

Uniprocessor - Scheduler Simulator

Siddharth SAHAY
Jordi UGARTE

Prof. Joël GOOSSENS, Yannick MOLINGHEN

November 2024



Contents

1	Introduction	1
2	Methodology	2
2.1	Implementation Overview	2
2.2	Project Structure	3
2.3	Implementation of Algorithms & Determination of Feasibility	4
2.3.1	The Scheduler Class	4
2.3.2	Deadline Monotonic	4
2.3.3	Earliest-Deadline First	5
2.3.4	Round Robin	5
2.3.5	Additional Work	5
3	Experimental Setup	7
3.1	Usage	7
3.2	Return Codes	7
3.3	Dataset Structure	8
3.4	Evaluation Metrics	8
3.5	Parallel Processing	8
3.6	Dataset Anomalies	9
4	Experiment Results	10
4.1	Comparative Algorithm Performance	10
4.1.1	80% utilisation	10
4.1.2	10-tasks	12
4.2	Individual Algorithm Performance	13
4.2.1	Deadline Monotonic	13
4.2.2	Earliest-Deadline First	15
4.2.3	Round-Robin	17

5	Additional Implementations	18
5.1	Rate Monotonic	18
5.2	Audsley	18
5.3	Comparative Performance	18
5.3.1	10 Tasks	18
5.3.2	80% Utilisation	19
6	Summary	21

Abstract

This project involves the development of a simulator that runs the following scheduling algorithms: Deadline Monotonic, Earliest-Deadline First and Round-Robin. The simulator receives a command that specifies the type of algorithm and a specific task set in a CSV format containing the respective attributes of offset (O), computation time (C), deadline (D), and period (T). The simulator checks various metrics about the taskset and outputs whether the task set is schedulable, not schedulable (or additionally, if it took too much time to be executed). The return codes are described in later chapters.

There is also a provision to run multiple tasksets by passing an entire folder in the command. In this case, the simulator plots a pie chart that shows the amount of schedulable, not-schedulable and infeasible task sets. It then also compares the performance of the specified algorithm vs. an optimal one (EDF).

This report dives into the analysis we have performed regarding these scheduling algorithms, and how they perform against each other in two settings.

Scheduling algorithms, as we know, are strategies used to allocate resources to tasks to optimize certain objectives like efficiency, speed, or fairness. For this project, we look at these tasks through the lens of real-time operating systems, where these tasks are thought of as having a hard deadline, i.e. missing a deadline for a task could be catastrophic, if not leading to system failure or severe performance degradation. Real-world applications of such scheduling algorithms are countless - they are essential in areas like networking, manufacturing, and of course operating systems, and many more. The scheduling algorithms we study in this project are the following:

- Deadline Monotonic (DM)
- Earliest-Deadline First (EDF)
- Round-Robin (RR)

The idea of determining feasibility is simple: we need to certify real-time applications and machines so that we can know in advance that the systems will not fail in real-life situations. We must therefore study these algorithms, determine optimality and figure out applications ahead of time.

A very quick brief of the algorithms we will consider:-

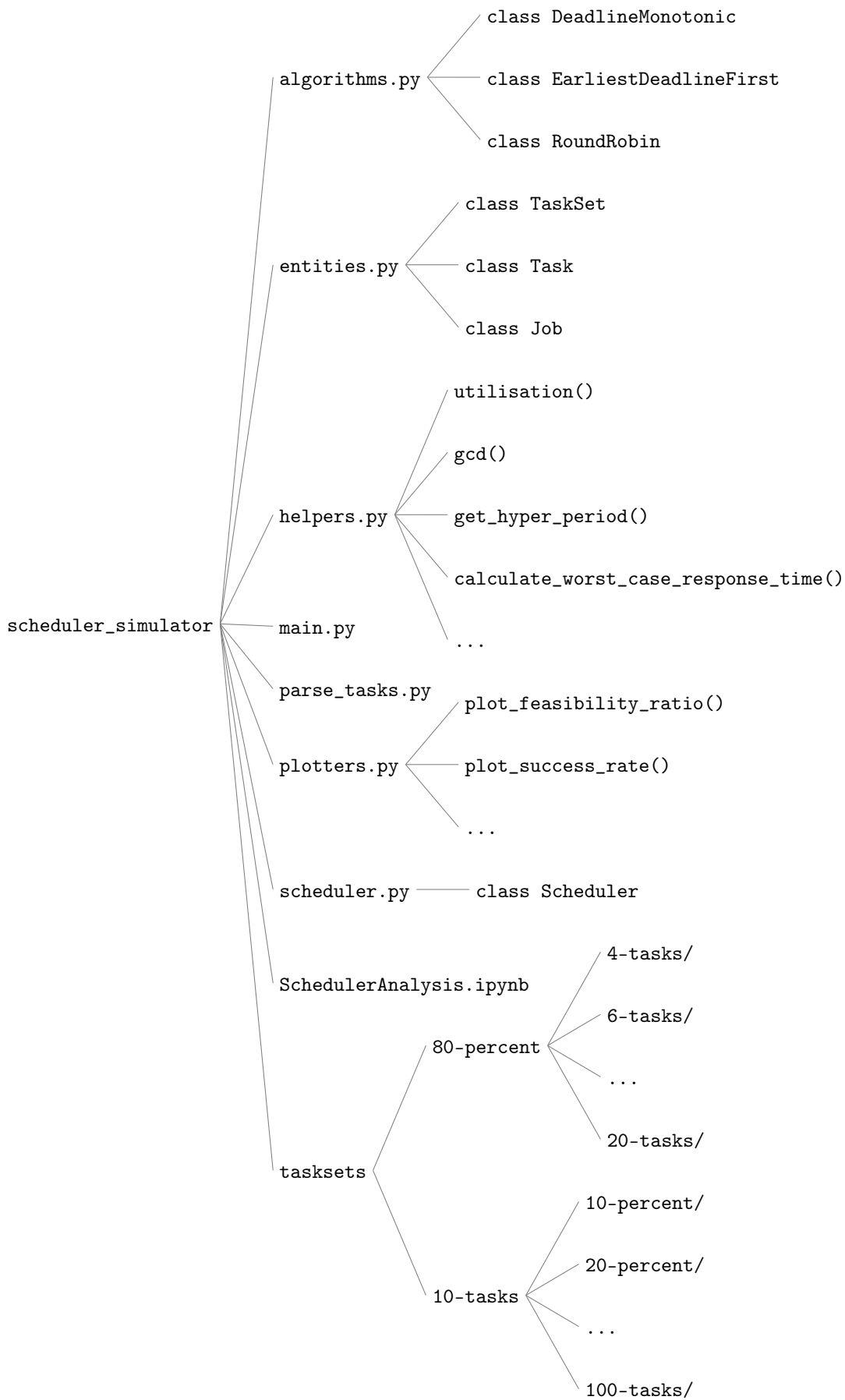
- **Deadline Monotonic (DM):** A fixed-priority algorithm where tasks are assigned priorities based on their deadlines—tasks with shorter deadlines receive higher priority. DM is widely used in real-time systems where tasks have hard deadlines, as it can guarantee schedulability under specific conditions, especially for periodic tasks with fixed constraints. But it is not an optimal algorithm, as we will see.
- **Earliest-Deadline First (EDF):** A dynamic-priority algorithm that prioritizes tasks with the nearest deadline. EDF is considered optimal for single-processor real-time scheduling as it maximizes the likelihood of all tasks meeting their deadlines, given feasible task sets (i.e. utilisation ≤ 1)
- **Round-Robin (RR):** A time-sharing, cyclic scheduling algorithm that allocates equal time slices to each task in a rotating order. Though it is not typically used for strict real-time constraints, RR can *sometimes* offer fairness and simplicity, making it useful in scenarios where responsiveness and workload distribution are key. In real-time operating systems though, it is usually never the right option, as we will establish in later chapters.

