \frac{2}{m} \right) + \frac{2m}{3m-2} \\ &= 1. \end{aligned}$$

□.

Therefore, since $\tau^{(k)}$ is schedulable for any k by algorithm DM, $\tau^{(n)} = \tau$, a “light set” is schedulable by algorithm DM.

4.3.2 Arbitrary sets

In this section, we relax property 1, and prove that any task system satisfying property 2 alone is schedulable by algorithm DM-US $[m/(3m-2)]$. Let τ be any task set satisfying

$$U^d \leq \frac{m^2}{3m-2}. \quad (24)$$

For such systems, the algorithm DM-US $[m/(3m-2)]$ assigns static priorities to tasks as follows :

- If $U_i^d > \frac{m}{3m-2}$, then τ_i has the highest priority (ties broken arbitrarily).
- If $U_i^d \leq \frac{m}{3m-2}$, then τ_i has deadline-monotonic priority.

Theorem 4.3 *Any task set τ satisfying $U^d \leq m^2/(3m-2)$ is schedulable by algorithm DM-US $[m/(3m-2)]$.*

Sketch of Proof : We first observe that since $U^d \leq m^2/(3m-2)$, at most $(m-1)$ tasks can have utilization $> m/(3m-2)$. Let the number of such tasks be k_o . Let $m_o = m - k_o$. Let $\hat{\tau}$ be the set of tasks,

each having $U_j^d \leq m/(3m - 2)$, and $\tau^{(k_o)}$ be the rest of the tasks. It can be proved by using the above inequalities that for each $\tau_i \in \hat{\tau}$, we have

$$U_i^d \leq \frac{m_o}{3m_o - 2} \quad (25)$$

and

$$U^d(\hat{\tau}) = \sum_{\tau_j \in \hat{\tau}} U_j^d \leq \frac{m_o^2}{3m_o - 2} \quad (26)$$

Therefore, since $\hat{\tau}$ is a “light set” of tasks, it is schedulable by algorithm DM on m_o processors.

Now, there are k_o processors left, and k_o tasks $\in \tau^{(k_o)}$ remaining. Consider the task system $\tilde{\tau}$ obtained from τ by replacing each task in $\tau^{(k_o)}$ by a task that has the same deadline, but whose computation time is equal to the deadline. Clearly, algorithm DM-US[$m/(3m - 2)$] will devote k_o processors exclusively to the tasks having $U_j^d = 1$, and will schedule the remaining tasks on m_o processors by algorithm DM. Thus, $\tilde{\tau}$ is schedulable. Now, consider the task set τ : it can be considered as a task set obtained from $\tilde{\tau}$, where the tasks with $U_j^d = 1$ do not execute to their full capacity, but execute only to their original capacities. Clearly, by Ha-Liu’s theorem 4.2, τ is schedulable. \square

As an illustration of the use of the test, consider the following periodic task set containing 8 tasks of type (2, 8), and 1 task of type (8, 9), and where $m = 8$. The utilization of this task set is $26/9 < 8^2/(3.8 - 2)$. Therefore, this task set is schedulable on 8 processors. Since $8/9 > 8/(3.8 - 2)$, the task (8, 9) is allocated to a single processor, and the rest of the tasks are scheduled on the remaining 7 processors. Now, note that this task set is not schedulable using the RM algorithm. This is because the algorithm would first execute the 8 tasks of type (2, 8) on each of the 8 processors for 2 time units. This would render (8, 9) not schedulable. In fact, for certain task sets containing tasks with large utilization such as the above, RM would not be able to schedule them, but RM-US would be able to. The authors[2] call this effect, the *Dhall* effect, as a similar example was first given by Dhall and Liu[22].

4.3.3 $m = 2$ case

The RM-US[$m/(3m-2)$] algorithm[2] can schedule any task set whose utilization is ≤ 1 . Andersson et al.[2] also mention that it was shown that some partitioning algorithms can schedule any task set with utilization $\leq (\sqrt{2} - 1)(m + 1)$ which is ≤ 1.242 for $m = 2$.

It was shown by Andersson et al.[2] that no static priority multiprocessor algorithm (used to schedule tasks for *any* m), whether partitioned or global, can offer a bound greater than $m/2$. We can adapt their proof slightly to prove that for $m = 2$ case: *no static priority algorithm, whether partitioned or global, can offer a bound greater than 1.5*. Consider the task set $\{(L, 2L - 1), (L, 2L - 1), (L, 2L - 1)\}$ where $L \geq 2$. The utilization of this task set is $3L/2L - 1 > 1$. No partitioned algorithm can schedule this because any two tasks have utilization greater than 1. No global algorithm can either because at least one task would take time $2L$ to finish. Thus this task set is not schedulable. As $L \rightarrow \infty$, the utilization $\rightarrow 1.5$. This means no algorithm can offer a bound b greater than 1.5 because if it does, we can choose L large enough so that its utilization is less than b .

4.4 EDF-US[$m/(2m - 1)$]

We saw in the previous section that the proof technique for RM-US[$m/(3m-2)$] [2] is quite adaptable to the DM case. Indeed, it turns out that it is also adaptable to the EDF case: we present a dynamic priority algorithm EDF-US[$m/(2m - 1)$] that schedules any task set whose utilization $U \leq m^2/(2m - 1)$.³

³This result was published by Srinivasan and Baruah[24]. We had a glance at the abstract, which stated the schedulability result, and at the paper, whose organization and proof strategies were similar to those for the RM-US[$m/(3m - 2)$][2]. That motivated us to derive the result presented in this subsection. The reader may refer to the paper by Srinivasan and Baruah [24] for a complete proof.

