Improving Dual Priority Scheduling Algorithm By Implementing Various Low Priority Orders

by

Khushboo Baghadiya

(Under the Direction of Shelby Funk)

Abstract

Dual Priority scheduling is a variation of the Fixed Priority scheduling paradigm in real-time systems. It was introduced to make non-real-time jobs complete sooner while ensuring that real-time jobs still meet their deadlines. In Fixed Priority scheduling, each task is assigned with a fixed priority and the task executes in that priority throughout the schedule. In Dual Priority algorithm every task has two priorities, high and low – each task runs in low priority as soon as it arrives and continues executing in its low priority order until it reaches its priority promotion time, and executes at its high priority level until completion. In original Dual Priority scheduling algorithm the low and high priority orderings are the same. This thesis focuses on implementing various low priority orderings to examine which low priority ordering should be chosen to make more tasks schedulable. We found that in many cases, having the low priority order be the reverse of the high priority order allows more task sets to be schedulable. Task sets whose low priority is ordered according to decreasing laxity (the time a job can remain in wait queue without missing a deadline) also seemed to be schedulable more often than other options.

Index words: Real-time, Dual priority, Scheduling, Multiprocessors, Fixed priority

Improving Dual Priority Scheduling Algorithm By Implementing Various Low Priority Orders

by

Khushboo Baghadiya

B.Tech., Jawaharlal Nehru Technological University, 2012

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master's of Science

Athens, Georgia

2015

Improving Dual Priority Scheduling Algorithm By Implementing Various Low Priority Orders

by

Khushboo Baghadiya

Improving Dual Priority Scheduling Algorithm By Implementing Various Low Priority Orders

Khushboo Baghadiya

July 20, 2015

Acknowledgments

I would sincerely like to thank Dr. Shelby Funk for her endless support and cooperation. She was an incredible help in my research and defense. I am very grateful to her for her guidance throughout my Master's program. I would also like to thank Dr. Liming Cai and Dr. Shannon Quinn for being on my committee and helping me improve my work. I would also like to appreciate the work by Dr. Chiahsun Ho and Mayur Jhadav which gave me a strog base for my research and I thank them for directing me in my research.

My family and friend shave also been a great support throughout and hugely appreciate all their effort in helping me to achieve my goal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A real-time system is any computing system which has to perform all its computations within specific time constrains. Each task in a real-time operating system (RTOS) has a restricted time period to complete its execution [1]. Accuracy of the system depends on both the logical result and the time it takes to complete the tasks. If the system fails to respond in the time period assigned for that particular task, it is considered to be an incorrect response.

Many real-time systems have jobs that repeat. We call these repeating jobs periodic jobs. Real-time systems (RTS) are used in various industries like health, defense, automobiles, etc. [2]. Each task may have one or more jobs, period, execution time and a deadline associated with it. The period is defined to be the amount of time between successive job releases. A deadline is a time period assigned to a task within which it is supposed to complete its execution. Apart from the period, execution time and deadline every task has its own priority which is determined by the scheduling algorithm.

Missing a deadline may have consequences based on what type of real-time system it is. When the system cannot afford to miss a deadline or missing a deadline has life threatening consequences then it is called a hard real-time system (HRT). An example of a HRT can be an aircraft control system in which a computer controls the flight of that aircraft. Every single action is timed in this case and if the system is not prompt enough to do the same then there might be a huge irreversible loss. Few more examples of hard real time systems could be an automated car, missile system, etc. [3].

When some deadline misses are allowed, it is called a soft real-time system (SRT). However the system tries to meet as many deadlines as possible. Online video streaming would serve as the best example for SRT. If the system misses some deadlines and a few frames are lost, the consequences are not catastrophic. However, it degrades the performance of the system indeed. Other examples of soft real-time system includes online reservation system, sound system, laser printer, etc. [3].

Tasks which have all the jobs of that task arrive in fixed time intervals are called periodic tasks. On the other hand when the jobs are not released in a strictly periodic pattern it is called sporadic task. However for sporadic tasks there is a minimum time gap between releases of every job of that task [4]. When a task of higher priority can interrupt execution of a task with lower priority then the scheduling algorithm is called preemptive. On the contrary when every task completes its execution without any interruption the scheduling algorithm is called non-preemptive.

A real-time system may involve one or more number of processors. A system that has just one processor is called a uniprocessor system where as if the system has two or more processors it is called a multiprocessor system. More and more multiprocessor scheduling algorithms are coming into picture. The increase in popularity is because multiprocessor systems can run many tasks on the system concurrently [3]. Using multi-core processor also satisfies todays need to acquire maximum processing power. Multiprocessor scheduling is basically divided into two types: Global scheduling and partitioned scheduling [5]. In global scheduling there is just one single global queue of tasks and a global scheduler assigns jobs to the processors for scheduling. Migration of tasks over processors is allowed in these algorithms. In partitioned scheduling every task set is partitioned into various subsets of the tasks. Each task subset has its own local scheduler which assigns jobs from that subset to a processor. Real-time scheduling in multiprocessors can be either online scheduling or offline scheduling. In an offline scheduling algorithm, the system is completely aware of the details and the timings of the tasks in advance. This knowledge is used to predict the behavior of the schedule and hence the scheduling decisions are made even before the system starts running. In an online scheduling algorithm there is no preexisting information about the task specifics and so all the scheduling decisions are made at run-time.

If the priorities of the tasks remain the same throughout the schedule then it is supposed to be static where as if the priorities of the tasks keep changing according to the scheduling algorithm during the schedule then it is called as dynamic [1]. Two of the multiprocessor scheduling algorithm is Standard Dual Priority algorithm (SDP) [6] and Modified Dual Priority (MDP) algorithm. These algorithms take into account high and low priority bands while scheduling. Standard Fixed Priority (SFP) algorithm is a fixed priority scheduling algorithm which assigns every task a fixed priority which remains constant throughout the schedule. The task with the higher priority is always given more preference than the tasks with relatively lower priority. It makes sure that only the task with the highest priority runs at any point in the schedule. Therefore every task has only one priority associated with it [1].

The dual priority scheduling algorithm is an online scheduling algorithm where every task has its own low and high priority and a priority promotion time [6]. Each job initially executes in its low priority order until it reaches its priority promotion time. At that point the job acquires high priority and continues executing in its high priority. A particular order is chosen for high priority band and low priority band while simulating either Standard Dual Priority algorithm [6] or Modified Dual Priority algorithm [3]. This thesis focuses on simulating Standard Dual Priority scheduling algorithm [6] on multiple unique task sets considering different priority orderings for high priority and low priority band and concluding which proves to the best out of all the combinations in different scenarios. The performance is gauged depending on how many task sets are schedulable or unschedulable with that particular low priority ordering.

Chapter 2

Model and Definitions

This is a general reference model for real-time systems. The terminology below will be used throughout this thesis in order to maintain consistency. A job is nothing but a unit of work that is scheduled and executed by the system. Many jobs combine to form a task which performs a specific function. Let there be m processors in the system. Various tasks together form a task set which is denoted by $\tau = \{T_1, T_2, \ldots T_n\}$ where $T_1, \ldots, T_n$ are the tasks. A task T_i is represented as a tuple (p_i, e_i) where p_i is the time period in which every job is released and e_i is the execution time of the job. In addition to all the parameters above, every job of task T_i has an arrival time which is represented as $a_{i,k}$ where $a_{i,k} = (k-1) \cdot p_i$ which means that the k^{th} job of task T_i, denoted by $T_{i,k}$ is released at that arrival time $a_{i,k}$. The deadline of the job is expressed as $d_{i,k} = k \cdot p_i$, where $d_{i,k}$ is the deadline of the k^{th} job of task T_i. The job $T_{i,k}$ is expected to execute for $_i$ time units in the interval $[a_{i,k}, d_{i,k}]$. The average amount of the time a task consumes the processor for its execution or the processing time is termed as processor utilization and is indicated as u_i and is calculated as $u_i = e_i/p_i$ [7].

The property indicating if all the tasks in a real-time system can meet its deadlines is called schedulability and a blue print of the tasks under execution is called a schedule. A task set is said to be valid if all the tasks in that task set meet its deadline and a schedule is said to be feasible if atleast one valid schedule exists for that particular task set [7]. We assume that the processor is idle while a job is waiting to execute and a higher priority job will never be in the wait queue while a lower priority job is executing.

This thesis focuses on improving the Standard Dual Priority scheduling algorithm by considering different priority orderings of high and low priority bands. The maximum amount of time that can elapse between arrival and completion of the job is called its worst case response time (WCRT), R_i. Worst case response time can be calculated by adding execution time e_i and worst case interference I_i. Interference in a task model is defined as the amount of the time a job has to wait after getting interrupted by a higher priority job before it completes its execution. In Dual Priority scheduling [6] a task T_i runs at its low priority (Ti,low) until a point where it acquires its high priority ($\tau_{i,high}$) and that point is called the priority promotion time or simply promotion time. At a certain point in execution a task gets promoted to higher priority level and then it continues to be in that high priority state until it completes its execution. The promotion time of a task T_i is represented as λ_i. The original Dual priority algorithm has a middle priority range in addition to low and high priority ranges. All the real-time jobs have two fixed priorities each – i.e., low and high. The jobs which are not time critical are called non real-time jobs and they execute in FIFO, first in first out order and acquire the middle range priority [6]. The purpose of this algorithm was to improve the response time of non-real-time jobs while still ensuring real-time jobs meet their deadlines.

A job's laxity [1] is defined as the maximum time a job can wait ideally without being executed without having to miss its deadline. The relative deadline of a task

Table 2.1: Basic notation

Notation	Description
T_i	Task number i
e_i	Worst case execution time of task T_i
R_i	Worst case response time of task T_i
D_i	Relative Deadline of task T_i
λ_i	Promotion time of task T_i
$\tau_{i,low}$	Low priority assigned to task T_i
$\tau_{i,high}$	High priority assigned to task T_i
p_i	Period of task T_i
a_i	Arrival time of job of task T_i
n	Number of tasks
m	Number of processors
u_i	Utilization of task T_i

is defined as the time elapsed between the arrival and completion of each job in the task. The absolute deadline is the point in time in the schedule when a job has to finish its execution at or before that time. A task is said to have implicit deadlines when the deadline of the task is equal to its period. Whereas when the deadline of the task is less than or equal to its period it is called constrained deadline.

Chapter 3

Related Work

As discussed above, the tasks can be scheduled using global and partitioned scheduling algorithms [5]. The scheduling algorithms can also be divided into Fixed Priority and Dynamic Priority scheduling algorithms. In Fixed priority algorithms each task has just one constant priority throughout the schedule whereas in Dynamic Priority algorithms the priority of the task keeps changing during the schedule due to various factors. This thesis focusses on Fixed Priority algorithms, however, Earliest Deadline First (EDF) [1] algorithm is one of the very popular Dynamic Priority algorithms. According to EDF the task with the earliest or the shortest deadline gets the higher priority and the task with the farthest deadline gets the least priority. Some of the important Fixed Priority scheduling algorithms are discussed below.

3.1 Fixed Priority Scheduling

In a Fixed Priority scheduling algorithm [1] every task has one fixed priority assigned to it. That task holds the given priority throughout the schedule. This priority decides which task will execute at any point in the schedule. The fixed priority algorithm makes sure that only the highest priority task among the arrived tasks execute at any time during the schedule.

Example 1 *Consider an example of a schedule where $T_i = (e_i, p_i)$ is the task at index i with execution time e_i and period p_i. They have the arrival time a_i and a deadline $d_i = p_i$. Let $T_1 = (2, 7)$; $T_2 = (3, 8)$ and $T_3 = (4, 12)$ with arrival times $a_1 = 2$, $a_2 = 3$ and $a_3 = 0$. The tasks T_1, T_2 and T_3 have priorities 1, 2 and 3 respectively. Figure 3.1 illustrates the schedule of these tasks. The rectangles in the schedule represent the execution time of the task, up arrows indicate arrival times of the task and down arrows indicate the deadline of the task.*

At $t = 0$ T_3 is begins executing and continues executing until T_1 arrives at time $t = 2$. At this point, the scheduler will suspend T_3's execution and T_1 starts executing since T_1 has higher priority than T_3. This is called preemption where T_1 is preempting T_3. T_1 then executes until completion, causing T_3 to wait for the processor. Meanwhile, T_2 arrives at time $t = 3$ while T_1 is still running. Task T_2 must wait to execute since T_1's priority is greater than the priority of T_2. In this case, after T_1 completes its execution, when T_2 and T_3 both are waiting to acquire the processor, T_2 will start executing and T_3 continues waiting to execute because T_2 has higher priority than T_3. Finally, T_3 resumes its execution after T_2 completes its execution successfully.