2.1 Implementation Overview

Python is our choice of programming language for this project. The code is structured in the following manner:-

- There are classes describing the task sets we will consider; the main TaskSet class contains a set of Tasks. Tasks are the model of a task containing attributes such as offset (O), computation Time (C), deadline (D) and period (T). Each task will in turn spin instances of Jobs, which are described by the remaining computation time and the release time.
- There is a main Scheduler class, which is the parent class, almost abstract in its implementation. It is meant to be extended by each algorithm to override default behaviour and functions, e.g., getting top priority, getting the simulation interval, etc. The basic simulation method which is the core algorithm within all schedulers is implemented in this class.
- Each algorithm class inherits from the Scheduler class and overrides basic functionality. These child classes help dictate various properties of each algorithm. How each algorithm determines feasibility is defined here in the child classes.
- There are also helper functions that help determine the feasibility of a task set and include some common functions to categorize it, such as: utilisation formulas, GCD (greatest-common divisor), LCM (least-common multiple), hyper-period, feasibility interval, busy period, delta time step, etc. There are also functions to perform operations like calculating the worst-case response time.
- We also have methods to plot the results we obtain from the simulator.
- The whole project can be visualised entirely by running a Python notebook called SchedulerAnalysis, which contains all the executions to run the folder tasksets. Once the notebook is fully run, it will plot the success rate of each folder. The folders of tasks sets are described later.

2.2 Project Structure



2.3 Implementation of Algorithms & Determination of Feasibility

2.3.1 The Scheduler Class

The `Scheduler` class serves as an abstract base class for implementing different scheduling algorithms, encapsulating common functionality and structure for the subclasses representing specific algorithms. The class is initialized with a `TaskSet`, which holds the tasks to be scheduled.

The `is_feasible()` method is implemented as an abstract method, meaning each subclass, or algorithm, must define how it evaluates the schedulability of the task set according to its unique algorithm.

Simulation Function The `simulate_taskset()` method provides a generalized structure for task set simulation, which any algorithm can potentially use to confirm feasibility by simulating job executions over time. The simulation loop evaluates deadlines, checks for job releases, and simulates scheduling active tasks. This design ensures that all subclasses have access to a consistent simulation process, with customizable logic for each algorithm's unique scheduling strategy. As we will see, most algorithms do not require this simulation, but this was a great starting point, for understanding each algorithm more granularly.

Feasibility Interval Calculation The `get_simulation_interval()` method, also abstract, enables each subclass to determine its feasibility interval based on either a fixed period (e.g., for DM) or a dynamic interval such as a busy period (e.g., for EDF and RR). The `is_task_set_too_long()` helper method checks if the interval length exceeds a specified limit, helping in excluding overly lengthy task sets early. This comes in handy for algorithms like RR, where we have to simulate till the end of the feasibility interval, which can be in billions since it's the LCM of the periods.

2.3.2 Deadline Monotonic

The Deadline Monotonic class begins by sorting the tasks in the task set according to their deadlines, using the period as a secondary sort key (for breaking ties, for example). This priority can be used throughout since DM is a fixed-priority scheduling algorithm which uses the same priority throughout the execution. We do not need this function since we use the Worst-Case Response Time (WCRT) to determine feasibility.

Feasibility Determination

- **Utilisation Check:** If the total utilisation of the task set is less than 1, DM can schedule the tasks, as utilisation acts as a preliminary feasibility shortcut. If the utilisation exceeds 1, the algorithm immediately returns infeasibility with a code of 3.
- **Worst-Case Response Time (WCRT) Analysis:** When utilisation alone is inconclusive, DM proceeds to perform a WCRT analysis. The WCRT is calculated for each task in the sorted task list, ensuring that each task's WCRT remains within its deadline. If the WCRT of any task exceeds its deadline, the task set is marked infeasible due to a potential deadline miss.

Our Approach Since DM is a fixed-priority algorithm, it can leverage WCRT analysis as an efficient alternative to task simulation. This approach minimizes the need for extensive runtime simulation, making it suitable for the static nature of DM scheduling.

2.3.3 Earliest-Deadline First

In EDF, priority is assigned dynamically based on each job's individual deadline, rather than task-specific parameters. The task with the earliest deadline receives the highest priority at any given scheduling decision point.

Feasibility Determination

- **Utilisation Check:** EDF is an optimal algorithm for task sets with utilisation less than 1. If utilisation exceeds 1, the task set is immediately deemed infeasible, with a return code of 3.
- **Simulation for Constrained Systems:** Due to the dynamic priorities in EDF, the feasibility check involves simulating task execution using the `simulate_taskset()` function. This simulation accounts for constrained deadlines by evaluating job completion within the simulation interval. The `get_top_priority()` function helps here to continually calculate the top priority task based on the active jobs that are in the system. This shows the dynamic nature of the algorithm - it does not depend on a precomputed priority!

Our Approach Simulation is essential for EDF due to its dynamic nature, where priority is determined by each job's deadline. Thus, the simulation verifies that jobs are completed within their deadlines, which cannot be assured by shortcuts alone, unlike DM. Also, since we are dealing with constrained deadline task sets, and not implicit deadline task sets, we cannot just use the utilisation to conclude feasibility. Had this been the latter case, feasibility determination would have been an even quicker affair, and no simulation would have been needed.

2.3.4 Round Robin

The RoundRobin class uses a time quantum derived from the average computation time of tasks. This quantum ensures fair rotation among tasks in the scheduling queue. We had to do a bit of research to chance upon this method of quantum calculation!

Feasibility Determination In Round Robin (RR) scheduling, feasibility is evaluated solely through simulation. Unlike DM and EDF, RR does not optimize for meeting deadlines or leverage utilisation-based shortcuts. As such, it relies entirely on simulation to assess feasibility, evaluating each task's ability to meet deadlines within the allotted quantum. If all deadlines are met within the defined interval, the task set is deemed schedulable; otherwise, it is considered infeasible.

Our Approach Due to the lack of deadline-awareness in RR, simulation is the most reliable approach for feasibility assessment. Simulation observes task behavior over time, checking for deadline misses, which RR's round-based approach alone cannot guarantee (as we will see!)

2.3.5 Additional Work

2.3.5.1 Rate Monotonic

RM algorithm assigns priorities based on task periods, where tasks with shorter periods have higher priorities. RM is a fixed-priority scheduling algorithm and determines feasibility by checking if the total utilization is less than or equal to 1, as per the Liu and Layland bound. If utilization is within this bound, the task set is considered schedulable;

otherwise, it proceeds to Worst-Case Response Time (WCRT) analysis. By analyzing each task's WCRT, RM verifies if response times remain within task deadlines, confirming feasibility without requiring runtime simulation. This static approach is both efficient and effective for tasks with predictable execution times and deadlines.