We only define “light sets” here, and show the proof of a lemma similar to lemma 4.2. Rest of the lemmas and their proofs are similar to their counterparts in the RM-US $[m/(3m - 2)]$ case, and do not use EDF-scheduling as a key point in the proof.

Definition 4.2 Light set : A task set τ is said to be a “light set” if it satisfies the following properties:

Property 3 : $U_i \leq \frac{m}{2m-1}$

Property 4 : $U \leq \frac{m^2}{2m-1}$

Let τ be a “light set” of n tasks. We will prove that τ is schedulable by algorithm EDF.

It can be proved, in lines similar to the proof of lemma 4.1 that

$$W(EDF, m, 1, \tau^{(k)}, t) \geq W(PSS, m, \frac{m}{2m-1}, \tau^{(k)}, t) \quad (27)$$

We now prove the key lemma which states that :

Lemma 4.3 τ is schedulable by EDF on m unit speed processors.

Proof :

We will prove that τ_k is schedulable for any $1 \leq k \leq n$ - it will then follow that τ is schedulable.

Let us assume that the first $(l - 1)$ instances of τ_k have met their deadlines under the algorithm EDF; we will prove that the l^{th} instance of τ_k also meets its deadline. The lemma will then follow by induction starting from $l = 1$.

The l^{th} instance of τ_k arrives at time-instant $(l - 1)T_k$, has a deadline at time-instant lT_k , and needs C_k units of execution. From (27), and the fact that PSS schedules each task τ_j for $(l - 1)T_k \cdot U_j$ units over the interval $[0, (l - 1)T_k)$, we have

$$W(EDF, m, 1, \tau, (l - 1)T_k) \geq (l - 1)T_k \left(\sum_{j=1}^n U_j \right). \quad (28)$$

Also at least $(l - 1)T_k \cdot \left(\sum_{j=1, j \neq k}^n U_j \right)$ units of this execution by algorithm EDF was of tasks in $\tau' = \tau / \{\tau_k\}$ because *exactly* $(l - 1)T_k U_k$ units of work was generated by τ_k prior to $(l - 1)T_k$ - so the remainder of the work must have been generated by τ' .

Now, the cumulative execution requirement of all instances generated by τ' that arrive prior to lT_k , and whose deadline is $\leq lT_k$, is bounded from above by

$$\begin{aligned} & \sum_{j=1, j \neq k}^n \left\lfloor \frac{lT_k}{T_j} \right\rfloor C_j \\ & \leq \sum_{j=1, j \neq k}^n \left(\frac{lT_k}{T_j} \right) C_j \\ & = lT_k \sum_{j=1, j \neq k}^n U_j \end{aligned} \quad (29)$$

Since at least $(l - 1)T_k \sum_{j=1, j \neq k}^n U_j$ amount of work gets done prior to $(l - 1)T_k$, the amount of work to be done in $[(l - 1)T_k, lT_k)$ on tasks in τ' is at most

$$\left(T_k \sum_{j=1, j \neq k}^n U_j \right) \quad (30)$$

$$(31)$$

Therefore, the total processor capacity left unused by τ' during the interval $[(l-1)T_k, lT_k)$ is at least

$$m \cdot T_k - \left(T_k \sum_{j=1, j \neq k}^n U_j \right) \quad (32)$$

Since there are m processors available, the cumulative length of the intervals over $[(l-1)T_k, lT_k)$ during which τ' leave at least one processor idle, is minimized if the different processors tend to idle simultaneously (in parallel); hence, a lower bound on this cumulative length of the intervals over $[(l-1)T_k, lT_k)$ during which τ' leave at least one processor idle is given by $\left(m \cdot T_k - \left(T_k \sum_{j=1, j \neq k}^n U_j \right) \right) / m$, which equals

$$T_k - \frac{1}{m} \left(T_k \sum_{j=1, j \neq k}^n U_j \right) \quad (33)$$

For the l^{th} instance of τ_k to meet its deadline, it suffices that this cumulative interval length be at least as large as τ_k 's execution requirement; that is,

$$\begin{aligned} T_k - \frac{1}{m} \left(T_k \sum_{j=1, j \neq k}^n U_j \right) &\geq C_k \\ \Leftrightarrow \frac{C_k}{T_k} + \frac{1}{m} \left(\sum_{j=1, j \neq k}^n U_j \right) &\leq 1, \\ \Leftrightarrow U_k \left(1 - \frac{1}{m} \right) + \frac{1}{m} \left(\sum_{j=1}^n U_j \right) &\leq 1, \end{aligned}$$

Now,

$$\begin{aligned} &U_k \left(1 - \frac{1}{m} \right) + \frac{1}{m} \left(\sum_{j=1}^n U_j \right) \\ &\leq \text{(By property 4)} \\ &U_k \left(1 - \frac{1}{m} \right) + \frac{m}{2m-1} \\ &\leq \text{(By property 3)} \\ &\frac{m-1}{2m-1} + \frac{m}{2m-1} \\ &= 1. \end{aligned}$$

□.

The rest of the lemmas and proofs are similar to the RM-US $[m/(3m-2)]$ case.

4.5 Performance Evaluation : Some Notes

Andersson et al. [2] discuss the performance of RM-US $[m/(3m-2)]$ algorithm, for $m = 32$ processors. They found that even though the utilization bound provided by the algorithm is low (34%) compared to previous multiprocessor scheduling algorithms (one algorithm provided 41%), it nevertheless performs better than other algorithms in practice. Some task sets having utilization close to 80% were schedulable by RM-US $[m/(3m-2)]$, whereas the multiprocessor RM failed to schedule task sets having much smaller utilizations due to the Dhall effect.