Because these are periodic tasks, T_1 will release a new task at time $t = 9$ and T_2 and T_3 will release new jobs at times 11 and 12, respectively. The scheduling decisions will continue in the same manner as described above

3.2 RM (or DM)

The Rate Monotonic algorithm (RM) proposed by Liu and Layland [1] assigns higher priority to tasks with shorter periods. Therefore, the task with the least period will have the highest priority and the task with the longest period will have the

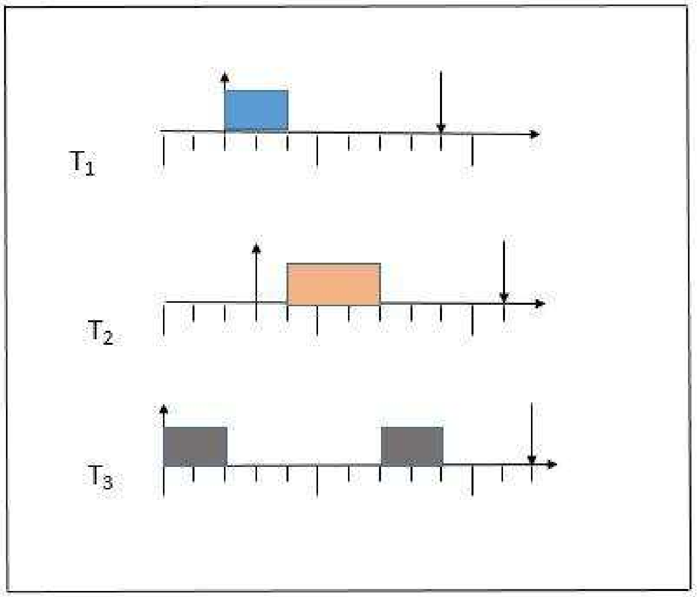


Figure 3.1: A sample Fixed Priority schedule.

lowest priority. Rate Monotonic scheduling algorithm is optimal for Fixed Priority on uniprocessor systems when deadlines are implicit, which means that if a task set can meet all deadlines by any Fixed Priority scheduling algorithm then it can meet all its deadlines by the Rate Monotonic algorithm [1].

In addition to the assumptions made above, they assumed a couple of more things which were not really required or are not applicable. This theory expected the tasks to have constant execution time but the analysis proves that Rate Monotonic algorithm works fine even when the execution times are just bounded above – i.e., e_i is the *worst-case* execution time, but actual execution time may be smaller than e_i. It will still succeed if the execution time is less than the period of the task. They also thought that the tasks have to be strictly periodic. However, when analyzed this theory works even when there is a task T_i which arrives *at most* once in every p_i units of time.

According to the model where deadlines are equal to periods, the scheduling algorithm is called Rate Monotonic algorithm but what if the deadlines are constrained i.e., if $D_i \leq p_i$ for all $i = 1, 2, \ldots, n$. When deadlines are not equal to periods but are less than or equal to periods, they are called constrained deadlines. As expected, the rate monotonic algorithm for choosing priority of the task based on its period will not turn out to be optimal when the deadlines are constrained. For example a task has a longer period but a shorter relative deadline would keep waiting for other tasks to execute since the priority chosen for that task was lower because of its large period. So some infrequent task might be more urgent than a frequent task. Therefore Deadline Monotonic policy (DM) [8] is more efficient than Rate Monotonic policy in this case. In the Deadline Monotonic algorithm a task with a shorter relative deadline should be given higher priority and the task with the farthest relative deadline will have the least priority. For the constrained systems, priority ordering according to the Deadline Monotonic technique proved to be optimal on uniprocessor systems [8].

Rate monotonic algorithm did turn out to be optimal but only for uniprocessor systems when the deadline is equal to the period. In case of a multiprocessor system there was another issue as to how would the tasks be assigned to the processors. Therefore rate monotonic algorithm is not optimal for multiprocessor systems.

3.3 TDA

Time Demand Analysis (TDA) is a technique used to perform fixed priority scheduling analysis. Liu and Layland [1] proved that if a task set with implicit deadline has a utilization $\leq n(2^{1/n} - 1)$ then it is RM schedulable on uniprocessor systems. Note that $\lim_{n\to\infty} n(2^{1/n} - 1) = \ln 2 \approx 69.3\%$. When utilization of the task set is greater than 69.3% then more accurate methods should be used to determine schedulability. Hence TDA is used to calculate the worst case response time of a task. TDA was initially introduced for uniprocessor [9] and was later extended for multi-processors [10, 11, 12]. A periodic task set's schedulability using a fixed priority scheduling algorithm is determined with the help of TDA.

On uniprocessors, the maximum requested processing time of task T_1 through T_i in a time interval of length t is calculated as [9]:

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \cdot e_k \tag{3.1}$$

The above equation is a consequence of the critical instant theorem [1]. Critical instant theorem can be stated as follows: Given a task T_i executing in a fixed priority schedule on a uniprocessor, worst case response time of T_i will occur when T_i release a job at time t and all higher priority tasks release jobs at the same time. In TDA the expression $sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \cdot e_k$ measure the high priority demand in an interval of length t assuming all high priority tasks are released at the beginning of the interval. By the Critical Instant Theorem, if $t*$ is the minimum value such that $w_i(t*) = t*$ then $t*$ is worst case response time of T_i.

The sum in Equation 3.1 defines the worst case high priority interference during an interval of length t in uniprocessor systems with constrained deadlines. The value $w_i(t)$ is defined as the worst-case system demand of a task $T_1, \ldots, T_i$ during a time interval of length t. If there exists a t^* such that $w_i(t_*) = t^* \leq D_i$ then naturally T_i will complete its execution without missing any of its deadlines. That is because the worst case system demand is less than the time interval so even if the T_i executes in its worst case scenario, it will still meet all its deadlines.

The longest interval in which the processor executes tasks at priority k or higher is known as level-k busy period. Various kinds of jobs in multi-processor systems executing in level-k busy period are called body, carry-in and carry-out. Jobs are

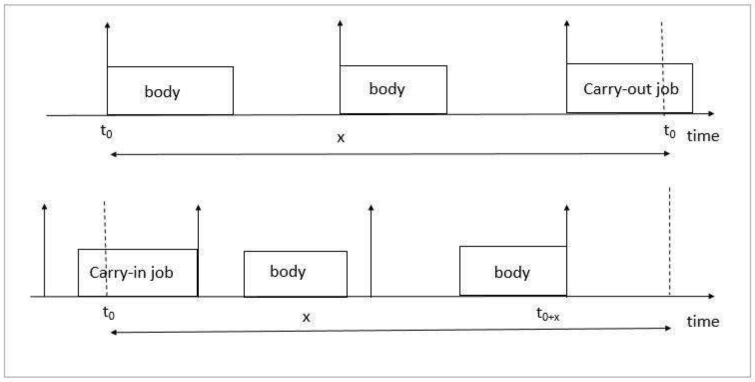


Figure 3.2: Carry-in and No-carry-in jobs.

classified in the above categories based on their arrival times and deadlines with respect to level-k busy period as described below and illustrated in Figure 3.2.

- Body: Jobs that arrive in level-k busy period and also have a deadline in level-k busy period.
- Carry-in: These jobs also have their deadlines in level-k busy period, however, they arrive before the level-k busy period.
- Carry-out: Jobs that arrive in the level-k busy period but have deadlines after the level-k busy period.

These are considered since the critical instant theorem [1], which applies for uniprocessors, does not apply for multi processors [10].

Research by Bertogna et al. [10, 11] and Guan et al. [12] proposed an extended model to compute fixed priority response times more accurately in case of multi processor systems. Their approach first computes the worst-case length of time a task T_i might prevent task T_k from executing – this is called T_i's interference on T_i. The analysis takes two types of interferences into consideration interference from carry-in ($I_k^{CI}(T_i, x)$) and no-carry-in ($I_k^{NC}(T_i, x)$) jobs, and proves that there cannot be more than $m-1$ carry-in jobs. Note I_k^{NC} includes both body and carry-out jobs.

Let $\tau_{<k} = \{T_1, \ldots, T_{k-1}\}$ be the tasks with priority higher than task T_k. Guan's method determines the worst-case partition of $\tau_{<k}$ into τ^{NC} and τ^{CI} that results T_k's worst-case response time. Specifically, let $\mathcal{Z}_{\|} \subseteq \tau \times \tau$ be the set of all partitions of $\tau_{<k} = T_1, \ldots, T_{k-1}$. Guan's method partitions $\tau_{<k}$ into τ^{NC} and τ^{CI} such that $\tau^{NC} \cup \tau^{CI} = \tau_{<k}$, $\tau^{NC} \cap \tau^{CI} = \emptyset$, and $|\tau^{CI} \leq m-1$.

Guan et al. also derived a formula to calculate an upper bound on the total interference in an interval of length x caused by all the high priority jobs:

$$\Omega_k(x) = \max_{(\tau^{NC}, \tau^{CI}) \in \mathcal{Z}} \left(\sum_{T_i \in \tau^{NC}} I_k^{NC}(T_i, x) + \sum_{T_i \in \tau^{CI}} I_k^{CI}(T_i, x) \right) \tag{3.2}$$

Once the worst-case interference has been calculated, the Response Time Analysis (RTA) can be found in a manner similar to the uniprocessor TDA method. The worst-case response time is no larger than the smallest value x such that

$$\left\lfloor \frac{\Omega_k(x)}{m} \right\rfloor + e_k \leq x. \tag{3.3}$$

The equation 3.3 can be best explained using Figure 3.3. In the worst case, task T_k only executes while no other tasks are executing, leaving $m-1$ idle processors. If this occurs, then the $\Omega_k(x)$ time units of interference must occur in parallel.

This determines the upper bounds of the task T_k's response time. Therefore, it is concluded that the task may not be schedulable if $x > D_k$. The test above is a sufficient only test since the results of the analysis are pessimistic. Note that a sufficient only test may make fixed priority-schedulable task sets fail this test.

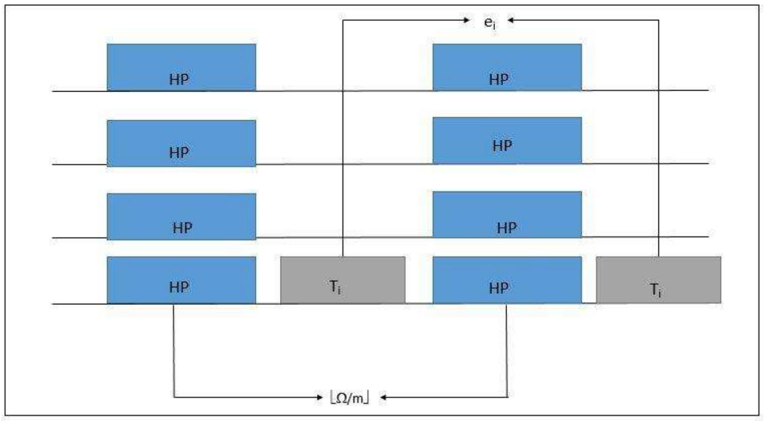


Figure 3.3: Guans's Response Time Analysis.

3.4 OPA

Initially Audsley devised an algorithm called Optimal Priority Assignment (OPA) algorithm which derives priority levels for a task set if the task set is FP schedulable [13][14]. When deadlines are longer than periods neither RM nor DM is and optimal FP scheduling algorithm. R. Davis and A. Burns then proved that OPA works in case of multiprocessors also under certain conditions with respect to schedulability. Optimal Priority Assignment relies on a schedulability test S: "For a given system model, a priority assignment policy P is referred to as optimal with respect to a schedulability test S, if there are no task sets, compliant with the system model that are deemed schedulable by test S using another priority assignment policy, that are not also deemed schedulable by test S using policy P" [15].

The pseudo code for OPA algorithm by Davis et al. [15] is given below:

Optimal Priority Assignment Algorithm

```
for each priority level $k$, lowest first
    {
        for each unassigned task $T_i$
        {
            if $T_i$ is schedulable at priority $k$ according to schedulability test $S$
            {
                assign $T_i$ to priority $k$
                break (continue outer loop)
            }
        }
        return unschedulable
    }
    return schedulable
```

The algorithm above does not account for the order in which tasks should be considered at each priority level higher than k. OPA algorithm will run the schedulability test S at most $n(n+1)/2$ times, where n is the number of tasks. This is a much better solution than actually inspecting all the $n!$ possible orderings. The above algorithm ensures a schedulable priority assignment according to schedulability test S, if one exists. They also proved that using Audley's OPA algorithm for any global FP schedulability test for periodic or sporadic task sets is said to be optimal if it simply abides by the following three conditions [15]:

1. Higher priority tasks may affect the schedulability of task T_k according to the test S but the relative priority of those higher priority tasks have no effect on it.

2. Similarly, lower priority tasks may affect the schedulability of task T_k according to the test S but the relative priority of those lower priority tasks have no effect on it.

3. According to the schedulability test S, if two tasks with the adjacent priority are swapped the task which is assigned the higher priority after swapping will never be unschedulable if it was schedulable before swapping. (As a corollary, a task with the lower priority after assignment can never become schedulable after swapping if it was unschedulable before swapping).

Davis and Burns concluded that Audsley's Optimal Priority Assignment algorithm can be taken into account with sufficient schedulability tests for global FP schedulability of periodic task sets. However, it cannot be it cannot be used for any exact schedulability test for the same. Note that global FP scheduling refers to global fixed-priority preemptive scheduling algorithm where the all the tasks are placed in a single global queue and are allowed to migrate from one processor to another according to the requirement [15].

3.5 FPZL

Fixed Priority until Zero Laxity (FPZL) [16] is a multiprocessor scheduling algorithm for real-time systems. It is called a "minimally dynamic" algorithm, meaning a job's priority can change at most once between its arrival and its completion. This approach is almost the same as global fixed priority preemptive scheduling algorithm with an addition. According to the FPZL algorithm if a task's laxity becomes zero that task should get the highest priority at that point. Fixed Priority until Static Laxity (FPSL) [16] and Fixed Priority until Critical Laxity (FPCL) [16] are similar to FPZL with some variation.