2.3.5.2 Audsley

Audsley algorithm uses an iterative approach to assign priorities by beginning with the lowest priority and checking if each task is schedulable at this priority level. If a task can meet its deadlines at the lowest priority level, it is assigned this priority, and the algorithm moves on to the next lowest priority for the remaining tasks. Audsley's method continues this way until all tasks have been assigned a unique priority level or until it finds an unschedulable task, marking the task set infeasible. This approach ensures optimal priority assignment and is particularly advantageous in systems where task deadlines may vary or priority reassignment might enhance schedulability. We will see later that in constrained deadline systems, Audsley can essentially equate to DM, since the best static priority that can be assigned is based on the deadline.

3.1 Usage

To run the code, use the following command:

```
python3 main.py [rm|dm|audsley|edf|rr] <task_set_location> -v -f
```

- **Algorithm:** Choose from `dm`, `edf`, `rr` (or even `rm` and `audsley`).
- **Task Set Location:**
 - If a **single task set file** is specified, the code evaluates just that file.
 - If a **directory** is specified, it evaluates all task sets within the folder (recursively) in parallel.
- **Options:**
 - `--verbose` or `-v`: Provides detailed output including the whole schedule, if a simulation is done, and a text output instead of just the return code.
 - `--force_simulation` or `-f`: Forces full simulation, even if utilization is below a set threshold. This was mainly used for debugging purposes and testing.

3.2 Return Codes

The program returns specific codes to indicate the outcome of each task set evaluation:

- **0:** The task set is schedulable, and the execution was simulated.
- **1:** The task set is schedulable, and a shortcut was used instead of full simulation.
- **2:** The task set is not schedulable, and the execution was simulated.
- **3:** The task set is not schedulable, and a shortcut was used instead of full simulation.
- **4:** Time-out, this was added on top of the existing return codes to account for the many task sets that had too long a feasibility interval to do a simulation.

These return codes provide insight into both the schedulability of each task set and whether the evaluation required a full simulation. We also introduced an additional return code of 4 to represent task sets that time out! These "undecided" task sets also form a small portion of the task set data we were provided, as we will see in later chapters.

3.3 Dataset Structure

The dataset is organized into two main types of folders: 80-percent and 10-tasks, each grouping task sets based on specific parameters.

- **80-percent:** This folder contains task sets with an average utilization of approximately 80%, with varying numbers of tasks. Here, our objective is to determine how the number of tasks affects the schedulability of each algorithm.
- **10-tasks:** This folder includes task sets where each set has exactly 10 tasks, but utilization varies. Our goal here is to assess how changing utilization impacts the schedulability and success rate of each scheduling algorithm.

We have used the first dataset that was provided, where the periods of the tasks was slightly longer, but since we implemented optimised methods of checking schedulability, this was not a huge concern.

In the next chapter, we will present the results of our analysis of these folders by calculating the ratio of feasible task sets and the success rate of each algorithm for each folder. This helps us understand the performance and limitations of the algorithms in different scenarios.

3.4 Evaluation Metrics

The key metrics used to assess algorithm performance are:

- **Schedulability:** Whether a task set meets its deadlines under each algorithm.
- **Success Rate:** Calculated as the ratio of schedulable task sets to total task sets for each algorithm.

If we run the command mentioned before on a particular folder, we plot statistics related to the different return codes, and also compare the performance of the algorithm with an optimal scheduling algorithm (which we consider as EDF).

3.5 Parallel Processing

We have used Python's multiprocessing module to implement parallel processing of task sets, in the case a folder containing multiple files is provided as input. The constant **NUMBER_OF_PARALLEL_PROCESSES** in *main.py* is set to 8, to use 8 cores but can easily be changed to suit a different processor and environment.

This is done to be able to efficiently process the entire folders of 80-percent/ and 10-tasks/ that each contains close to 5000 task sets.

3.6 Dataset Anomalies

An example of an anomalous task set can be seen in `tasksets/10-tasks/10-percent/taskset-421`. This task set, despite being categorized under the 10% utilization folder, was found to be infeasible. The set includes multiple tasks with identical parameters: three tasks have a computation time $C = 1$ and a deadline $D = 2$, which would of course create scheduling conflicts.

The specific tasks are represented as follows:

```
0,1,2,45 <---  
0,1,8,25  
0,1,3,26  
0,1,37,87  
0,1,2,45 <---  
0,1,37,43  
0,1,22,88  
0,1,34,76  
0,1,72,79  
0,1,2,57 <---
```

This particular task set was expected to be feasible due to its low overall utilization (10%), but the configuration—particularly the tasks with short deadlines relative to their periods—likely led to deadline misses, making it infeasible under certain scheduling algorithms. This demonstrates how specific task configurations, even with low utilization, resulted in unfeasible schedules even for the optimal scheduler. This was due to the random generation of the dataset, it's explained in further detail later on.

From our task set data, we can compare our three algorithms (DM, EDF, RR) side by side in two distinct scenarios. The first scenario involves fixing the utilisation at 80 percent and varying the number of tasks. This allows us to observe how each algorithm handles different levels of task concurrency under a high-utilisation environment. By examining performance as the number of tasks increases, we can evaluate each algorithm's ability to maintain schedulability as the system becomes more loaded, which is particularly useful for applications where the task count may fluctuate but the overall workload is consistently high. Here, we consider 80% to be high in this regard, since the Liu and Layland formula already tells us that as the number of tasks tends to infinity, the utilisation of fixed-priority task sets is around 69%, and 80% is well above that limit.

The second scenario involves fixing the number of tasks at 10 and varying the utilisation. This setup enables us to assess the algorithms' performance as the intensity of the workload changes. As utilisation increases, tasks consume a larger proportion of available processing time, which can push the system "closer to its limits". This scenario is valuable for understanding each algorithm's adaptability to increasing demands, particularly in systems where the number of tasks is stable but task execution requirements may vary, such as in applications with changing workloads.

Comparing the algorithms in these two scenarios provides insight into their strengths and weaknesses under different types of load conditions. It allows us to identify which algorithms are better suited for systems with fluctuating task counts versus those with fluctuating utilisation, helping inform algorithm choice for real-world applications with similar constraints. Though, in our case, as we will see, there is a clear winner!

4.1 Comparative Algorithm Performance

4.1.1 80% utilisation

4.1.1.1 Varying number of tasks

Let's first look at the ratio of tasks that are feasible to establish the baseline in Figure 4.1.

This graph shows the feasibility ratio of task sets for EDF scheduling at a fixed 80% utilisation while varying the number of tasks. Feasibility remains constant almost throughout around 50%. This provides us with a benchmark to now calculate the success rate of each of the other algorithms and discuss how well they performed.