5 New Results on Fixed-Priority Aperiodic Servers

In this section, we study some new results derived for fixed-priority aperiodic servers by Bernat and Burns[4]. The main contention of the authors is that the DS server is as efficient as the SS server, if not better, though the traditional view has been otherwise because the utilization-based sufficient test for schedulability for SS server offers a higher bound than the bound offered by the test for DS server. The authors show using response-time-based analysis and experimental results that choosing the server parameters using a new method gives the same utilization for both DS and SS. Further, they show that both DS and SS can achieve the same aperiodic responsiveness.

5.1 Review of Erstwhile Methods used in Aperiodic Servers

Let τ be a given set of periodic tasks. The response time based analysis for DS and SS was presented by Strosnider et al.[13], and it was based on computing the maximum interference that the server can produce on the periodic tasks. For all periodic tasks, J_i is assumed to be zero. In the following section, we will use superscripts of DS and SS for the parameters of deferrable and sporadic servers respectively.

5.1.1 Response time based tests

For a DS server, because of the *double hit* phenomenon, the critical instant for an instance of a periodic task is at the beginning of a double hit period. Therefore, the worst-case interference that the DS server τ_s^{DS} can introduce in any periodic task in the interval $[0, w]$ is

$$\left(1 + \left\lceil \frac{w - C_s^{DS}}{T_s^{DS}} \right\rceil\right) C_s^{DS}.$$

Therefore, the worst-case response time of a periodic task τ_i is given by R_i , the fixed-point solution to the following recursive equation, starting with $R_i^0 = C_s^{DS}$:

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j + \left(1 + \left\lceil \frac{R_i^k - C_s^{DS}}{T_s^{DS}} \right\rceil\right) C_s^{DS} \quad (34)$$

A periodic task is schedulable if $R_i \leq D_i$.

Now, for a sporadic server, since τ_s^{SS} can be treated as a periodic task, the interference it produces is simply

$$\left\lceil \frac{w}{T_s^{SS}} \right\rceil C_s^{SS}$$

and therefore, the response-time equation is

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j + \left\lceil \frac{R_i^k}{T_s^{SS}} \right\rceil C_s^{SS} \quad (35)$$

where $R_i^0 = 0$.

Equations 34 and 35 have solutions if $U_p + U_s^{DS} \leq 1$, and $U_p + U_s^{SS} \leq 1$. Note that the time complexity of using this approach for schedulability analysis is pseudo-polynomial in the input size.

5.1.2 Server parameter selection

Here, we discuss the criteria used by erstwhile methods to decide the server task parameters.

Two key factors determine selection of parameters for the server task :

1. The schedulability of the periodic task set : this should not be compromised.

2. The response time of aperiodic requests : this should be minimized as much as possible.

The properties desirable of the server task parameters for both SS and DS are :

1. C_s/T_s must be as large as possible.
2. C_s must be as large as possible. This reduces the probability of the aperiodic task exhausting the server capacity. *This is useful if there are critical sections present during execution*, because then the probability of the aperiodic task remaining in the critical section after server capacity is exhausted and thereby introducing additional interference to lower priority tasks, is probably reduced.
3. τ_s must have as high priority as possible.

The above three properties together try to ensure good aperiodic responsiveness. Toward these ends, Bernat and Burns mention[4] that the traditional methods followed the following heuristics :

1. Select T_s as large as possible, usually the same as T_1 , so that the tasks are still in rate-monotonic order.
2. Then select C_s as large as possible, either using the sufficient tests, or if possible, using the above mentioned RTA test.
3. Strosnider et al.[13] discuss that by choosing T_s to be less than T_1 can result in higher utilization, when judged based on the RTA test, because of favorable phasing with the rest of the tasks.

Performance analysis : Performance analysis to compare relative performance of DS and SS were performed by Strosnider et al.[13] and Sprunt et al.[23] using the above criteria for choosing parameters. Bernat and Burns mention that the simulations, however, were not extensive, and did not consider aperiodic task sets whose computation times exceeded the capacity of the server. The conclusions that were arrived from these experiments, and previous known properties about SS were

- SS can be treated as a periodic task, and DS not; therefore, analysis of SS is easier.
- SS has a higher utilization than DS and allows larger capacities.

Bernat and Burns propose that these are not quite correct and in the following subsections, we shall see some of their conclusions.

5.2 New Results Concerning DS and SS[4]

Bernat and Burns[4] actually include two additional parameters for characterizing a task τ_i : the blocking factor B_i due to lower priority tasks, and the worst-case time $C_{i,b}$ that the task can spend in critical section b . They then discuss the *double-blocking problem*⁴ which occurs when an aperiodic task remains in a critical section after the server capacity is exhausted. However, in the experiments, they do not mention about the blocking factor, nor do they explicitly discuss about selecting parameters so that the blocking effect is minimized. The results seem valid even if this is not considered and therefore, we omit discussing this in this paper.

5.2.1 DS as a periodic task

For purposes of showing DS as a periodic task, we change our periodic model slightly. We make a distinction between *release* of any instance of τ_i and the time by which it becomes *ready*, that is, it is *recognized* by the scheduler, and put into the scheduler queue for execution. The maximum delay that any instance of τ_i can experience after its arrival and before it becomes ready is defined as the *release jitter* RJ_j .

⁴To be distinguished from double-hit phenomenon.

Audsley et al. [3] showed that the response time analysis equation (6) changes to the following if release jitter is taken into account :

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k + RJ_j}{T_j} \right\rceil C_j \quad (36)$$

We now note that the worst-case response time for a periodic task occurs when it is released at the *double-hit* instant, and when the server task executes to its fully capacity at the beginning of each of its periods. Therefore, for purposes of schedulability analysis, we can treat the DS as a periodic task having a release jitter $RJ_s^{DS} = T_s - C_s$, and the worst case scenario for periodic tasks is when the first instance of τ_s executes at time $T_s - C_s$ and every other instance executes at the beginning of its period. All the periodic tasks have a release jitter of $RJ = 0$.