FPZL is also a global scheduling algorithm [5] in which a global queue of jobs is maintained and a single scheduler decides which processor will run which job. FPZL is based on the global fixed priority preemptive scheduling technique which is generally known as FP scheduling. In FPZL all the jobs are scheduled based on their fixed priority until their laxity becomes zero. When a job has zero laxity it must execute in order to avoid missing its deadline. Clearly, every task set that is FP schedulable is also FPZL schedulable.

In the case where a task set is FP-schedulable, FP is always preferred over FPZL. That is because FP algorithm is simpler than FPZL and checking for zero laxity just adds to the computation time and complexity. Hence if a task set is FP schedulable, FP is used over FPZL. In fact, the FPZL schedule will be the same as the FP schedule for FP-schedulable task sets using the same priority assignments. However, there exist task sets that are FPZL schedulable but not FP schedulable. Hence FPZL outshines the performance of global FP scheduling algorithms because both of them schedule the tasks almost in the same manner except for the fact that FPZL assigns the tasks with zero laxity at the highest priority which makes them meet their deadlines but the latter fails to make the task with zero laxity meet its deadline.

3.6 Dual Priority Scheduling

As the name suggests this algorithm paradigm assigns each task with two priorities as opposed to Fixed Priority algorithm [1] where each task has just one priority. This was developed to ensure that HRT jobs meet their deadlines and to improve the response time for all the non-real time jobs. According to Davis et al. [6] priority levels are split into three ranges: high, medium and low. Each HRT periodic task in Dual Priority algorithm is assigned with two priorities – one being in the lower priority range and another being in the higher priority range – as well as a priority promotion time. In addition all the non-real-time jobs execute at the medium priority level in first in first out (FIFO) order. The goal of this algorithm was to improve the response time of non-real-time jobs, while ensuring all the real time jobs still meet their deadlines.

Priorities assigned to the periodic task T_i are represented as $\tau_{i,high}$ and $\tau_{i,low}$, where $\tau_{i,high}$ is the high priority and $\tau_{i,low}$ is the low priority assigned of task T_i. Each periodic task T_i also has a priority promotion time associated with it which is denoted by λ_i. The promotion time plays a very vital role in the Dual Priority scheduling algorithm since it is used to determine if the task T_i is currently executing in the high priority or in the low priority level. Initially when the task arrives, it always executes in its low priority range $\tau_{i,low}$. If it does not complete its execution before its promotion time, it gets promoted as soon as it reaches its promotion time λ_i and from that point forward it continues executing at priority $\tau_{i,high}$ until completion of its execution. This algorithm makes sure that at any point in execution only the highest priority job executes and all the jobs continue executing in a priority driven manner [7]. Look at the example below [3].

Example 2 *Figure 3.4 illustrates a uniprocessor Dual Priority schedule. Let there be three tasks* $T_1(0, 5, 2)$, $T_2(0, 7, 1)$ *and* $T_3(0, 8, 3)$. *The promotion time of task* T_1 *is* $\lambda_1 = 2$, *promotion time of task* T_2 *is* $\lambda_2 = 1$ *and of task* T_3 *is* $\lambda_3 = 3$. *They are synchronous tasks meaning both arrive at the same time 0. Since task* T_1 *has higher priority than task* T_2 *and task* T_3 , *it starts executing. At time 1* T_2 *gets promoted and so it acquires the processor and starts executing. At time unit 2,* T_1 *gets promoted and since it has the highest priority level it starts executing and first completes its execution at time unit 3.* T_1 *and* T_2 *have finished executing by now and* T_3 *gets promoted and it continues executing until completion. Meanwhile second job of task* T_1 *keeps waiting and is given the processor only after* T_3 *completes since* T_3 *is promoted and* T_1 *is in its low priority level. The schedule continues in a similar manner.*

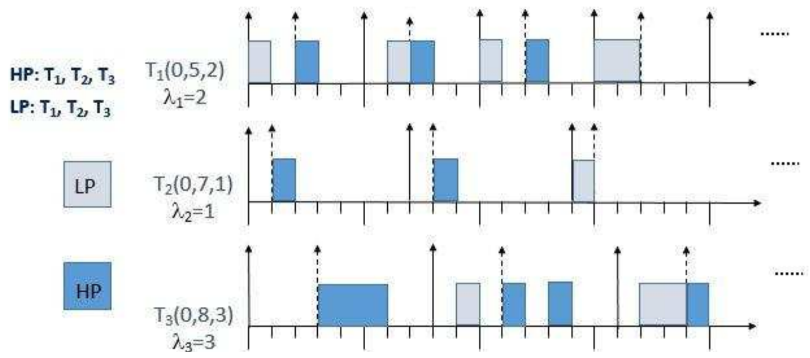


Figure 3.4: A sample Dual Priority schedule.

Though Dual Priority algorithm was originally designed to improve response times of non-real-time jobs and to ensure HRT tasks meet their deadlines for uniprocessors, it was applied in different ways to achieve distinct goals. With the help of Dual Priority scheduling, utilization bounds for RM scheduling with deferred preemptions were determined by Gopalakrishnan et al. [17]. Uniprocessor Dual Priority scheduling was extended to reduce power consumption by Jejurkar et al. [18]. WRCT analysis was performed by Bril et al. [19] for uniprocessor fixed priority systems with preemption.

3.7 PNPDP

The Partially Non Preemptive Dual Priority scheduling algorithm [20][21] is a developed version of Dual priority algorithm. PNPDP [20] has the same characteristics as the standard dual priority with a few differences which are discussed below. In PNPDP there are no non real-time jobs like SDP. The main difference between PNPDP and SDP is that lower priority jobs in PNPDP are not allowed to initiate preemptions. A task can only preempt another task when it is promoted to in its high priority level. However both high and low priority jobs can be preempted. Ho et al. [21][20] used worst case response time analysis developed by Guan et al. [12] in order to calculate promotion offset for every task. This approach is focused on reducing preemption and migration overhead.

Guan et al. [12] demonstrated a scheme for calculating interference of the high priority tasks on the low priority tasks in a fixed priority schedule. This approach was adopted to calculate the interference on the low priority tasks – i.e. the amount of time each low priority task waits for the higher priority task to complete its execution when interrupted. Worst case response time R_i of a task T_i is calculated by adding execution time e_i to its worst case interference I_i (i.e., $R_i = e_i + I_i$), where I_i is computed using Guan's method.

Ho et al. [21][20] concluded that when a task executes in its lower priority level, the worst case response time of the high priority promotion time of the task will always decrease. This follows before the value of e_i in Equation 3.3 will be smaller and the value of Ω_i will not increase. Hence a delay in promotion time will not cause any adverse impact on the schedulability – i.e., delaying the promotion time when the task is executing in its lower priority level would not cause the task to miss its deadline if it was meeting its deadline otherwise.

Consider the example below below [20].

Example 3 *Table 3.1 describes a task set to be scheduled on two processors. When PNPDP algorithm is run on this task set it generates the schedule illustrated in Figure 3.5. In the figure of the schedule the solid up arrow indicates the arrival of a new job, dashed up arrow means priority promotion, rectangles indicate that the job is executing in the rectangular indicative priority level.*

T_2 and T_3 arrive at $t = 0$ and start executing in their lower priority. T_1 arrives at $t = 1$. Similarly task T_0 arrives at time $t = 2$. Both task T_0 and task T_1 have higher priority than T_2 and T_3 but they do not preempt the lower priority tasks. Tasks T_2

Table 3.1: Task set for PNPDP example.

Task T_i	Period p_i	Execution Time e_i	WC Response Time R_i	Promotion Time λ_i
T_0	4	2	2	2
T_1	7	3	4	3
T_2	10	4	3	7
T_3	25	5	6	19

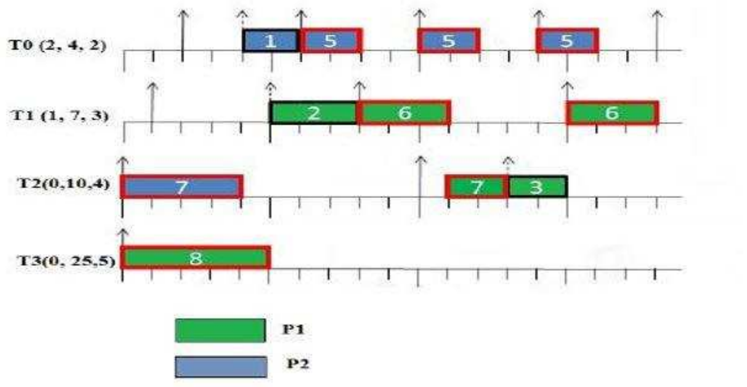


Figure 3.5: A sample PNPDP schedule.

and T_3 continue executing without any preemption until they reach their promotion time after λ_i time units have elapsed. At that point they get promoted to their high priority level and preemptions are initiated.

Chapter 4

Considering Low Priority Levels in Dual Priority Scheduling

Standard Dual Priority [6] was introduced in order to improve response times of non-real-time jobs without causing real-time jobs to miss their deadlines. Additionally, though, it has been observed that tasks which were not meeting its deadlines with Standard Fixed Priority were able to meet their deadlines using Standard Dual Priority. In fact, there is a conjecture that the Dual Priority algorithm is optimal for tasks executing on a uniprocessor [22].

The way high and low priority assignments are determined plays a key role in SDP. In fact the improvement in schedulability of the tasks from SFP [1] to SDP [6] is because of the way tasks are assigned priorities. To emphasize this point, Jadhav introduced Modified Dual Priority algorithm [3], an extension to SDP [6] which delays the priority promotion time. It is based on the idea that the more time a task spends executing in its low priority level the less interference it causes on the lower priority promoted tasks. Obviously the delayed promotion time was chosen in a manner where that delay does not cause a deadline miss for that particular task.

This thesis deals with various priority orderings for SDP when executed on multiprocessors and examines which priority ordering (if any) are the best in a given scenario. Number of processors, tasks per task set, etc were considered. To our knowledge, previous work on xprocessor DP scheduling assumed low priority ordering is the same as the high priority ordering for all the tasks. However, there is no reason this should be the case. For example, consider T_1, the task with the highest priority level. When this task is promoted, it will execute right away and will not be preempted. Therefore, there is no reason to let this task have the highest low priority level. What has been proposed in this thesis is that the priority ordering for lower priority range may be set using different techniques and some of them turn out to be more reasonable choices than setting up both the ranges in the same order. Ideally, this idea will make more tasks meet their deadlines under different conditions. All priority orderings below have been experimented under both implicit and constrained deadlines with different number of processors, tasks and utilization values. Different Low priority ordering techniques are listed below:

1. Same order as high priority (HP)
2. Reverse high priority order
3. Least execution time first (LEF)
4. Highest execution time first (HEF)
5. Least laxity first (LLF)
6. Highest laxity first (HLF)

Each of the low priority assignment orders has been performed with both OPA and RM being the high priority ordering. In all cases we use Guan's method [12] to determine the priority promotion time.

1. Same order as high priority

   In this mthod, high priority and low priority orders are the same. This is the typical priority ordering seen in many results for multiprocessor Dual Priority Scheduling [6].

2. Reverse HP

   Reverse HP means that the low priority order is reverse of the high priority order. The reverse priority is calculated as the difference between the number of tasks and the high priority order of the task. Let us consider an example to understand how reverse priority assignment works. Assume the high priority order assigned using OPA of tasks T_1, T_2, T_3 and T_4 is 3, 2, 0 and 1, respectively. If $T_{i,high}$ is the higher priority then $T_{i,low}$ = n - $T_{i,high}$ - 1 where n is the number of the tasks in the task set. So in the above example the low priority of $T_{1,low}$ = 4-3=1; $T_{2,low}$ = 4-2=2; $T_{3,low}$ = 4-0=4 and $T_{4,low}$ = 4-1=3. Therefore the low priority ordering using OPA-Reverse for tasks T_1, T_2, T_3 and T_4 would be 0, 1, 3 and 2 respectively. This technique balances the priority given to the task such that if it has highest priority when in high priority band then it should get lowest priority when in low priority band. Similarly if the task has lowest priority in the high priority band then it should have highest priority when in the low priority band. So any task will get a higher relative priority in the low priority range if it has a relative lower priority in high priority range.

3. Least execution time first (LEF)

   This is a simple strategy where the task with the least execution time is given the highest priority. It is intended to make the tasks wait for the least amount of time. So if the tasks which can quickly complete are chosen to execute over the tasks which would take comparatively longer to finish their execution.

This is a very intelligent strategy which has an impact on various factors of the scheduling algorithm. Firstly if the shorter tasks execute first, they are most likely to successfully complete their execution hence meeting their deadlines. It is also the periodic task equivalent to the a Shortest Processing Time First (SPF) [23] scheduling algorithm where we are eliminating the wait time of the shorter jobs so that they do not have to wait for longer jobs to execute before them. In real-time systems, when the execution time of the job is smaller as compared to the other task's execution time, it will sometimes miss its deadline due to waiting for the longer job to execute. Another advantage to it where if the tasks with shorter execution times are getting executed first then they would leave more amount of processor time for the longer jobs to execute without interruption in the higher priority band. Hence LEF is a simple technique to make more tasks meet their deadlines.