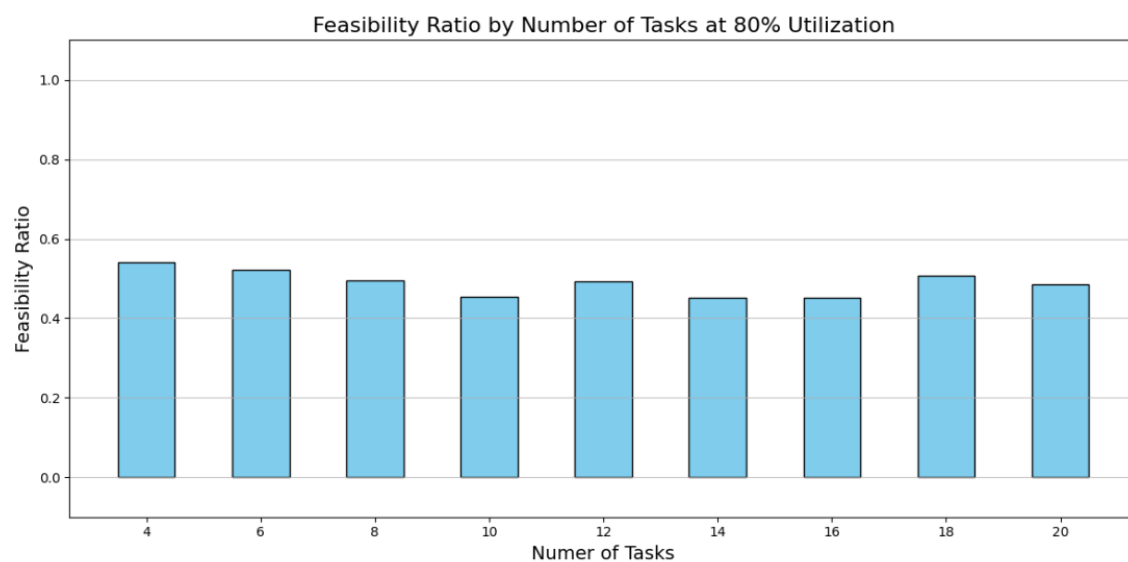


Figure 4.1: Feasibility Ratio - 80% Utilisation, Varying Number of Tasks

Info Note

This low percentage is possibly also a function of the fact that the dataset was randomly generated. Although intended for 80% utilisation, since deadlines are randomly generated we sometimes end up with "hard to schedule" task sets like the one that was described in the previous chapter, making them infeasible even with an optimal scheduler. EDF is of course an optimal scheduling algorithm, which means being able to successfully schedule any theoretically feasible task set that has a utilisation below 100% (for implicit deadline systems).

4.1.1.2 Success Rate of DM, EDF, RR

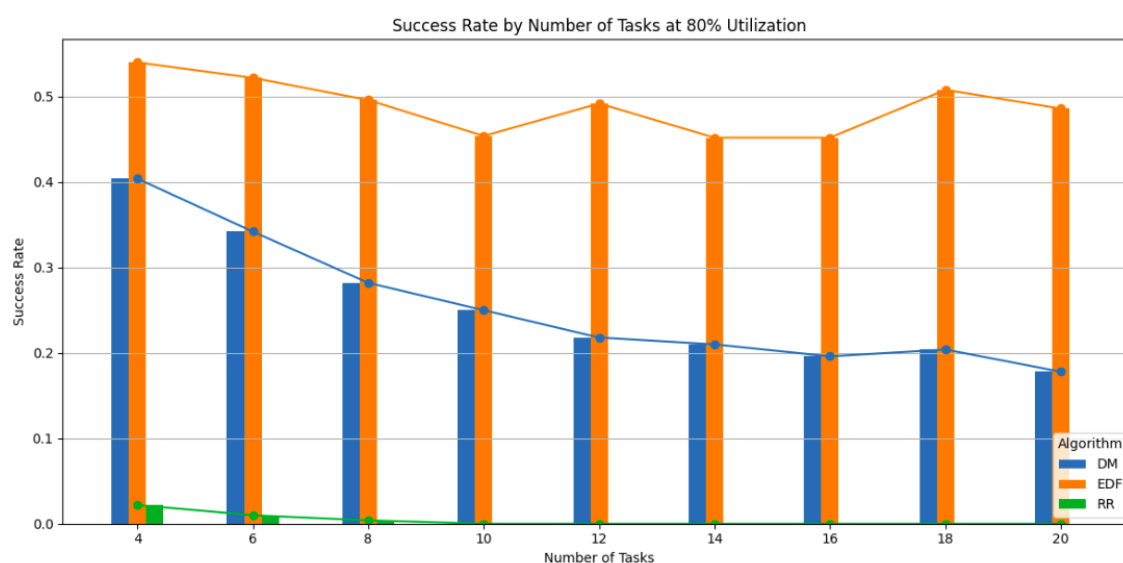


Figure 4.2: Success Rate - 80% Utilisation, Varying Number of Tasks

The plot demonstrates a distinct performance pattern for each algorithm. EDF maintains a relatively high success rate across all task counts, consistently around 0.5. This stability highlights EDF's strength in handling higher

utilisation levels, as it dynamically prioritizes tasks with the earliest deadlines, adapting well to changes in task count without a significant drop in feasibility.

DM, on the other hand, shows a decreasing trend in success rate as the number of tasks increases. This is because DM assigns fixed priorities based on task deadlines, which can become less effective as the task count grows. With more tasks, the likelihood of deadline conflicts increases, causing DM's performance to degrade. This reflects DM's limitation in handling high utilisation with larger task sets compared to EDF.

RR performs the worst overall, with success rates close to zero across most task counts. This is expected, as Round Robin is not a priority-based algorithm and does not account for task deadlines in its scheduling decisions. Instead, RR simply cycles through tasks in a time-slice fashion, which is generally ineffective for meeting strict real-time deadlines, especially at higher utilisation levels.

4.1.2 10-tasks

4.1.2.1 Varying utilisation

The feasibility ratio for EDF with varying utilisation and a fixed number of tasks can be seen in Figure 4.3. This plot provides a benchmark for comparing DM and RR against the optimal scheduler for our dataset.

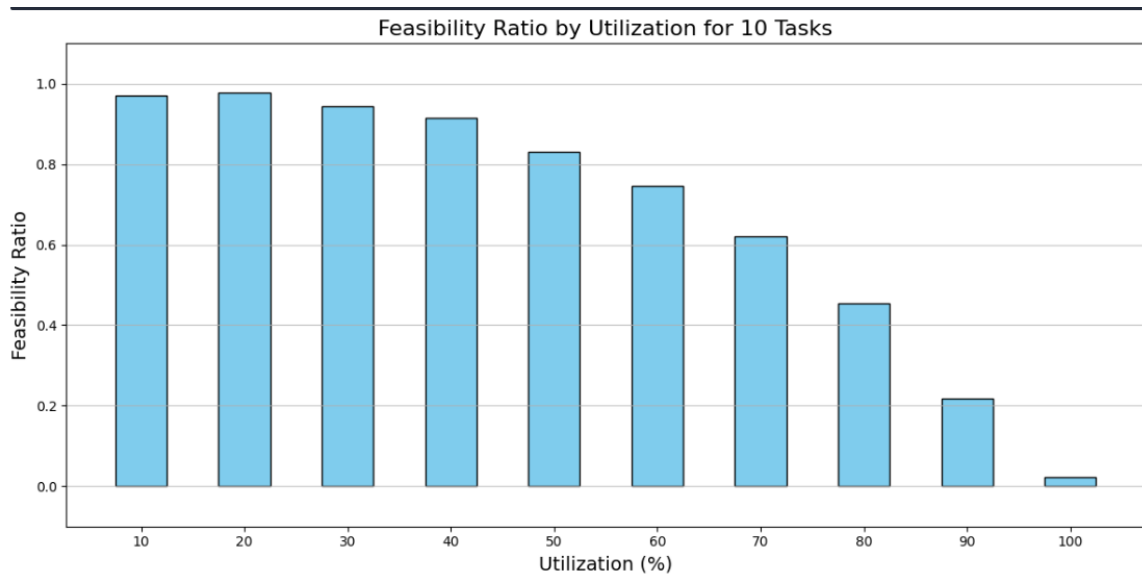


Figure 4.3: Feasibility Ratio - 10-tasks, Varying Utilisation

Figure 4.3 shows the feasibility ratio for EDF scheduling as utilisation increases, for a fixed set of 10 tasks. At lower utilisation levels (10% to 40%), the feasibility ratio remains close to 1.0, indicating that EDF can reliably schedule nearly all task sets in this range. This is expected!