If τ_s is considered as the 0^{th} periodic task τ_0 , the equation for calculating the response time of any task $\tau_i, i \geq 0$ is

$$R_i^{k+1} = C_i + \sum_{j=0}^{i-1} \left\lceil \frac{R_i^k + RJ_j}{T_j} \right\rceil C_j \quad (37)$$

Thus, the DS too can be considered as a periodic task, and its exact schedulability analysis is same as that for periodic tasks.

5.2.2 Maximum utilization of DS and SS servers

The authors show that appropriate selection of server parameters can make DS perform similar to that of SS.

The authors reverse the procedure of selecting the parameters : they first select C_s and then decide T_s . They show that there is a range from which C_s can be chosen : $[1, C^*]$, where C^* is the value corresponding to the amount of time the highest priority task could run without making any lower priority task miss its deadline, i.e., the *minimum* of the total amount of idle time at each priority level. For DS, actually, the maximum capacity becomes $C^*/2$ because of double hit, whereas for SS it is C^* . Now, for each possible C_s in this range, we can determine a corresponding T_s such that the task set is schedulable (decided using exact schedulability analysis) and the utilization is maximized. For the range of T_s , the fact that the total utilization must be less than 1 gives a lower bound. For the upper bound, we can choose a sufficiently large value so that the task set becomes schedulable (for eg. 10 times the largest period).

Example of server parameter selection : We illustrate the server parameter selection using the old methods and using the method used by the authors. Consider the task set $\{(2, 8), (1, 10), (1, 20)\}$ used in section 2 to illustrate operation of DS and SS. The utilization of this task set is 0.4. The corresponding bounds for DS and SS utilizations using the sufficiency tests are 0.256 and 0.34. Now, the old methods fix $T_s^{DS} = T_s^{SS} = 8$. The sufficiency tests fix the upper bounds on the capacities to be $C_s^{DS*} = 2$ and $C_s^{SS*} = 2$ yielding a utilization of 0.25 for both the servers.

When the response time analysis test is used, the parameters remain as (2, 8) with a utilization of 0.25 for DS, but change to (4, 8) for SS with a utilization of 0.5. So using the exact tests improves SS utilization.

We now use the method of the authors. The maximum value for which a task can execute from $t = 0$ to $t = C^*$ at the highest priority without making any lower priority task (any of the three periodic tasks) miss its deadline is 5. Thus, the C^* values for DS and SS are $5/2 = 2$ and 5 respectively, that is C_s^{DS} can lie in the range $[1, 2]$ and C_s^{SS} can lie in the range $[1, 5]$. Both (1, 2) and (2, 4) can be feasible parameters for the DS server, both at a utilization of 0.5. For SS, the possible parameters at the highest possible utilization of 0.5 are (1, 2), (2, 4), (3, 6), (4, 8), (5, 10). These were derived by manually simulating the task sets from $t = 0$ to $t = 40$, when the periodic tasks are released at their critical instants.

The results are summarized in table 3. The results for this example indicate the following :

Server	Possible Server Parameters for Maximum Utilization		
	Sufficient tests	Old RTA method	New method
DS	(2, 8)	(2, 8)	(1, 2), (2, 4)
SS	(2, 8)	(4, 8)	(1, 2), (2, 4), (3, 6), (4, 8), (5, 10)

Table 3: The server parameters for DS and SS using various methods for the periodic task set $\{(2, 8), (1, 10), (1, 20)\}$.

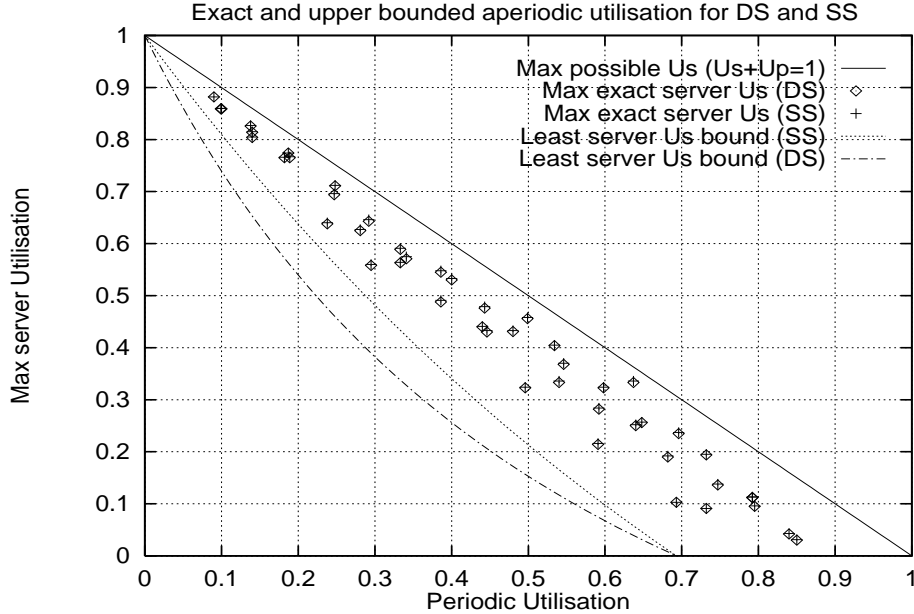


Figure 5: Maximum possible utilizations for DS and SS servers

- Using the new method, much higher utilization can be achieved for the DS server. The maximum utilizations are the same for both DS and SS.
- The maximum utilizations are achieved at periods different from T_1 .
- The maximum utilizations are achieved at more than one value of the capacity of the server.