4. Highest execution time first (HEF)

   In HEF the tasks with higher execution time get a higher priority low priority level and the priority of the tasks decreases with the decreasing execution time. The longest job (job of the task with the highest execution time) has the highest low priority and the shortest job (job of the task with the least execution time) will get the lowest low priority. The idea behind this approach is that if the longest jobs execute first then the shortest ones can execute quickly without any interruptions. Naturally the longer jobs will cause more interruption than the shorter jobs. So they need to be executed before the shorter ones in order to avoid interference in the lower low priority tasks.

   The expectation is that if the longer jobs to have higher priority in low priority band then they will execute more before being preempted and therefore put less demand on the system when executing in the high priority band.

5. LLF

   Recall the laxity of a task in real-time systems can be defined as the amount of the time a task can wait before executing and still meet its deadline. When a job of task T i initially released, its laxity is $D_i - e_i$. So in LLF the low priority ordering is from the smallest $D_i - e_i$ to the largest.

   While the high priority order of the tasks are ordered using either OPA or RM the low priority order of the task can be defined using laxity. Because laxity is a measure of task's slack, it is reasonable to use it as a priority metric. In fact Liu and Layland introduced the Least laxity First algorithm in this seminar paper [1].

6. HLF

   This strategy is the reverse of the LLF technique. In this case, tasks with higher laxity have higher priority level in the low priority band.

Chapter 5

Experiments and Results

Standard Dual Priority [6] is simulated in various scenarios using the low priority orderings discussed above. The process followed for performing experiments is explained below:

- **Task set generation:** First the task utilization values are randomly generated using Stafford's *randfixedsum* procedure [24]. n, m, s, a, b which generates a $n \times m$ array of values between a and b whose sum is s. For our purpose we let $n = 2, 5$ or 10 for different scenarios, $m = 4, 8, 16$ or 32. We always had $a = 0$ and $b = 1$. The sum was our target, which ranges from $0.6 \times m$ to $0.98 \times m$. The total utilization is calculated as the number of processors times the percentage of the utilization. For example if there are 4 processors and the utilization is about 50% then the total utilization would be $4 \times 0.50 = 2.0$. Consider a case where the system has 8 tasks per task set then using Random Fixed Sum technique 8 (equal to number of tasks per taskset) values are generated which would sum up to 2.0 (total utilization). These utilization values are then used to determine the execution time of the task.

  Task sets are then derived by randomly generating periods and computing the execution times from the utilizations and the periods ($e = u \cdot p$). The period is a random number between 1 and 200. Note that task index for tasks $\{T_1, T_2, \ldots T_n\}$ would be $\{0, 1, \ldots n-1\}$. The deadline of a task is determined in two ways. The experiments are conducted using implicit deadlines, where deadline is equal to period ($D_i = p_i$) and constrained deadlines, where deadlines are less than or equal to the periods ($D_i \leq p_i$).

- **Simulation:** These tasks were run on Standard Fixed Priority scheduling algorithm. Only the task sets that were not schedulable with SFP i.e., SFP - unschedulable task sets were taken in account for further experimenting. The idea behind using only the tasks which are SFP unschedulable is because SDP is a variation of SDP and is used to make the tasks meet their deadlines which did not meet their deadlines with SFP.

The SFP-unschedulable tasks were then simulated under Standard Dual Priority [6] under multiple scenarios that are illustrated below:

- Number of processors $m = 4, 8, 16, 32$.
- Number of tasks $n = 2 \times m, 5 \times m, 10 \times m$.
- Number of utilization values $u = 0.625 \times m, 0.675 \times m, 0.725 \times m, 0.775 \times m, 0.825 \times m, 0.875 \times m, 0.925 \times m, 0.975 \times m$.

  As shown above the utilization values are taken in such a way that they consider the case where the processor utilization range from about 62% to 98%. The utilization bound was proposed by Phillips et al. [25] which should start from 50% and range up to 98%. However this was a pessimistic approach and so it was decided to consider utilization bounds to start at around 60% because for 50% - 60% utilization it was very challenging to obtain SFP-unschedulable

Table 5.1: Scenario

Scenario	SFP-unschedulable (%)
p4n8u0.525	6
p4n8u0.575	14
p4n8u0.625	15
p8n16u0.525	0.2
p8n16u0.575	2
p8n16u0.625	5.3
p8n16u0.675	11.6
p16n32u0.525	0.5
p16n32u0.575	3.1
p16n32u0.625	8
p16n32u0.675	15.9

tasks. Also in some of the cases the utilization bound would just start with later values and not 62% for the same reason.

- High priority ordering: Optimal Priority Assignment [15] and Rate monotonic [1].

- Low priority ordering: All the low priority orderings discussed in Chapter 4 were experimented with both OPA and RM as the high priority order. One optimal i.e., OPA prioriy ordering and another not optimal technique i.e., RM was considered.

- Deadlines: Implicit and constrained.

  All the experiments were performed with both implicit and constrained deadlines. Implicit deadlines just make the deadline be equal to period. Constrained deadlines were determined finding a random number in the range $[f(u), p]$, where

$$f(u) = 2(p - e) \times u + (2e - p) \tag{5.1}$$

and u is the utilization percentage of the processor, p is the period of the task, e is the execution time of the task.

- Number of runs $r = 1000$ when m is 4, 8, 16 and 500 when m is 32.

  1000 runs were performed for each scenario except for the experiments with 32 processors because of the time constraints. it took a long time (up to 5 or 6 hours per experiment) to run.

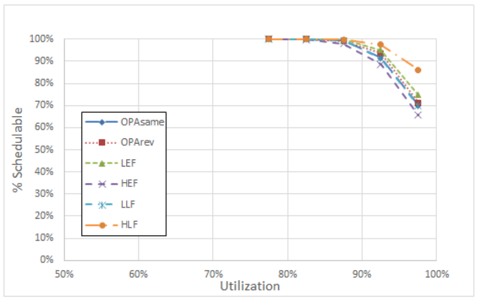
Each scenario was captured using the percentage of the schedulable tasks obtained from 1000 runs versus the percentage of the processor utilization.

5.1 High Priority Order : OPA, Deadline: Implicit

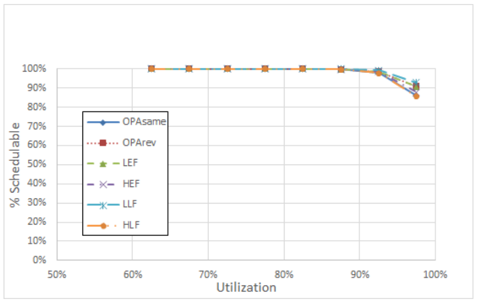
Below, the different scenarios are explored separately, according to the number of processors.

5.1.1 4 Processors

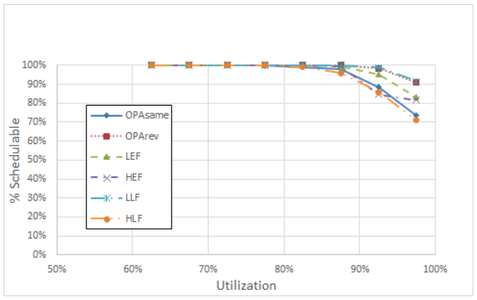
The x-axis in the graph shows the utilization percentage of the processor and the y-axis shows the percentage of the tasks schedulable (out of 1000). Each line in the graph represents the result of assigning low priority technique and compares its performance with various other low priority ordering while OPA remains the high priority ordering strategy. The low priority orders include same as OPA, reverse of OPA, Least Execution Time First, Highest Execution Time First, Least Laxity First, and Highest Laxity First. The first low priority ordering where the low priority band has the same priority order as high priority order which in this case is OPA,



(a) 8 tasks



(b) 20 tasks



(c) 40 tasks

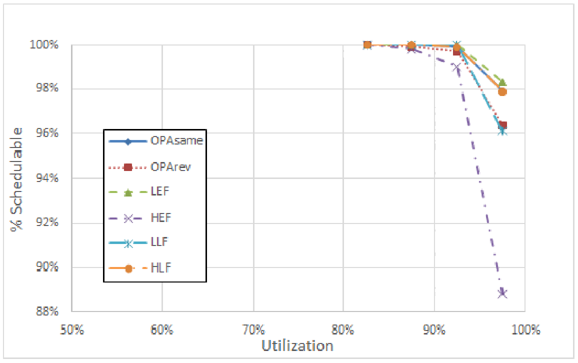
Figure 5.1: Simulations on 4 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Implicit" scenario.

is nothing but the original Dual Priority algorithm whose performance is compared with respect to other low priority orderings and the same high priority order.

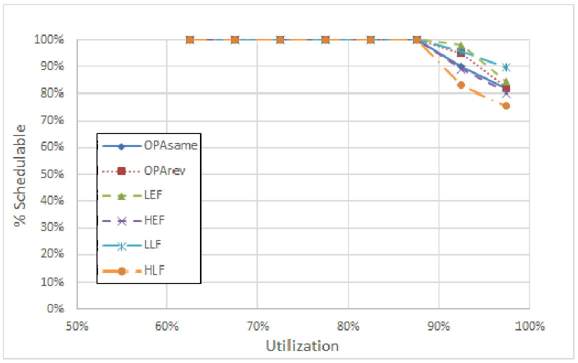
In Figure 5.1(a) where the number of tasks are twice the number of processors does not really contribute towards the conclusion since it is a trivial case where there are very few processors and the number of tasks are also few. On an average each processor has 2 tasks to to schedule which is fairly easy and possible. But when there are 20 tasks and 40 tasks per task set then the average number of tasks on each processor increases to 5 and 10 for $n = 20$ and $n = 40$ respectively. It can be noticed that Least Laxity First performs the best in Figure 5.1(b) and both Least Laxity First and OPA-Reverse performs the best in Figure 5.1(c). They are very much in accordance to the expected results. We expected the promising low priority orderings to be both LLF and the reverse low priority and in it did turn out in the way it was expected to be. That's because laxity is a strong factor to be considered when dealing with deadline oriented tasks. If the task with the least laxity is executed first then the tasks are more likely to meet their deadlines as explained in Chapter 4. Also reversing the high priority seems to be a reasonable choice when compared to other low priority orderings. Therefore, it is intuitive to expect that setting up the low priority order using LLF or OPA-Reverse will improve the performance of traditional SDP.

5.1.2 8 Processors

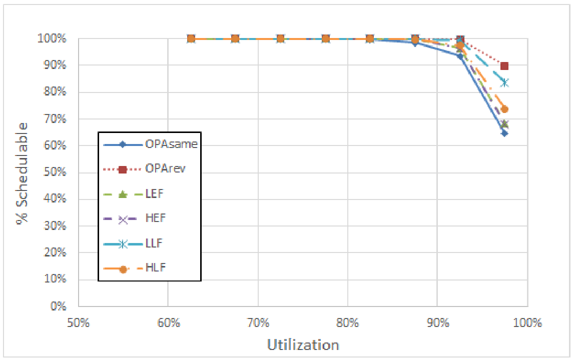
Similar to the first case, the result with number of tasks being twice the number of processors is very trivial. OPA-Reverse and LLF perform nearly the same but are not the best ones among all the other orderings in that case. However in $p = 8, n = 40$ and $p = 8, n = 80$, LLF and OPA-Reverse are the two approaches that perform best. They outperform all the other low priority orderings and definitely improve



(a) 16 tasks



(b) 40 tasks



(c) 80 tasks

Figure 5.2: Simulations on 8 processors with various numbers of tasks for the “High Priority Order: OPA, Deadline: Implicit” scenario.

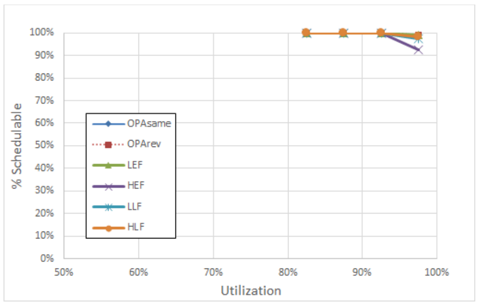
SDP's performance. In Figure 5.2(b), observe that OPA-Reverse performs the same as OPA-Same – i.e., the original SDP – but LLF surely makes more task sets meet their deadlines which were not able to be schedulable using SDP. In Figure 5.2(c), OPA-Same performs extremely poorly while OPA-Reverse performs the best which has again proved to be a wise low priority ordering to be chosen while simulating SDP.

5.1.3 16 Processors

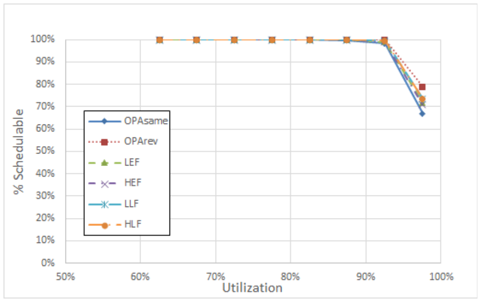
This case presents a strong example of how well OPA-Reverse performs. In the trivial case, Figure 5.3(a), OPA-reverse performs the best but the other priority orderings perform almost the same as OPA-Reverse. However in both Figures 5.3(b) and 5.3(c), OPA-Reverse turns out to be the best low priority ordering technique. It performs better than LLF also. Hence reversing the high priority order in low priority ordering might be very advantageous when powerful techniques like LLF fail to perform.