As utilisation approaches 50% and beyond, the feasibility ratio begins to decline steadily. This drop-off reflects the fact that, as more processing power is demanded, the likelihood of tasks meeting their deadlines decreases. By the time utilisation reaches 80% and above, the feasibility ratio falls sharply, with near-zero feasibility at 100% utilisation.

4.1.2.2 Success Rate of DM, EDF, RR

The success rate of the algorithms in question with a fixed number of 10 tasks execution and a varying utilisation is shown on Figure 4.4.

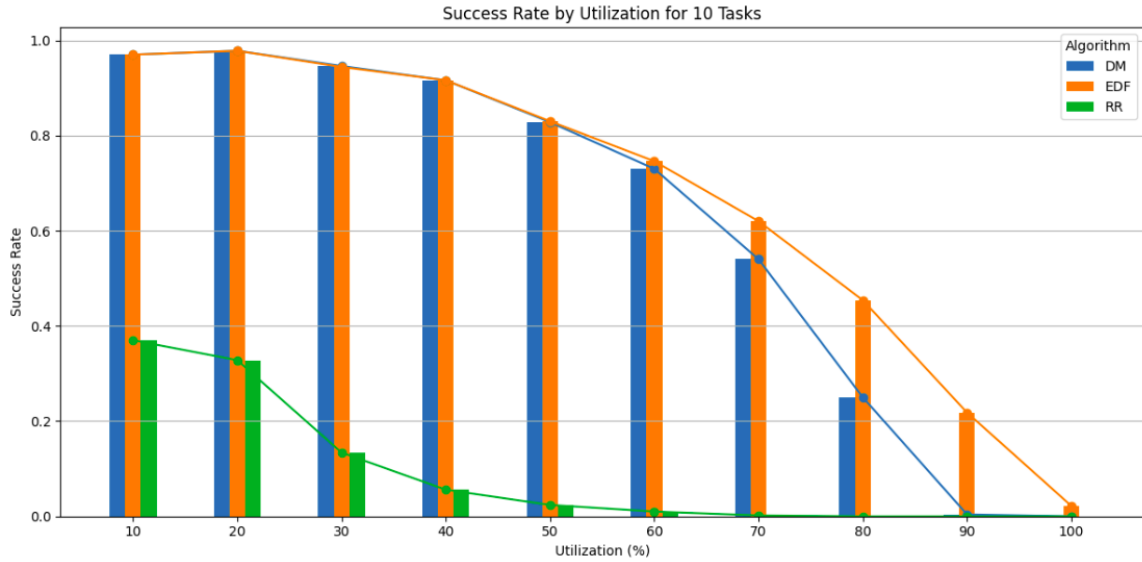


Figure 4.4: Success Rate - 10-tasks, Varying Utilisation

DM performs similarly to EDF at lower utilisation levels, where both algorithms achieve high success rates. However, DM's success rate begins to decline sooner than EDF's, around 60% utilisation, and falls off more sharply as utilisation increases. This difference is due to DM's fixed-priority nature, which can be less flexible than EDF's dynamic deadline-driven approach as task demands intensify. DM becomes less effective at higher loads because fixed priorities do not adapt to varying task deadlines, causing more deadline misses as utilisation rises.

RR performs significantly worse than both DM and EDF, with a low success rate across all utilisation levels. Even at low utilisation (10%-20%), RR struggles to meet deadlines due to its lack of deadline-awareness, achieving a success rate below 0.4 and decreasing rapidly as utilisation increases. By 40% utilisation, RR's success rate is nearly zero. This outcome reflects the limitations of Round Robin for real-time systems, where its equal time-slice approach is ineffective in meeting the strict timing constraints required by most real-time tasks, yet again!

4.2 Individual Algorithm Performance

Lastly, we will analyze the performance of each algorithm individually across the dataset, focusing on the 80% utilisation and 10-tasks categories.

In the graphs below we also see a portion attributed to "Timed Out". This is added specifically to be able to compare all the algorithms at a 1:1 ratio, instead of comparing different algorithms where a different number of tasks had a decided outcome (vs. tasks that were "undecided").

4.2.1 Deadline Monotonic

4.2.1.1 10 Tasks

- **Feasible Task Sets:** 61.6% of task sets were feasible and were identified using shortcuts, with none requiring full simulation.
- **Not Schedulable Task Sets:** 38.4% of task sets were determined to be unschedulable, all confirmed through shortcuts without the need for simulation.

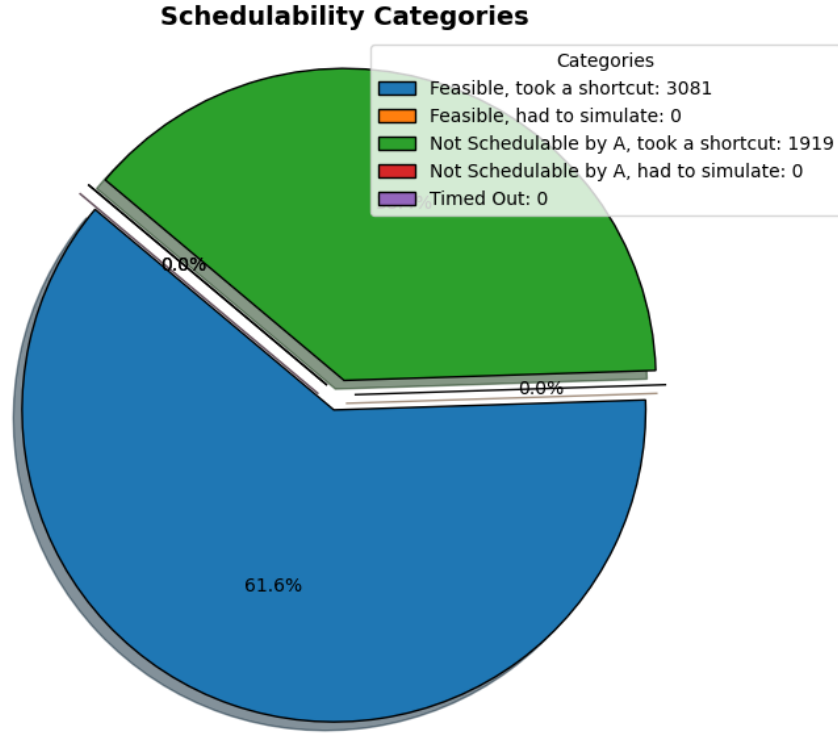


Figure 4.5: Feasibility of DM in 10-tasks/

With 10 tasks, DM effectively determined the schedulability of a majority of task sets (61.6%) using shortcuts alone, without the need for full simulation. The algorithm also efficiently identified un-schedulable sets through shortcuts (the use of WCRT), and no cases timed out. This high rate of shortcut-based schedulability assessment suggests that DM performed well in this scenario, with increased efficiency in feasibility determination due to the lack of simulated cases.