Experimental evaluation by the authors : The figure 5 is taken from the paper[4]. The maximum utilizations possible for DS and SS using the above technique was found for a fairly large set of periodic tasks (about 50). The diamonds and crosses are the points of maximum utilizations of the DS and SS servers for each of the task sets. As it can be seen, the values are always above the bounds determined by the sufficient utilization based tests - this shows that the new analysis test actually does quite well in practice compared to utilization based tests. Further, it can be observed that the maximum values for DS and SS servers were almost the same, although the authors note that they were attained at different server capacities. Server capacity does play an important role in aperiodic responsiveness, and therefore needs to be taken into account. *However, the figure shows that based on the maximum-achievable-utilization alone, distinguishing SS and DS is not correct.*

5.2.3 Server Capacity Selection

We study here the effect of the server capacity on the maximum possible utilization and the aperiodic responsiveness (measured as the average response time of the aperiodic tasks).

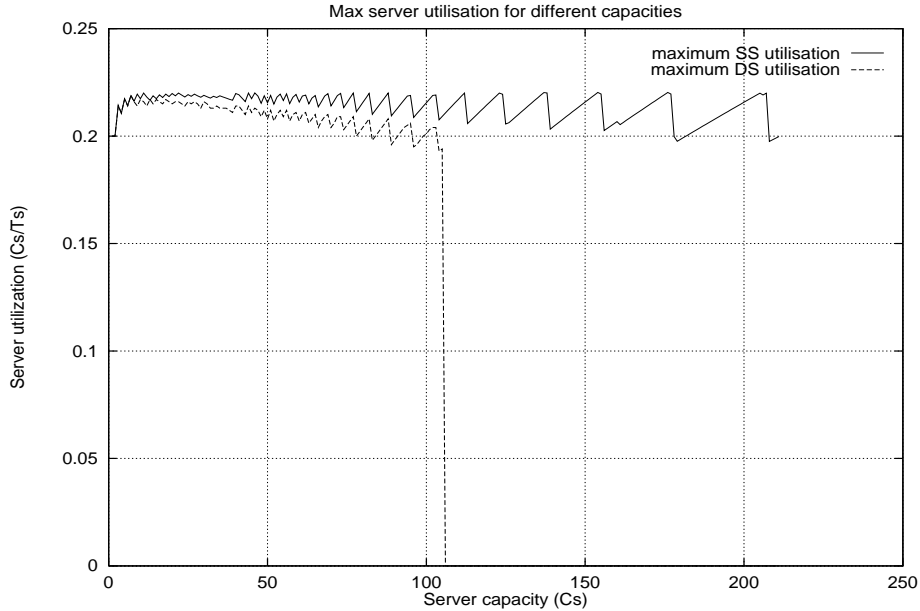


Figure 6: Maximum server utilizations possible at various capacities

The authors derived the maximum utilization possible for various server capacities, for a given task set; the graph in figure 6 shows their results. We can see that the utilization drops to zero for DS after about 110, while for SS it continues to about 220, as expected since the C^* value for DS is half that for SS. Further,

- The maximum possible utilization is slightly lower for DS than for SS.
- While, in general, the maximum possible utilization decreases for DS as the capacity increases, for SS, it can be attained for a number of capacities.

The authors also measured the aperiodic responsiveness at various possible server capacities for many aperiodic task requests. The graph in figure 7 shows their results. We can observe that

- The response time of DS is always smaller than that of SS for a given capacity.
- Both, however, do attain the same responsiveness at different capacities.

Based on the above results, we can infer that *it is best to choose largest capacity possible provided we can attain the maximum utilization at that capacity*. Since the server utilization is a non-monotonic function of capacity, we can instead choose the largest possible capacity at a local maximum of the server utilization.

For the **example** in table 3, the best parameters we can choose for DS are (2, 4) and that for SS are (5, 10).

5.3 Comparison between DS and SS

We compare DS and SS based on the light of the above results :

1. Both DS and SS *can be treated as a periodic task* in an appropriate task model.
2. *Maximum achievable utilization* is the same for both tasks though achieved at different capacities.