5.1.4 32 Processors

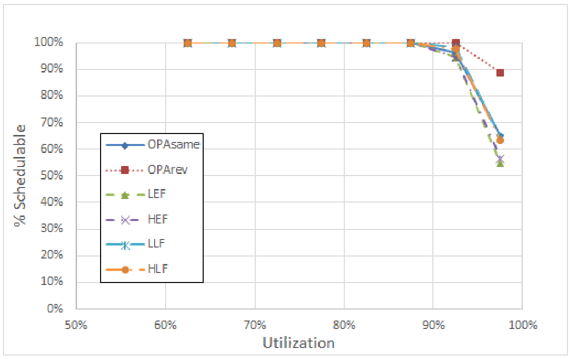
As there has been a pattern of the trivial case making all the low priority orderings perform more or less in the same manner follows in this case too. All the low priority orderings perform almost the same providing no contribution towards the conclusion. In Figure 5.4(b), there appears to be a large difference in the performance results of each low priority ordering, with OPA-Reverse performs slightly better than other strategies. OPA-Reverse outperforms all the other low priority orderings in Figure 5.4(c) with highest number of processors and highest number of tasks per task set considered. Hence reversing the priority order in low priority band can make a lot of difference in many cases.



(a) 32 tasks

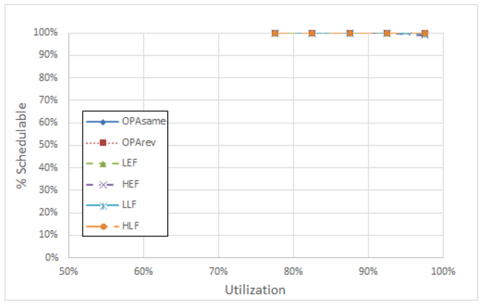


(b) 80 tasks

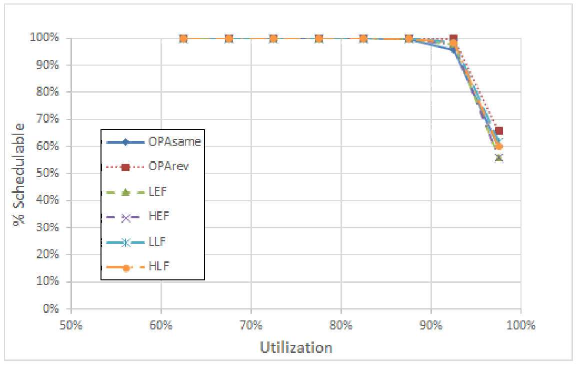


(c) 160 tasks

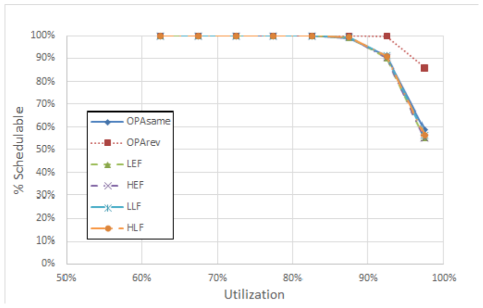
Figure 5.3: Simulations on 16 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Implicit" scenario.



(a) 64 tasks



(b) 160 tasks



(c) 320 tasks

Figure 5.4: Simulations on 32 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Implicit" scenario.

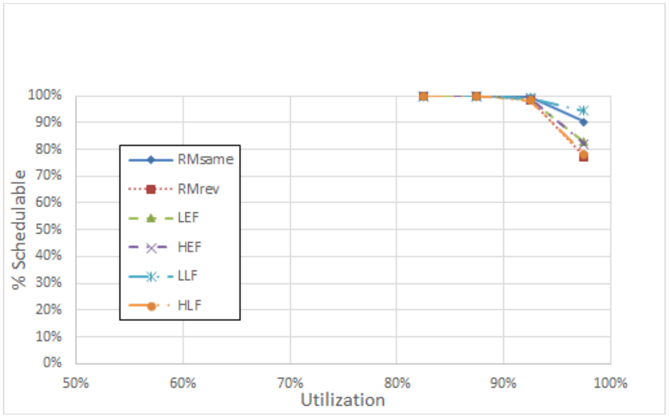


Figure 5.5: Percent of schedulable task sets with various low priority settings for 4 processors and task sets of 40 tasks for the "High Priority Order: RM, Deadline: Implicit" scenario.

5.2 High priority order : RM, Deadline: Implicit

Going forward only one figure for each of the processor scenarios is depicted. The results for the same number of processors with fewer task sets are similar to the ones given below. All the graphs representing experimental results can be found in Appendix. Only the extreme cases with number of tasks per task set is 10 times the number of processors are illustrated below with increasing number of processors.

5.2.1 4 Processors

In Figure 9(c), it is very clear than RM in its traditional way works better than the other low priority ordering techniques. Until a certain point in the graph all the low priority orderings perform in a similar manner because the processor utilization is relatively low. However when it comes to more processors they start varying in

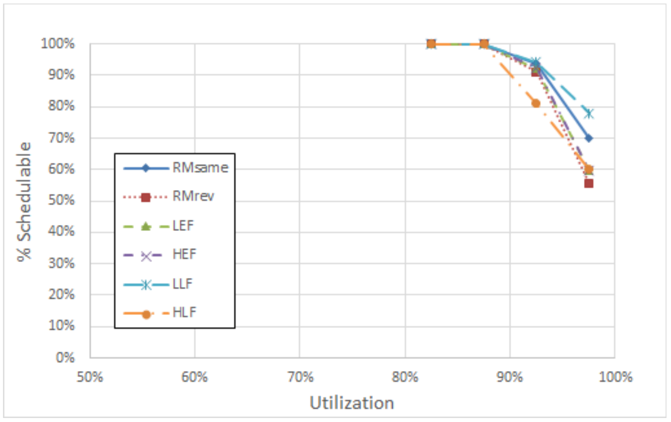


Figure 5.6: Percent of schedulable task sets with various low priority settings for 8 processors and task sets of 80 tasks for the “High Priority Order: RM, Deadline: Implicit” scenario.

their performance. LLF low priority ordering performs better than all the other low priority orderings and it does improve the original SDP to a certain extent when the utilization bound gets higher. On the other hand RM-Reverse performs the worst. This might be because RM itself is not an optimal priority ordering technique so reversing a non-optimal priority order in the low priority band does not provide a strong reason to improve the performance.

5.2.2 8 Processors

Figure 10(c) is very similar to the previous case where original RM performs better than most of the other low priority ordering techniques but LLF does perform even better than all the rest of the low priority orderings and so LLF can be a safe choice for low priority ordering in SDP when considering RM as high priority ordering.

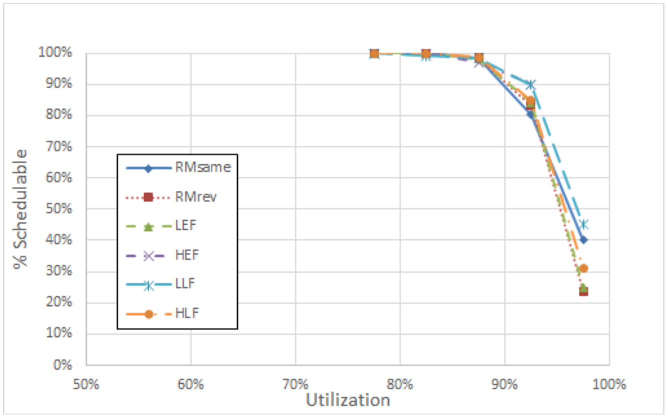


Figure 5.7: Percent of schedulable task sets with various low priority settings for 16 processors and task sets of 160 tasks for the “High Priority Order: RM, Deadline: Implicit” scenario.

5.2.3 16 Processors

Figure 11(c) clearly depicts that LLF is a certainly a fair choice to make over all the low priority orderings when RM algorithm defines the high priority ordering. It is very likely according to the results that RM-Reverse is going to perform the worst and hence should not be considered.

5.2.4 32 Processors

In Figure 12(c), the performance of the tasks is very difficult to be gauged in this case because there is no one low priority ordering that performs well throughout. All of them start in the same manner as they did for other cases being almost equal for lower utilization bounds. LLF turns out to perform decently for about 80% to 90% of the processor utilization but its performance drops as the utilization bound

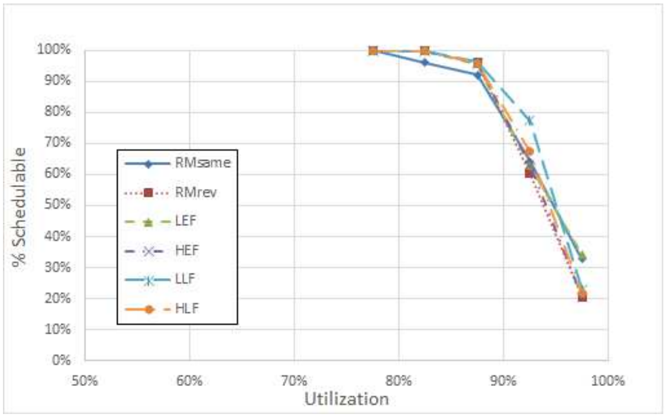


Figure 5.8: Percent of schedulable task sets with various low priority settings for 32 processors and task sets of 320 tasks for the "High Priority Order: RM, Deadline: Implicit" scenario.

increases. That might be because of the increase in number of processors and tasks to a high extent. This is an unusual behavior but out of all the choices, LLF turns out to be the better one if not the best choice.

5.3 High priority order : OPA, Deadline: Constrained

5.3.1 4 Processors

In Figure 1(c), the performance of OPA is very similar under implicit and constrained deadlines. In this case OPA-Reverse and LLF perform exactly the same again proving that choosing an intelligent low priority ordering when high priority order is determined but OPA makes more tasks in SDP meet their deadlines.

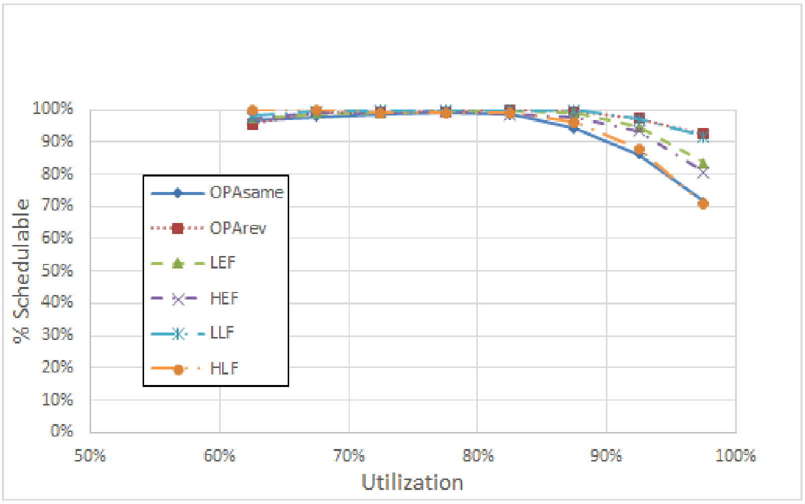


Figure 5.9: Percent of schedulable task sets with various low priority settings for 4 processors and task sets of 40 tasks for the “High Priority Order: OPA, Deadline: Constrained” scenario.

5.3.2 8 Processors

LLF has often performed same as OPA-Reverse or even better than that but there are cases when OPA-Reverse has has undoubtedly be better than LLF, as seen in Figure 2(c), According to the results so far, it is always preferable to choose LLF as the low priority ordering technique when considering OPA as high priority order but if the time and resources permit, OPA-Reverse should be the second best choice which may give even better results than LLF sometimes. OPA-same performs the worst. It means that using any other low priority technique than the same as high priority order technique will make some significant improvement in the performance of SDP.

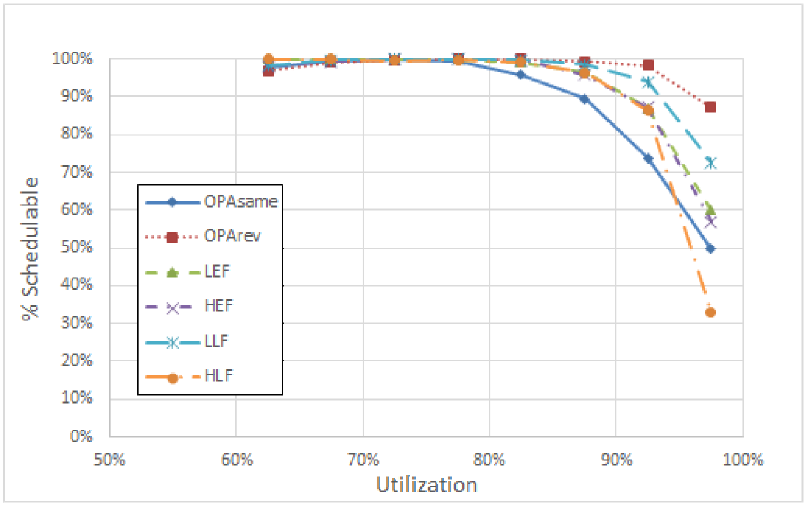


Figure 5.10: Percent of schedulable task sets with various low priority settings for 8 processors and task sets of 80 tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.