Use of WCRT for DM

The Worst-Case Response Time (WCRT) w_k for a task τ_i in Deadline Monotonic (DM) scheduling is calculated iteratively with the formula below:

$$\begin{cases} w_0 = C_i & \text{(initialisation)} \\ w_{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_k}{T_j} \right\rceil C_j & \text{(iteration)} \end{cases}$$

WCRT analysis is effective for determining feasibility because it calculates the maximum time a task could take from release to completion, accounting for potential delays from higher-priority tasks, which is why we've used it for our feasibility determination.

4.2.1.2 80% Utilisation

- **Feasible Task Sets:** 25.4% of task sets were feasible, all of which were identified using shortcuts, with none requiring full simulation.
- **Not Schedulable Task Sets:** 74.6% of task sets were determined to be unschedulable, all confirmed through shortcuts without the need for simulation.

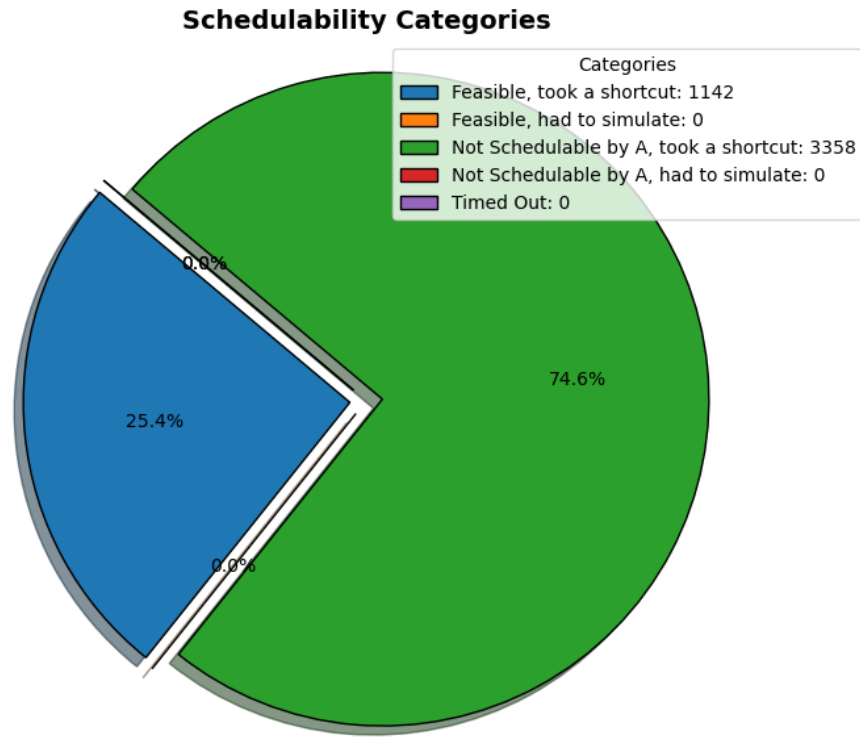


Figure 4.6: Feasibility of DM in 80-percent/

At 80% utilisation, DM was able to quickly assess schedulability for all task sets using shortcuts alone, without requiring simulation. A significant portion (74.6%) of task sets were deemed unschedulable, reflecting the challenges of maintaining feasibility as utilisation approaches higher percentages. This high rate of shortcut-based assessment highlights DM's efficiency at this utilisation level, even though the feasibility decreases as system load increases.

4.2.2 Earliest-Deadline First

4.2.2.1 10 Tasks

- **Feasible Task Sets:** 67.0% of task sets were feasible, all of which required full simulation, with no shortcuts applicable.
- **Not Schedulable Task Sets:** 33.0% of task sets were determined to be unschedulable, with only 0.1% confirmed through shortcuts and 32.9% requiring simulation to verify infeasibility.

For 10 tasks, EDF required full simulation for the majority of cases to determine schedulability, with no feasible sets identified through shortcuts. While EDF successfully scheduled 67.0% of task sets, it also had a significant portion (33.0%) of unschedulable cases, most of which required simulation to confirm infeasibility. This highlights EDF's reliance on simulation in scenarios where task deadlines and load complicate shortcut-based feasibility checks.

4.2.2.2 80% Utilisation

- **Feasible Task Sets:** 48.9% of task sets were feasible, all of which required full simulation, with no shortcuts applicable.
- **Not Schedulable Task Sets:** 51.1% of task sets were determined to be unschedulable, with all cases requiring simulation to verify infeasibility.