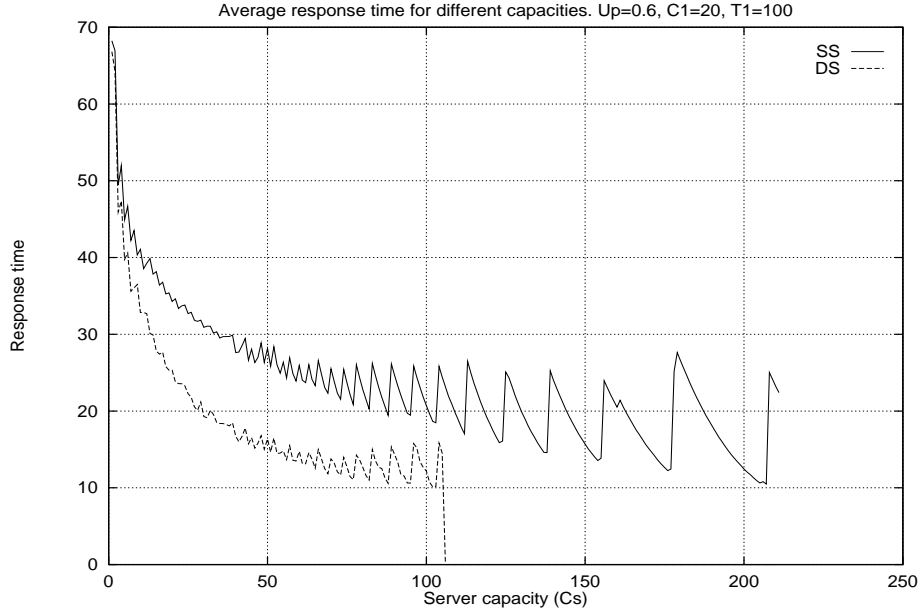


Figure 7: Average aperiodic response time for a particular periodic task set

3. *Server Parameters* : Though SS can have double the capacity of DS, the aperiodic response is the same. DS covers up the lack due to double-hit. However, SS can function at large capacities, whereas DS cannot.
4. *Parameter selection procedure* is very costly, since it involves determining the periods for each possible C_s in $[1, C^*]$.
5. *Ease of implementation* : DS is easier to implement because replenishment times are known in advance. SS is comparatively difficult to implement since we need to store the replenishment times and amounts. These could be $O(T_s)$ in number in worst case. Because of this overhead, since DS is comparable in performance with SS, it may be wiser to use DS.

6 Combining the Results/Concepts in the Papers

In this section, we attempt to combine the results presented in the three papers. We first derive the HET test for deadline-monotonic case and then for the deferrable server. We then propose models for deferrable and sporadic servers on multiprocessors.

6.1 HET tests for Other Static/Fixed Priority Algorithms

In this section, we will adapt the definitions and conventions used in section 3.

6.1.1 HET test for DM algorithm

Let $W_i(b)$ and $\psi_i(b)$ defined as in section 3. Clearly, (10) and (11) hold in the DM case also. That is

$$\forall t \in [0, b] : W_i(b) \leq \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t) \quad (38)$$

and

$$W_i(b) = \sum_{j=1}^i \left\lceil \frac{\psi_i(b)}{T_j} \right\rceil C_j + (b - \psi_i(b)). \quad (39)$$

Now, Γ_i is schedulable if and only if $C_i + W_{i-1}(D_i) \leq D_i$ for all i . It can be seen that lemma (6.1) holds in this case also, with a very slight modification of the proof (in the induction step, in case (1), the last instance of τ_{i+1} misses its deadline at $(\lfloor b/T_{i+1} \rfloor - 1)T_{i+1} + D_{i+1}$ which is $\leq \lfloor b/T_{i+1} \rfloor T_{i+1}$). We now derive the final statement of the theorem. Now the task set is schedulable iff

$$\begin{aligned} & \forall i : C_i + \sum_{j=1}^{i-1} \left\lceil \frac{\psi_i(D_i)}{T_j} \right\rceil C_j + (D_i - \psi_i(D_i)) \leq D_i \\ \Leftrightarrow & \forall i : C_i + \sum_{j=1}^{i-1} \left\lceil \frac{\psi_i(D_i)}{T_j} \right\rceil C_j \leq \psi_i(D_i) \\ \Leftrightarrow & \forall i : \sum_{j=1}^i \left\lceil \frac{\psi_i(D_i)}{T_j} \right\rceil C_j \leq \psi_i(D_i) \\ \Leftrightarrow & \forall i : \frac{\sum_{j=1}^i \left\lceil \frac{\psi_i(D_i)}{T_j} \right\rceil C_j}{\psi_i(D_i)} \leq 1 \quad (\psi_i(b) > 0) \\ \Leftrightarrow & \forall i : L_i = \left(\min_{t \in \mathcal{P}_i(D_i)} \frac{\sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \right) \leq 1 \end{aligned}$$

6.1.2 HET test for Deferrable Server Algorithm

Here, we modify the definition of i -active. The processor is i -**active** at t if any instance of either τ_s or Γ_i is active at t . The definitions of $W_i(b)$ and $\psi_i(b)$ use the new definition of i -active. We also say that the processor is s -active at t if an instance of τ_s is active at t . We define $W_s(b)$ to be the total time for which the processor is s -active in $[0, b]$, and $\psi_s(b)$ is the last instant in $[0, b]$ for which the processor is not s -active.

We assume that the periodic tasks and DS are invoked as in the worst case scenario. In this case, the equations (10) and (11) have to be slightly modified. We now have (for $b > 0$)

$$W_i(b) \leq \begin{cases} b & \text{for } t = 0 \\ \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{t-C_s}{T_s} \right\rceil C_s + (b-t) & \text{for } t \in (0, b] \end{cases} \quad (40)$$

The above equation holds because for $t = 0$, $W_i(b)$ can at most be b . Now, for $t > 0$, the same reasoning as for (10) holds except that we now have to include the server task. Here we assume that the ceiling of any x for $-1 < x \leq 0$ is 0.

Now, the following equation holds since $\psi_i(b)$ is the last instant in $[0, b]$ at which the processor is not i -active, that is, all the instances released before $\psi_i(b)$ have been executed by $\psi_i(b)$.

$$W_i(b) = \begin{cases} b & \text{for } \psi_i(b) = 0 \\ \sum_{j=1}^i \left\lceil \frac{\psi_i(b)}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{\psi_i(b)-C_s}{T_s} \right\rceil C_s + (b - \psi_i(b)) & \text{for } \psi_i(b) > 0 \end{cases} \quad (41)$$

Now, $\Gamma_i \cup \{\tau_s\}$ is schedulable if and only if $\forall i : C_i + W_{i-1}(T_i) \leq T_i$. Since we have an expression for $W_i(b)$ in terms of $\psi_i(b)$, all that remains is to find $\psi_i(b)$. The following lemma helps us to find $\psi_i(b)$:

Lemma 6.1 *Let $\Gamma_i \cup \{\tau_s\}$ be schedulable by DS and let $\mathcal{P}_i(b)$ be defined as follows :*

$$\mathcal{P}_i(b) = \begin{cases} \mathcal{P}_s(b) = \begin{cases} 0 & \text{if } b < C_s \\ \left\{ b, \left\lfloor \frac{b-C_s}{T_s} \right\rfloor T_s \right\} & \text{if } b \geq C_s \end{cases} \\ \mathcal{P}_1(b) = \mathcal{P}_s \left(\left\lfloor \frac{b}{T_1} \right\rfloor T_1 \right) \cup \mathcal{P}_s(b) \\ \mathcal{P}_i(b) = \mathcal{P}_{i-1} \left(\left\lfloor \frac{b}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(b) \end{cases}$$

Then

$$\psi_i(b) \in \mathcal{P}_i(b).$$

Proof : We first prove that the definition of $\mathcal{P}_s(b)$ is correct. Suppose $b < C_s$. Then since τ_s executes from $t = 0$ to $t = C_s$, $t = 0$ is the last instant at which the processor is not s -active. Therefore, $\psi_s(b) = 0$ in this case. Suppose $b \geq C_s$. Then, since τ_s is invoked for an interval of C_s at the beginning of every period starting from $t = C_s$, there are two cases depending on whether the processor is s -active or not at b . If the processor is not, then $\psi_s(b) = b$. If it is, then $\psi_s(b)$ is the last instant at which an instance of τ_s was released, that is, $\psi_s(b) = \left\lfloor \frac{b-C_s}{T_s} \right\rfloor T_s$.

We now prove that the definition of $\mathcal{P}_i(b)$ is correct by induction on i .

Initial Step : We have to prove that $\psi_1(b) \in \mathcal{P}_1(b)$ for all b . Now, consider the interval $[\lfloor b/T_1 \rfloor T_1, b]$. In this interval, two things can happen :

1. *The processor is 1-active throughout the interval :* Then $\psi_1(b) = \psi_1(\lfloor b/T_1 \rfloor T_1)$ because the processor is 1-active in $[\lfloor b/T_1 \rfloor T_1, b]$. Now, for any x , $\psi_1(x) \leq \psi_s(x)$. Further, it must be the case that $\psi_1(\lfloor b/T_1 \rfloor T_1) = \psi_s(\lfloor b/T_1 \rfloor T_1)$. Suppose, on the contrary, it is a strict inequality. Then the last instance of τ_1 would miss its deadline at $\lfloor b/T_1 \rfloor T_1$.
2. *There exists an instant of time at which the processor is not 1-active :* Let $x \in [\lfloor b/T_1 \rfloor T_1, b]$ be an instant of time where no tasks in $\Gamma_1 \cup \{\tau_s\}$ are active. Since at time x the $(\lfloor b/T_1 \rfloor + 1)^{th}$ job of τ_1 has finished execution, τ_1 is never active in $[x, b]$, that is, the only task that can be active in that interval is τ_s . This implies that $\psi_1(b) = \psi_s(b)$.

Induction Step : If $\psi_i(b) \in \mathcal{P}_i(b)$ for all b , we have to prove that, given a schedulable set Γ_{i+1} , $\psi_{i+1}(b) \in \mathcal{P}_{i+1}(b)$ for all b .

Consider the time interval $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$. In this interval, two things can happen :

1. *The processor is $(i+1)$ -active throughout the interval :* Then $\psi_{i+1}(b) = \psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1})$ because the processor is $(i+1)$ -active in $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$. Now, for any x , $\psi_{i+1}(x) \leq \psi_i(x)$, because Γ_{i+1} has one additional task compared to Γ_i . Further, it must be the case that $\psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1}) = \psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$. Suppose, on the contrary, it is a strict inequality. Then the last instance of τ_{i+1} would miss its deadline at $\lfloor b/T_{i+1} \rfloor T_{i+1}$.
2. *There exists an instant of time at which the processor is not $(i+1)$ -active :* Let $x \in [\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$ be an instant of time where no tasks in Γ_{i+1} are active. Since at time x the $(\lfloor b/T_{i+1} \rfloor + 1)^{th}$ job of τ_{i+1} has finished execution, τ_{i+1} is never active in $[x, b]$. This implies that $\psi_{i+1}(b) = \psi_i(b)$.

Since by induction hypothesis, $\psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$ and $\psi_i(b) \in \mathcal{P}_i(b)$,

$$\psi_{i+1}(b) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \cup \mathcal{P}_i(b) = \mathcal{P}_{i+1}(b) \text{ by definition.}$$

□

We now derive the final form of the test. We first note that $\psi_i(T_i)$ cannot be 0 for any T_i since that would mean τ_i misses its deadline at T_i , contradicting the fact that τ_i is schedulable. Now, we need

$$\begin{aligned} & \forall i : C_i + W_{i-1}(T_i) \leq T_i \\ \Leftrightarrow & \forall i : C_i + \sum_{j=1}^{i-1} \left\lceil \frac{\psi_i(T_i)}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{\psi_i(T_i) - C_s}{T_s} \right\rceil C_s + (T_i - \psi_i(T_i)) \leq T_i \\ \Leftrightarrow & \forall i : C_i + \sum_{j=1}^{i-1} \left\lceil \frac{\psi_i(T_i)}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{\psi_i(T_i) - C_s}{T_s} \right\rceil C_s \leq \psi_i(T_i) \\ \Leftrightarrow & \forall i : \sum_{j=1}^i \left\lceil \frac{\psi_i(T_i)}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{\psi_i(T_i) - C_s}{T_s} \right\rceil C_s \leq \psi_i(T_i) \\ \Leftrightarrow & \forall i : L_i = \left(\min_{t \in \mathcal{P}_i(T_i)} \frac{\sum_{j=1}^i \left\lceil \frac{\psi_i(T_i)}{T_j} \right\rceil C_j + C_s + \left\lceil \frac{\psi_i(T_i) - C_s}{T_s} \right\rceil C_s}{t} \right) \leq 1. \end{aligned}$$

6.2 Fixed-Priority Servers on Multiprocessors

In this section, we propose models for fixed-priority servers on multiprocessors.