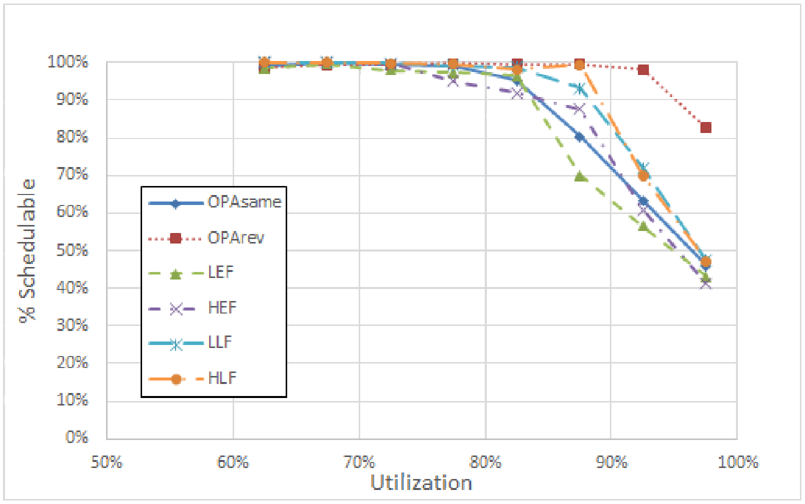


Figure 5.11: Percent of schedulable task sets with various low priority settings for 16 processors and task sets of 160 tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.

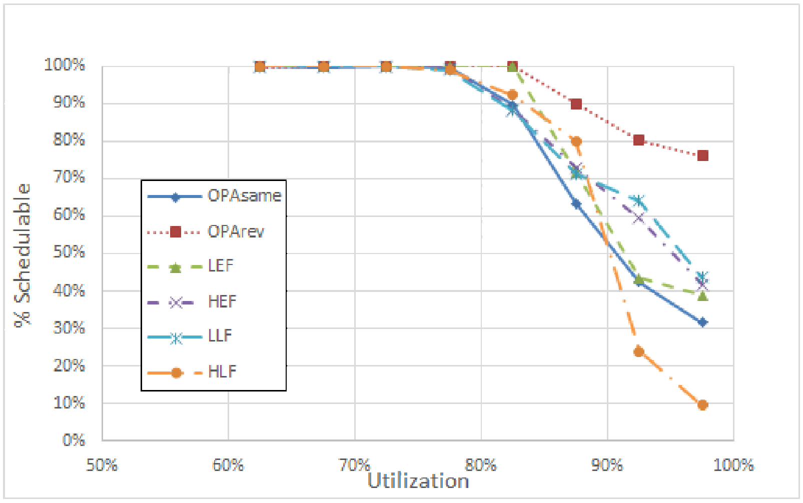


Figure 5.12: Percent of schedulable task sets with various low priority settings for 32 processors and task sets of 320 tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.

5.3.3 16 Processors

Figure 3(c) shows that the SDP performs poorly when compared to most of the low priority orderings. OPA-Reverese stands out in the figure above making it as reasonable choice as LLF.

5.3.4 32 Processors

Figure 4(c) HLF performs the worst in most of the cases and that is obvious because if the job that can wait for most amount of time to get executed is scheduled first then the jobs that need to get executed earlier or have cannot afford to wait for the execution will certainly miss their deadlines. No other low priority technique is any closer to the performance of OPA-Reverse in most of the cases which is why it should be chosen over LLF when OPA is high priority order and the deadlines are constrained.

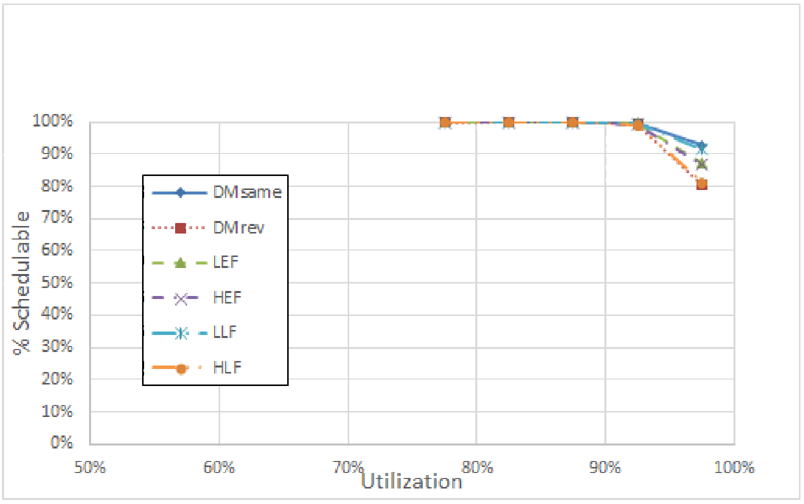


Figure 5.13: Percent of schedulable task sets with various low priority settings for 4 processors and task sets of 40 tasks for the "High Priority Order: DM, Deadline: Constrained" scenario.

5.4 High priority order : DM, Deadline: Constrained

5.4.1 4 Processors

The results in Figure 5(c) are interesting because DM performs nearly equal to the performance of good low priority orderings, such as LLF. As expected the RM-Reverse would perform the worst, but the LLF ordering and DM-same ordering is respectable. This may be because RM has not optimal and the behaves unexpectedly under constrained deadlines.

5.4.2 8 Processors

Figure 6(c) shows a very similar result as the previous case where the original DM with its high and low priority order the same as DM performs better than all the

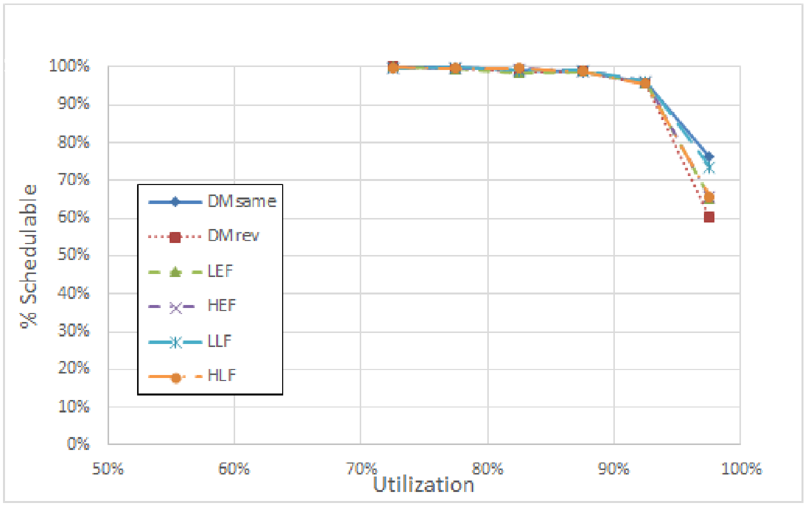


Figure 5.14: Percent of schedulable task sets with various low priority settings for 8 processors and task sets of 80 tasks for the “High Priority Order: DM, Deadline: Constrained” scenario.

other orderings. Again LLF is reasonably close but still not as good as the original SDP. Therefore letting the original algorithm be as it is when DM is the high priority order and the deadlines are constrained is a good option.

5.4.3 16 Processors

The results in Figure 7(c) are quite similar to those in Figure 6(c) apart from the fact that at this point it can be concluded that reversing the high priority order and determining low priority order according to that works best in case of OPA but works worst in case of DM.

5.4.4 32 Processors

With the increase in number of processors and the number of tasks the behavior of DM gets unpredictable, as seen in Figure 8(c). However traditional SDP performs

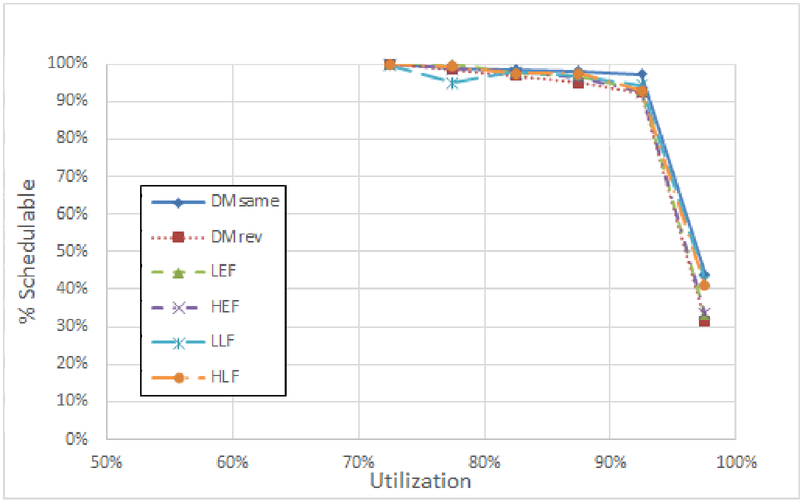


Figure 5.15: Percent of schedulable task sets with various low priority settings for 16 processors and task sets of 160 tasks for the “High Priority Order: DM, Deadline: Constrained” scenario.

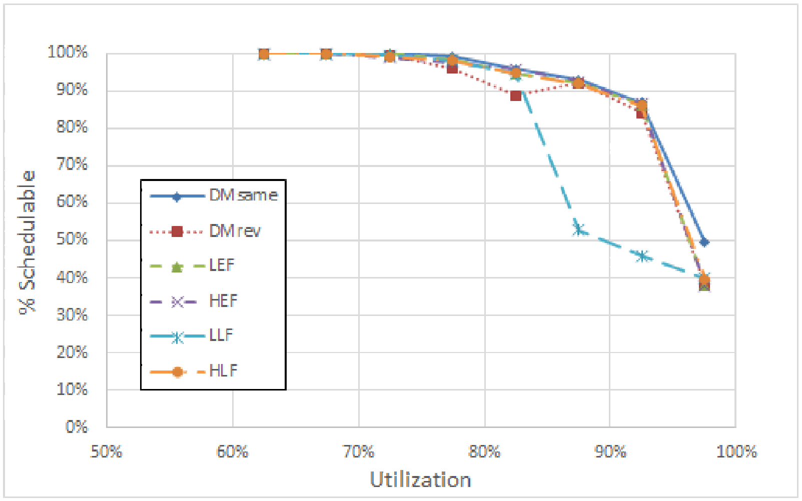


Figure 5.16: Percent of schedulable task sets with various low priority settings for 32 processors and task sets of 320 tasks for the “High Priority Order: DM, Deadline: Constrained” scenario.

better than the other low priority ordering techniques. It is safe to say that if a non-optimal high priority ordering technique is used under constrained deadlines, changing the low priority order might just not turn out to perform as expected.

Chapter 6

Conclusion

This thesis explored the problem of priority assignment techniques in Dual Priority algorithm and made an effort to improve the algorithm. This was achieved by having the low priority order be set according to different techniques. It compares the performance of different low priority ordering techniques in Dual Priority algorithm.

Experimental results demonstrate that there no single low priority ordering technique that outperforms all the other low priority techniques. The optimal low priority technique depends on the number of processors and number of tasks being considered. Hence in different scenarios with different number or processors and tasks different low priority ordering techniques work the best.

The expectation was that reversing the high priority order (OPA-Reverse/RM-Reverse) or Least Laxity First (LLF) would have worked the best in every case. The experimental results do not depict reverse priority of least laxity the best choice but they do perform well in nearly all the scenarios. The low priority ordering by reversing the high priority or according to least laxity first technique does improve schedulability of the tasks as compared to Standard Dual Priority. It makes many of the tasks that were unschedulable with traditional dual priority be schedulable by changing the low priority ordering. It also performs better than most of the other low priority orderings considered in the above experiments.It has been observed that the results differ when the deadlines are made constrained from implicit which may have an adverse effect on the performance of the algorithm as in the case of RM under constrained deadlines.

Hence it can be concluded that the reasonable choice for low priority ordering in Dual Priority Algorithm would be reversing the high priority order when considering OPA under constrained deadlines or Least Laxity First ordering when considering OPA/RM under implicit deadlines. Various other low priority ordering techniques can also be considered to improve the performance of original Dual Priority algorithm but in general LLF and OPA/RM-Reverse outperform other low priority orderings most of the time.

REFERENCES

[1] C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] R. Mall, *Real-Time Systems: Theory and Practice*. Prentice Hall, 2010.

[3] M. Jadhav, “Improving dual-priority scheduling,” Master’s thesis, University of Georgia, 2014.

[4] A. Mok, *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[5] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, “A categorization of real-time multiprocessor scheduling problems and algorithms,” in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis* (J. Leung, ed.), CRC Press LLC, 2003.

[6] R. Davis and A. Wellings, “Dual priority scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[7] J. W.-S. Liu, *Real-Time Systems*. Upper Saddle River, New Jersey 07458: Prentice-Hall, Inc., 2000.

[8] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, “Real-time scheduling: The deadline-monotonic approach,” in *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[9] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” in *IEEE Real-Time Systems Symposium (RTSS)*, 1989.

[10] M. Bertogna and M. Cirinei, “Response time analysis for globally scheduled symmetric multiprocessor platforms,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2007.

[11] M. Bertogna, M. Cirinei, and G. Lipari, “Schedulability analysis of global scheduling algorithms on multiprocessor platforms,” *IEEE Transactions on Parallel and Distributed Systems.*

[12] N. Guan, M. Stigge, W. Yi, and G. Yu, “New response time bounds for fixed priority multiprocessor scheduling,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[13] N. C. Audsley, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times,” December 1991.