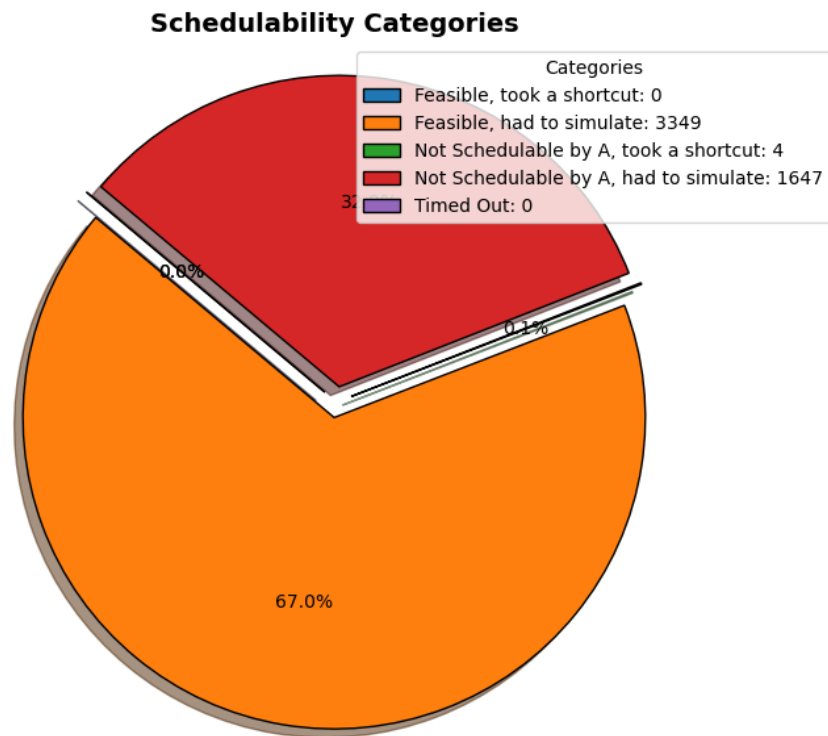


Figure 4.7: Feasibility of EDF in 10-tasks/

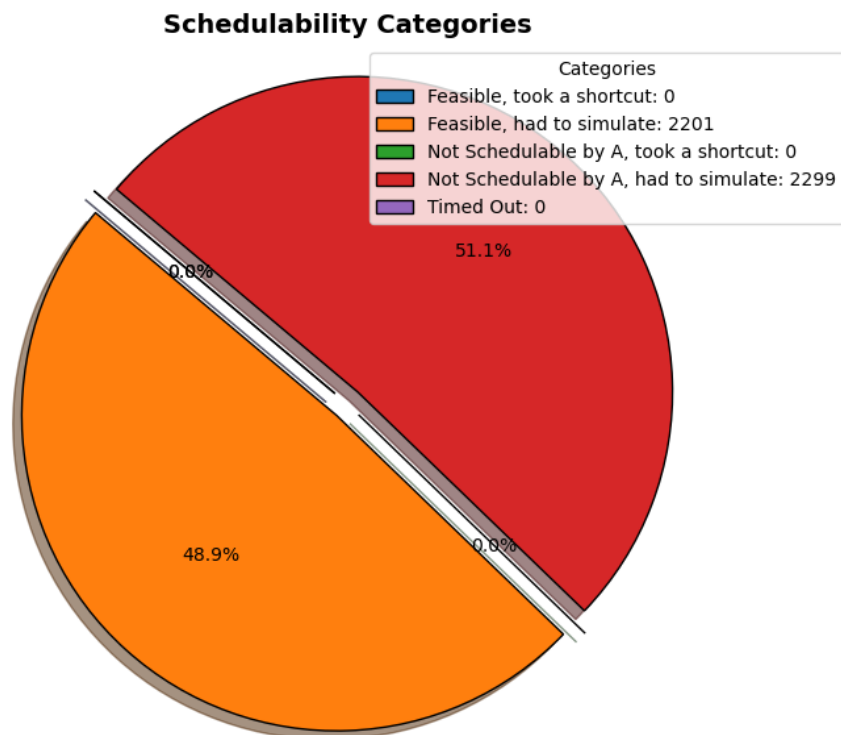


Figure 4.8: Feasibility of EDF in 80-percent/

We see here again, EDF's reliance on full simulation to confirm schedulability.

4.2.3 Round-Robin

Round Robin was largely infeasible compared to the other algorithms, struggling with almost all task sets. This clearly demonstrates that RR cannot handle deadline constraints effectively, as it processes tasks in a cyclic, time-sliced manner without considering task priorities or deadlines. With well over 90% of task sets deemed infeasible and no feasible cases identified through shortcuts, RR's equal time-slicing approach fails to meet real-time deadlines, making it highly unsuitable for real-time applications, especially at higher utilisation levels.

4.2.3.1 10 Tasks and 80% Utilisation

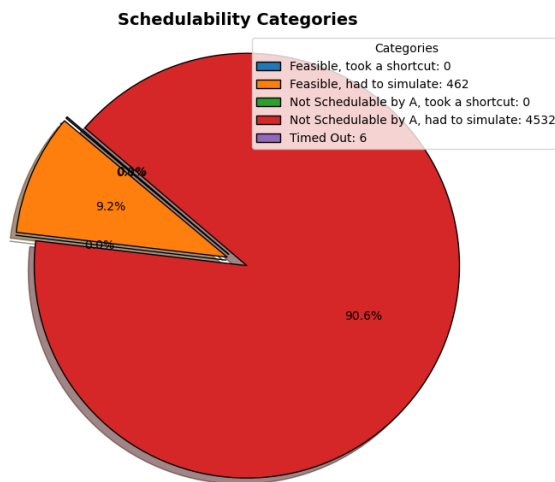


Figure 4.9: Feasibility of RR in 10-tasks

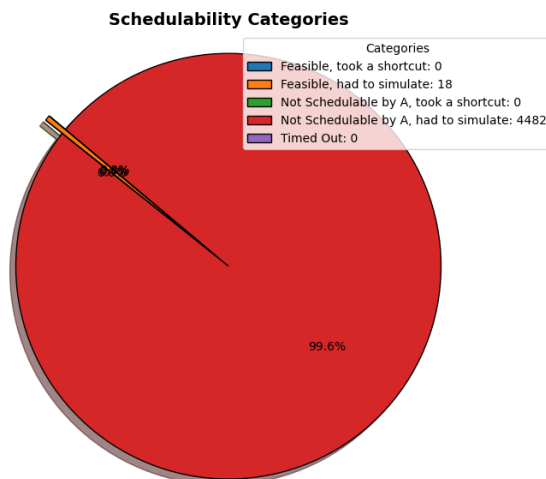


Figure 4.10: Feasibility of RR at 80% Utilisation

Additional Implementations

Other scheduling algorithms, such as Rate Monotonic and Audsley, were implemented as well to experiment with different algorithmic dynamics. The *main.py* accepts as argument `rr` and `audsley` in addition to the others.

5.1 Rate Monotonic

Rate Monotonic (RM) is also a fixed-priority scheduling algorithm where tasks are assigned priorities based on their periods—tasks with shorter periods receive higher priority. RM is commonly used in real-time systems for periodic tasks, as it offers guaranteed schedulability under certain utilization bounds (the Liu and Layland bound!) However, RM is not optimal for all real-time scenarios, as it may fail to meet deadlines in cases where task periods are non-harmonic or utilization exceeds specific thresholds.

5.2 Audsley

Audsley is an optimal priority assignment algorithm for fixed-priority scheduling. Audsley's Algorithm systematically assigns priorities to tasks in a way that maximizes schedulability, ensuring that each task's deadline can be met if there exists a feasible fixed-priority assignment. Although it provides an optimal priority order for schedulable tasks, it is still limited by the fixed-priority approach and cannot adapt dynamically to changing task demands as a dynamic scheduler like EDF would.

5.3 Comparative Performance

Adding RM and Audsley into the mix here is very interesting for the comparisons with the algorithms we've already seen!

5.3.1 10 Tasks

RM exhibits a moderate level of success at lower utilization levels, maintaining around a 40-50% success rate. However, this success rate begins to decline steadily after 40% utilization and drops to near zero beyond 70% utilization. This behavior, in a way, aligns with the Liu and Layland bound, which limits the schedulability of task sets as utilization increases in fixed-priority algorithms like RM. As a static priority assignment algorithm, RM can struggle to meet