6.2.1 Partitioned task sets

In this case, we could have one deferrable/sporadic server on each of the m processors. Since the periodic tasks on each processor are scheduled using RM, the server parameters could be determined by the method used in section 5. The aperiodic tasks, which are queued in FIFO manner, can be scheduled globally on any of the available deferrable servers.

Several heuristics could be used to assign the aperiodic tasks to the processors. Since the deferrable server on each processor is of the highest priority, given the aperiodic execution time requirement, we can precisely determine the time at which the task would be completed because we know the current capacity of the server and the replenishment times along with the replenishment amounts, though this could be slightly complex for the sporadic server because of its replenishment scheme. Therefore, we can assign a task to that processor which can give the shortest response time. However, in a model where the aperiodic tasks have soft deadlines, and the criterion for good responsiveness is the fraction of the soft deadlines met, it could be better to assign a task to the processor which, in spite of having a large response time, can still meet its deadline. In other words, we can delay the execution of the aperiodic task as long as it meets its deadline.

6.2.2 Global task sets

In this case, we can have a single deferrable or a sporadic server at the highest priority which can execute on any of the m processors. The replenishment rules and amounts remain the same. The replenished capacity can be used to execute on any of the platforms. For scheduling the periodic tasks, we could use either RM or RM[$m/(3m - 2)$]. However, it is not clear how to determine the parameters for the server in this case.

7 Conclusion

In this paper, we studied three papers dealing with static- and fixed- priority preemptive real-time scheduling.

In the first paper by Bini and Buttazzo[6], we saw a new exact schedulability test for RM scheduling on uniprocessors. The test always performed better than the classical RTA test, and can potentially perform much better than RTA on task sets with large variations in periods of the tasks. The exact test could also be tuned based on a parameter to be faster although it then accepts fewer tasks as schedulable.

In the second paper by Andersson, Baruah and Jonsson[2], we saw a sufficient utilization based schedulability test for static-priority multiprocessor scheduling. We noted that the proof strategies for the test were quite general, that is, they did not greatly depend on the RM algorithm which the authors used as the basis, and therefore, we could adapt it to the DM case, and the EDF case. We also saw that even though the bound of the test was lower than known algorithms, the algorithm in practice was able to schedule tasks with much larger utilization. We also proved a negative result for schedulability bound on 2 processors.

In the third paper by Bernat and Burns [4], we discussed the performance of the DS and SS servers. We found that the traditional methods did not explore all possibilities for the parameters of the server tasks and preferred SS to DS because the schedulability bound of SS was larger. A new parameter selection strategy was proposed, and it was found from the simulations in the paper that both DS and SS achieve high utilizations, and have equal aperiodic responsiveness, though at different capacities.

References

- [1] Björn Andersson, Sanjoy Baruah, and Jonsson. Static-priority scheduling on multiprocessors. Technical report, Dept. of Computer Science, University of North Carolina at Chapel Hill.

- [2] Björn Andersson, Sanjoy Baruah, and Jonsson. Static-priority scheduling on multiprocessors. In *22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 193–202, December 2001.
- [3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
- [4] Guillem Bernat and Alan Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 68–78, December 1999.
- [5] Enrico Bini, G. C. Buttazzo, and G. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *13th IEEE Euromicro Conf. on Real-Time Systems*, June 2001.
- [6] Enrico Bini and Giorgio Buttazzo. The space of rate monotonic schedulability. In *23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 169–178, December 2002.
- [7] Giorgio Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*, chapter 5. Kluwer Academic Publishers, 2000.
- [8] Giorgio Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*, chapter 4. Kluwer Academic Publishers, 2000.
- [9] R. Devillers and J. Goossens. Liu and layland's schedulability test revisited. *Information Processing Letters*, 73:157–161, 2000.
- [10] J. Goossens and P. Richard. Overview of real-time scheduling problems. In *9th International Conference on Project Management and Scheduling*, April 2004.
- [11] Rhan Ha and Jane Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *14th IEEE International Conference on Distributed Computing Systems*, pages 162–171, June 1994.
- [12] K. Jeffay, D.F.Stanat, and C.U.Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium (RTSS'91)*, pages 129–139, December 1991.
- [13] J.K.Strosnider, J.P.Lehoczky, and L.Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.
- [14] J.K.Strosnider, L. Sha, and J.P.Lehoczky. Enhanced periodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium, RTSS'87*, pages 261–270, December 1987.
- [15] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.
- [16] J.P.Lehoczky, L. Sha, and Ye Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium (RTSS'89)*, December 1989.
- [17] Chung L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time-environment. *Journal of the Association for Computing Machinery (JACM)*, 20(1):46–61, January 1973.
- [18] Jane S. Liu. *Real Time Systems*, chapter 7, page 218. Prentice Hall, New Jersey, 2000.
- [19] Jane S. Liu. *Real Time Systems*, chapter 6, page 121. Prentice Hall, New Jersey, 2000.
- [20] Cynthia Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *29th Annual ACM Symposium on Theory of Computing*, pages 140–149, May 1997.

- [21] Mikael Sjödin and Hans Hansson. Improved response-time analysis calculations. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 399 – 408, December 1998.
- [22] S.K.Dhall and C.L.Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.
- [23] B. Sprunt, L. Sha, and J.P.Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.
- [24] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multi-processors. *Information Processing Letters*, 84:93–98, 2002.