[14] N. Audsley, “On priority assignment in fixed priority scheduling,” *Information Processing Letters*, vol. 79, May 2001.

[15] R. Davis and A. Burns, “Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems,” in *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[16] R. Davis and S. Kato, "FPSL, FPCL and FPZL schedulability analysis," *Real-Time Systems*, vol. 48, November 2012.

[17] R. Gopalakrishnan and G. Parulkar, "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.

[18] R. Jejurikar and R. Gupta, "Procrastination scheduling in fixed priority real-time systems," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004.

[19] R. Bril, J. lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems*, vol. 42, August 2009.

[20] C. Ho, *Reducing Scheduling Overheads in Multiprocessor Real-Time Systems.* PhD thesis, University of Georgia, 2013.

[21] C. Ho and S. Funk, "Partially non-preemptive dual priority multiprocessor scheduling," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2011.

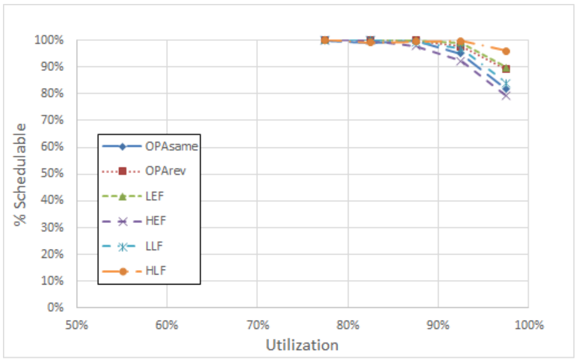
[22] A. Burns, "Dual priority scheduling: Is the processor utilisation bound 100%?," in *Proceedings of the International Real-Time Scheduling Open Problems Seminar*, 2010.

[23] M. Pinedo. Springer, 4th ed., 2010.

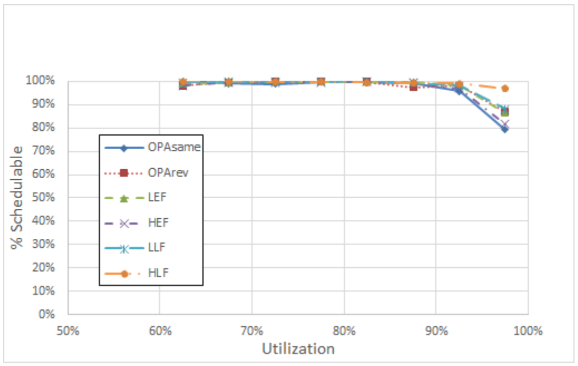
[24] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, 2010.

[25] C. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *ACM Symposium on the Theory of Computing (STOC)*, 1997.

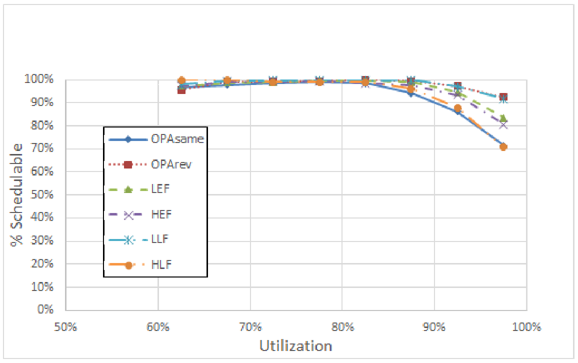
The results of all the experiments being performed presented in this chapter except for the results of the low priority orderings tested with high priority order as OPA and the deadlines being implicit as they have already being discussed in the previous chapter.



(a) 8 tasks

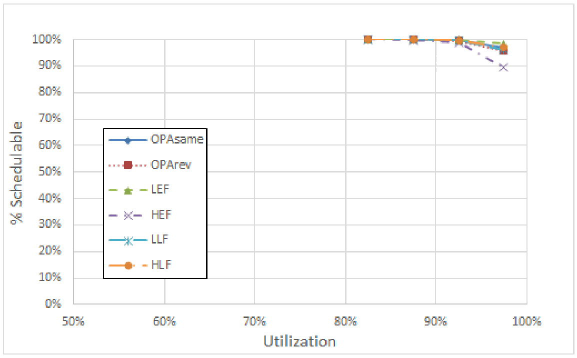


(b) 20 tasks

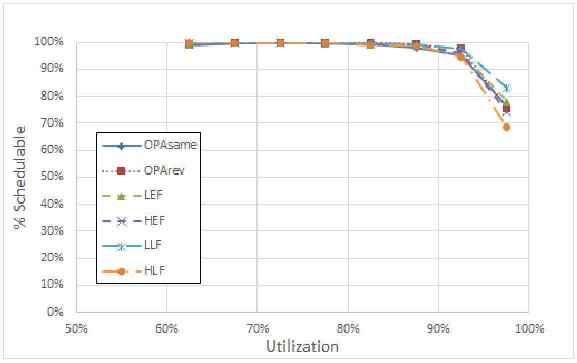


(c) 40 tasks

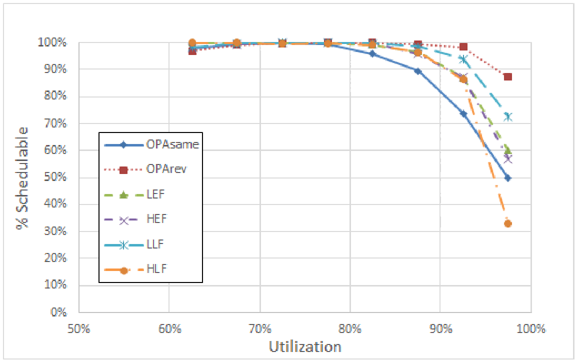
Figure 1: Simulations on 4 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.



(a) 16 tasks

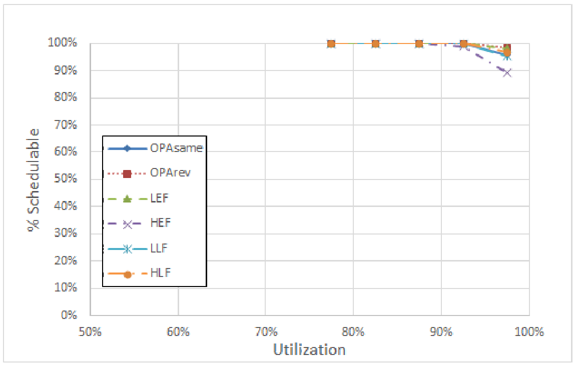


(b) 40 tasks

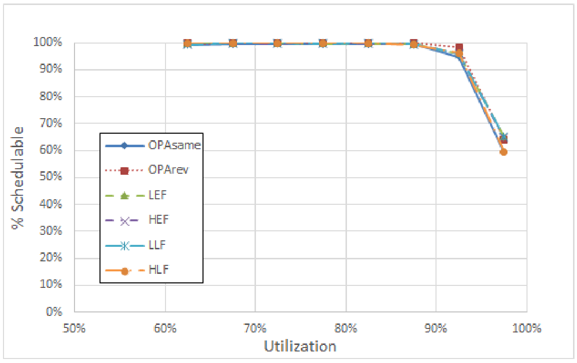


(c) 80 tasks

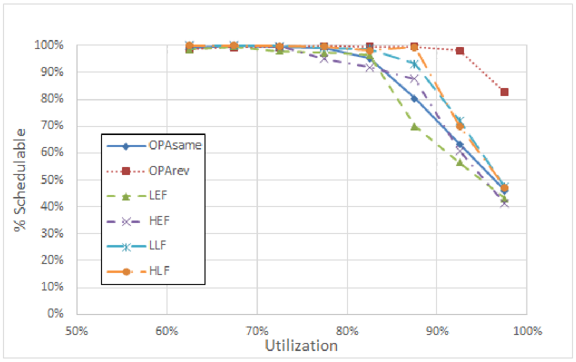
Figure 2: Simulations on 8 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.



(a) 32 tasks

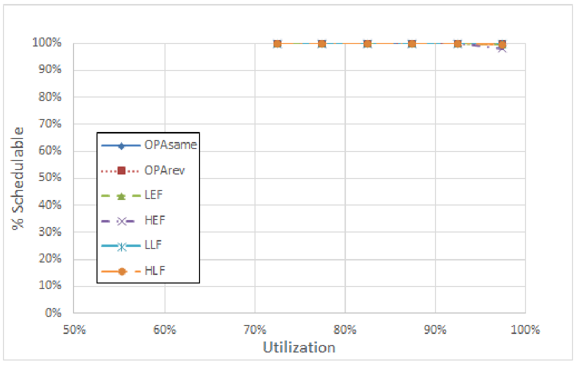


(b) 80 tasks

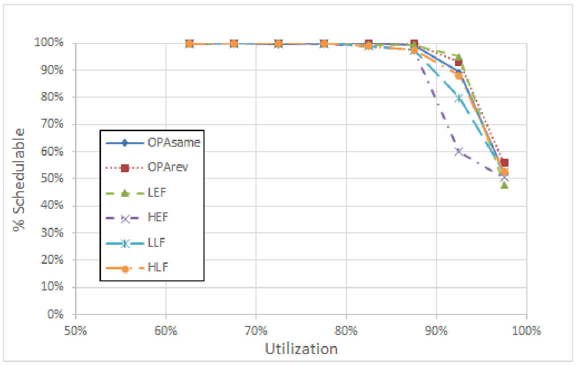


(c) 160 tasks

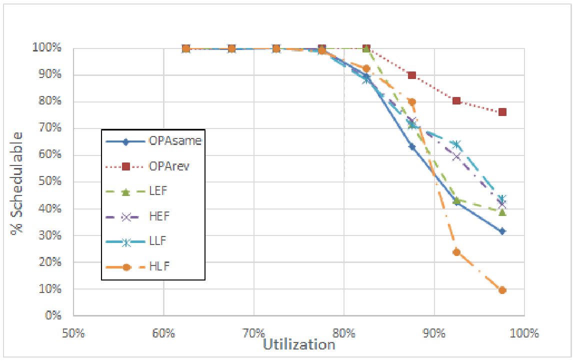
Figure 3: Simulations on 16 processors with various numbers of tasks for the “High Priority Order: OPA, Deadline: Constrained” scenario.



(a) 64 tasks

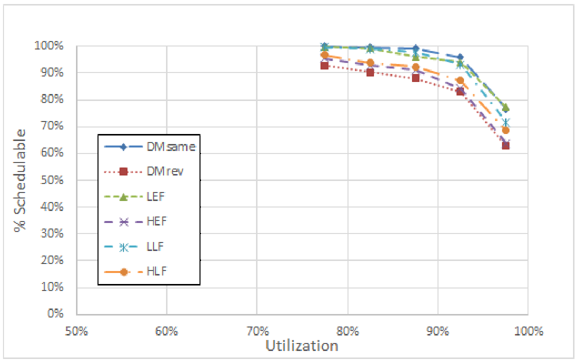


(b) 160 tasks

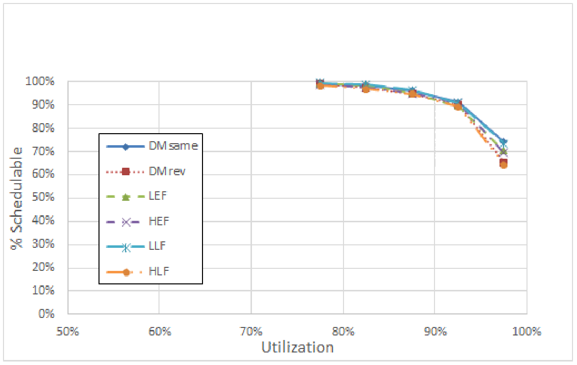


(c) 320 tasks

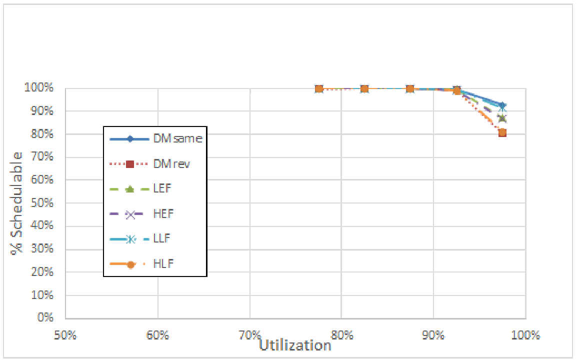
Figure 4: Simulations on 32 processors with various numbers of tasks for the "High Priority Order: OPA, Deadline: Constrained" scenario.