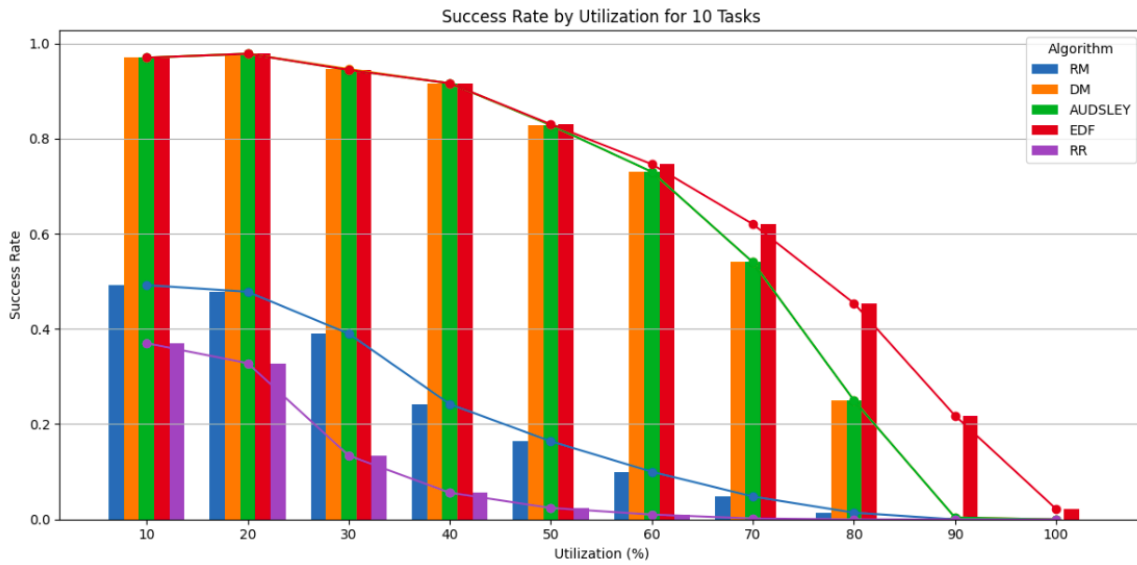


Figure 5.1: Success Rate of RM, Audsley in 10-tasks/

deadlines when task periods do not align harmonically, especially under higher utilization conditions, leading to more frequent deadline misses. It performs worse than DM in this regard (and of course, EDF).

Audsley's algorithm performs very similarly to DM, in fact so similarly that you cannot see the line for DM since it's hidden behind Audsley in Figure 5.1 and Figure 5.2, despite theoretically being a more flexible priority assignment algorithm! This highlights a very interesting insight: in constrained-deadline systems, Audsley's flexibility in priority assignment "converges" to the same priority configuration as DM, indicating that deadline-based priority assignment is optimal in this context.

Conversely, this also means that once priorities are set for Audsley, they remain static, and it doesn't dynamically adjust priorities like EDF. As utilization increases, Audsley's optimal priority assignment struggles with the load, just like DM!

5.3.2 80% Utilisation

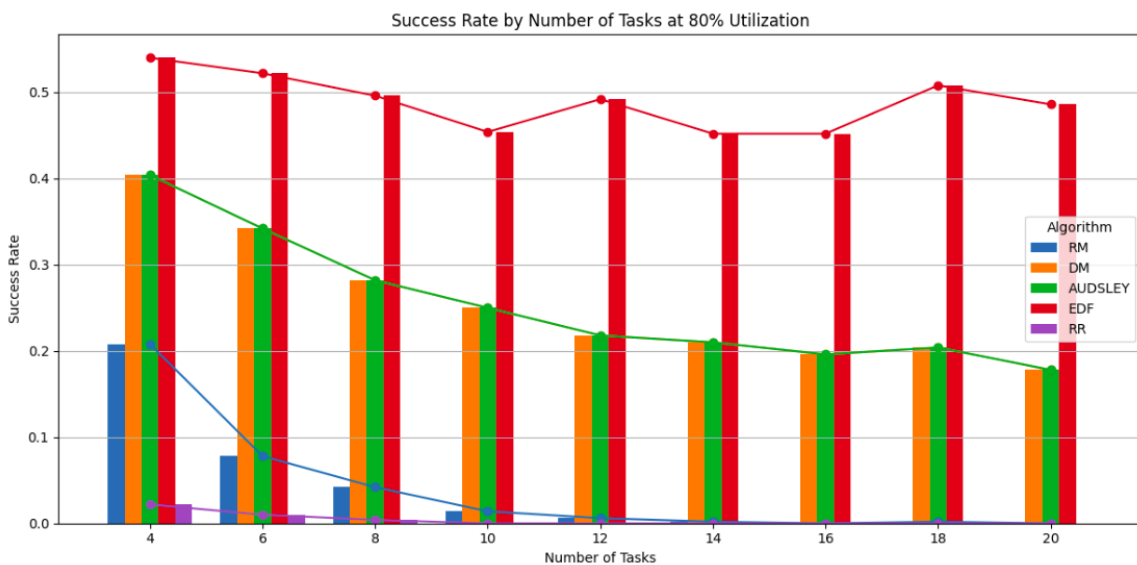


Figure 5.2: Success Rate of RM, Audsley in 80-percent/

The findings in Figure 5.2 are in line with what we observed in the previous section with a constant set of tasks and varying utilisation.

At this high utilization level, EDF stands out with the highest success rate, as we have seen, maintaining above 50% feasibility across all task counts.

Audsley and DM again perform almost identically, with success rates that decrease gradually as the task count rises. Both algorithms hover around a 20-30% success rate, indicating their limitations under high utilization and higher task counts due to static priorities.

RM does worse than both DM and Audsley, with success rates that start around 20% for lower task counts and drop off steadily as tasks increase. This is consistent with RM's reliance on fixed-priority assignments based on task periods, which is an even worse candidate than deadlines for priority setting!

This project explores and evaluates the performance of five scheduling algorithms—Deadline Monotonic (DM), Earliest-Deadline First (EDF), Round Robin (RR), Rate Monotonic (RM), and Audsley—through a scheduler simulator. Each algorithm represents a unique approach to task prioritization and scheduling, aiming to maintain task schedulability under different conditions.

Earliest-Deadline First (EDF) emerges as the clear winner and most optimal algorithm for feasible task sets, consistently meeting more deadlines across scenarios by dynamically prioritizing tasks with the nearest deadlines. Its dynamic priority assignment matches up to the task of changing needs that even a large task-set has over the course of its execution.

Deadline Monotonic (DM), a fixed-priority algorithm, performs effectively at lower task counts but struggles as task loads increase. DM assigns priorities based on deadlines, giving higher priority to tasks with shorter deadlines, but its static nature makes it less adaptable under high-utilization conditions, leading to increased deadline misses as the number of tasks or utilisation goes up. Where the number of tasks go up - deadline creep is an issue, and where utilisation goes up - a fixed priority scheduler may even cause priority inversion!

Round Robin (RR), a cyclic scheduling approach that allocates equal time slices to each task, proves highly unsuitable for real-time systems with hard deadlines. With no consideration for task deadlines, RR shows extreme infeasibility in maintaining deadlines across the board, regardless of the number of tasks and utilization.

Rate Monotonic (RM) prioritizes tasks based on their periods, with shorter periods receiving higher priority. While RM performs well under specific utilization limits, it lacks the flexibility of EDF and struggles with non-harmonic task periods, especially as utilization exceeds its theoretical bound. This results in a steep decline in success rate at higher task loads. It performs worse than DM in this context. Having the priority a function of the period and not the deadline is likely the reason for this, since in constrained deadline systems, the deadline is strictly shorter than the period and hence makes for a better priority metric.

Audsley's Algorithm, known for optimal priority assignment in fixed-priority scheduling, theoretically offers the most efficient task priority configuration. However, its fixed-priority nature limits its effectiveness at high utilizations, where it faces challenges similar to DM, despite optimal prioritization. Another interesting observation in this context is that DM is indeed the most optimal priority assignment for constrained deadline systems!

The simulator reveals EDF's robust performance, DM's moderate success with lower task counts/utilisations, and the utter shortcomings of RR for applications in real-time systems. RM and Audsley demonstrate interesting contrasts to the others but also highlight the limitations of fixed priority algorithms under fluctuating task demands!