(a) 8 tasks

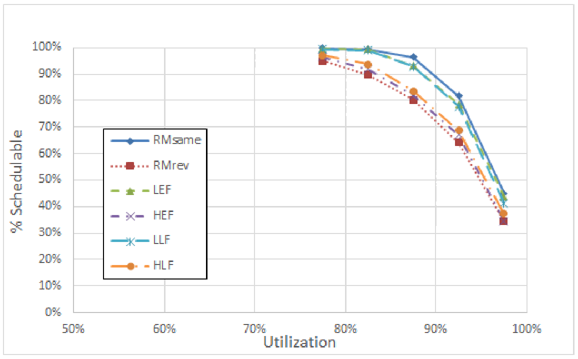


(b) 20 tasks

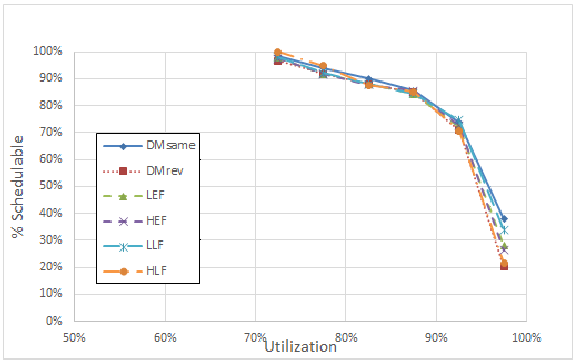


(c) 40 tasks

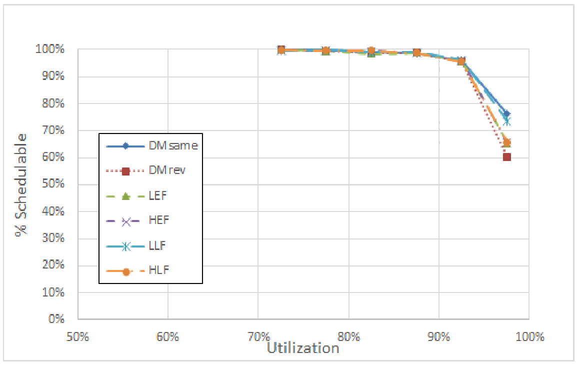
Figure 5: Simulations on 4 processors with various numbers of tasks for the "High Priority Order: DM, Deadline: Constrained" scenario.



(a) 16 tasks

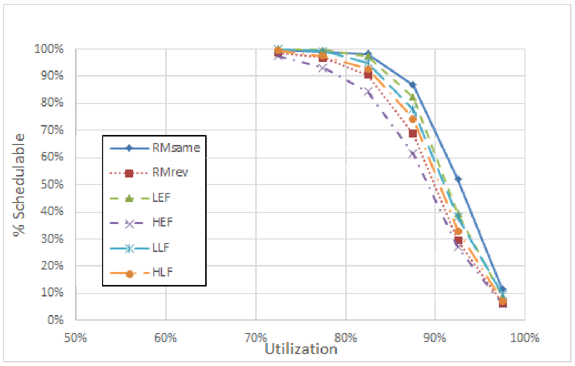


(b) 40 tasks

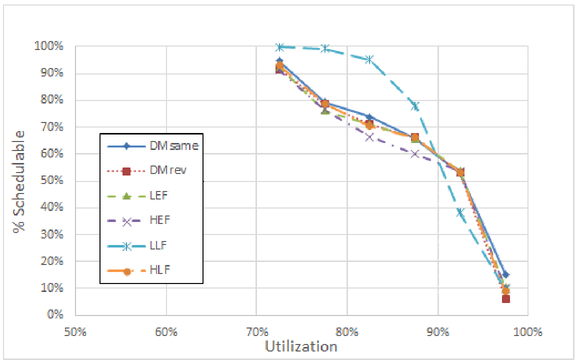


(c) 80 tasks

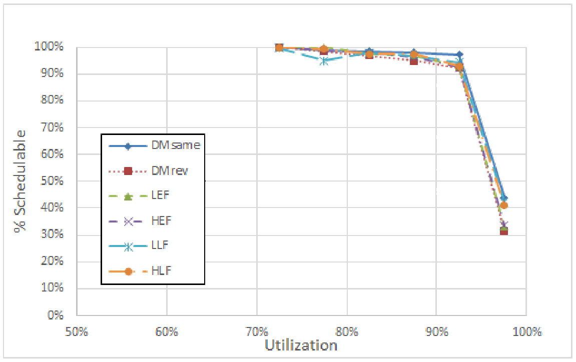
Figure 6: Simulations on 8 processors with various numbers of tasks for the “High Priority Order: DM, Deadline: Constrained” scenario.



(a) 32 tasks

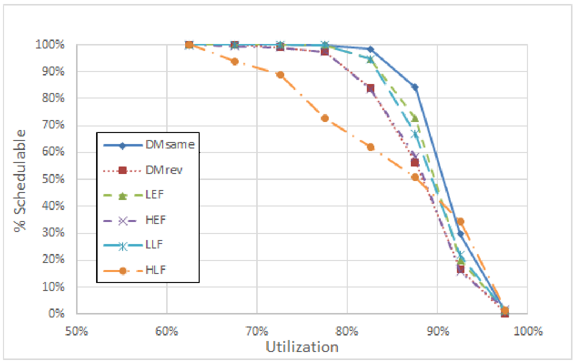


(b) 80 tasks

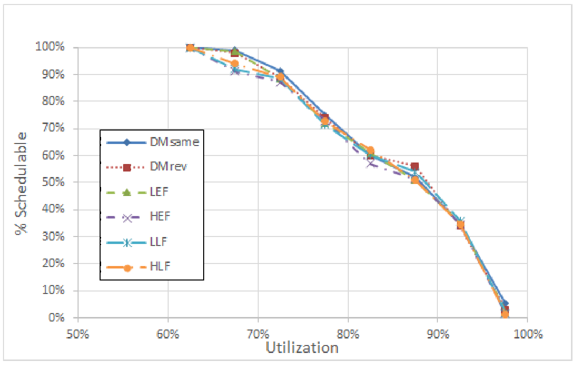


(c) 160 tasks

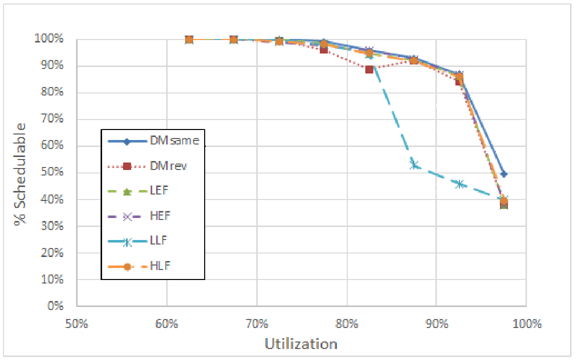
Figure 7: Simulations on 16 processors with various numbers of tasks for the "High Priority Order: DM, Deadline: Constrained" scenario.



(a) 64 tasks

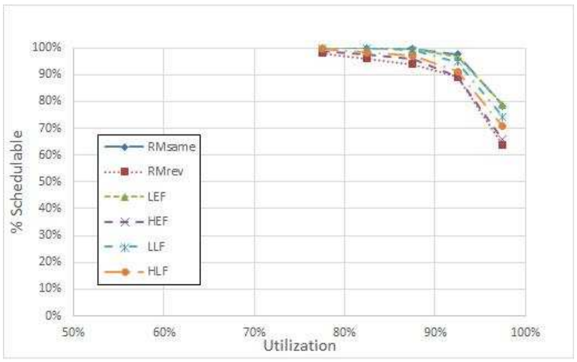


(b) 160 tasks

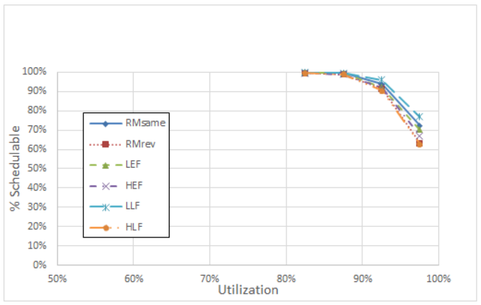


(c) 320 tasks

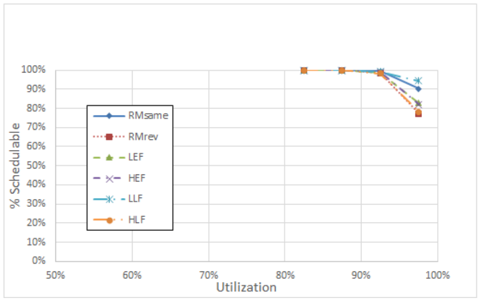
Figure 8: Simulations on 32 processors with various numbers of tasks for the "High Priority Order: DM, Deadline: Constrained" scenario.



(a) 8 tasks

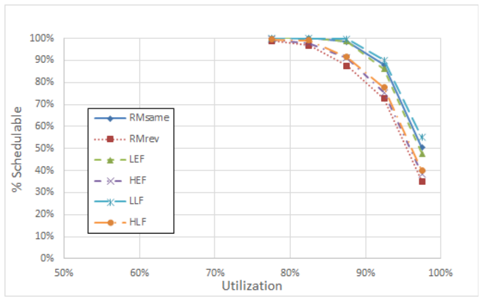


(b) 20 tasks

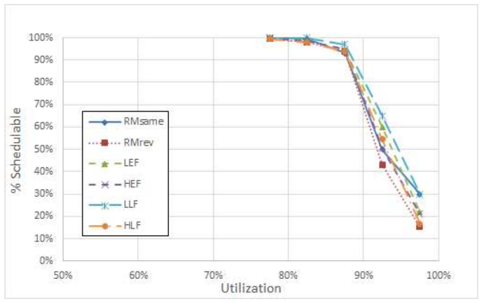


(c) 40 tasks

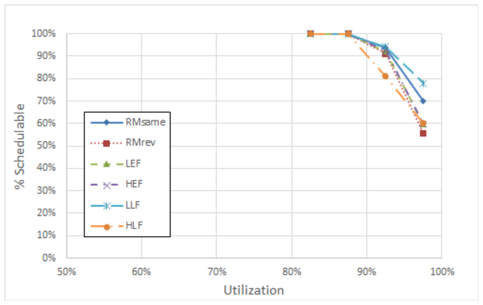
Figure 9: Simulations on 4 processors with various numbers of tasks for the “High Priority Order: RM, Deadline: Implicit” scenario.



(a) 16 tasks

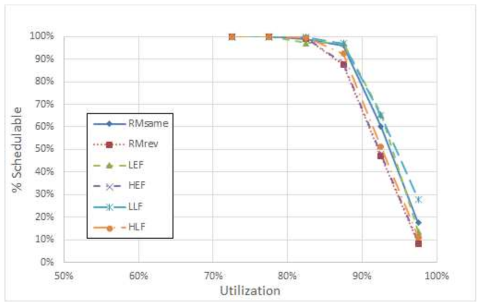


(b) 40 tasks

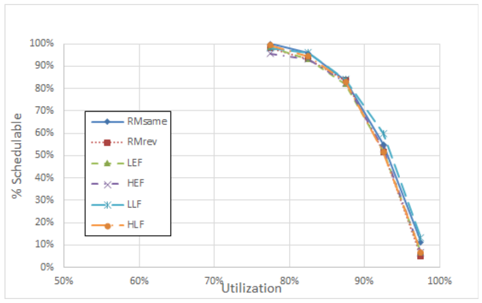


(c) 80 tasks

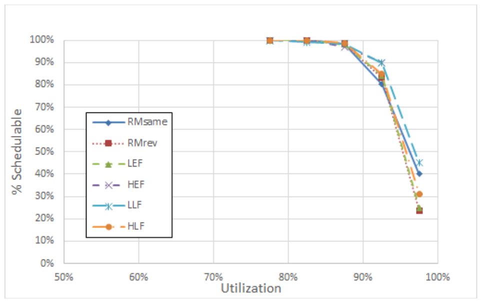
Figure 10: Simulations on 8 processors with various numbers of tasks for the “High Priority Order: RM, Deadline: Implicit” scenario.



(a) 32 tasks

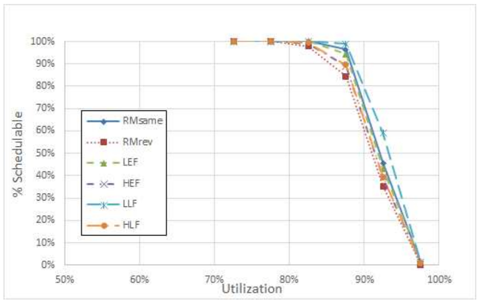


(b) 80 tasks

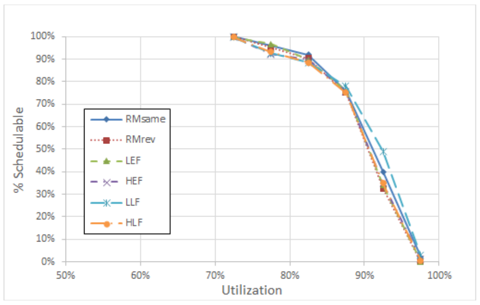


(c) 160 tasks

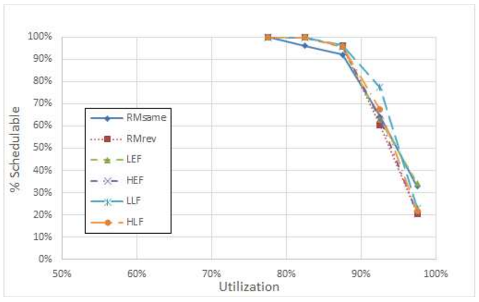
Figure 11: Simulations on 16 processors with various numbers of tasks for the “High Priority Order: RM, Deadline: Implicit” scenario.



(a) 64 tasks



(b) 160 tasks



(c) 320 tasks

Figure 12: Simulations on 32 processors with various numbers of tasks for the "High Priority Order: RM, Deadline: Implicit" scenario.