SHARIF UNIVERSITY OF TECHNOLOGY

COMPUTER ENGINEERING

REAL TIME SYSTEMS

Final Project

Authors:
Ahmadreza Hamzei (98101447),
Ali Ahmadi Kafshani (97105703)

Instructor:
Dr. Safari

February 2, 2024

Contents

List of Figures

1 Abstract

This report introduces a novel approach to task management in critical-mixed multi-core systems, focusing on the efficient handling of periodic Low Criticality (LC) and High Criticality (HC) tasks. The system operates under two core mapping policies: Worst Fit Decreasing (WFD) and First Fit Decreasing (FFD), prioritizing the mapping of both original tasks and their copies. In its normal state, the system is adept at handling core overruns by transferring LC tasks to other cores, thus introducing a new operational state termed 'HOST'. This adaptive mechanism ensures the continuous and effective execution of tasks, with a particular emphasis on HC tasks whose timely and accurate execution is critical. Task scheduling across cores is governed by the VD-EDF (Variable Deadline Earliest Deadline First) algorithm, balancing the criticality of HC tasks with the need to maintain a minimum quality of service system-wide. The intricacies of this system are explored in this report, underlining its potential in enhancing task management in complex multi-core environments. The codes and results for phase 1 and phase 2 of this project are available in the GitHub repository: `https://github.com/aliraad79/RealTimeProject`.

2 Introduction

In the realm of real-time systems, the management and allocation of tasks in multi-core environments pose significant challenges, especially when dealing with a mix of tasks varying in criticality. This report presents an in-depth analysis of a critical-mixed multi-core system, specifically designed to handle peri-vedic LC and HC tasks. The uniqueness of this system lies in its dual worst execution time framework for HC tasks and a singular worst execution time consideration for LC tasks.

The core of the system's functionality revolves around two mapping policies: WFD and FFD, which prioritize the allocation of both original tasks and their duplicates. The system's robust design allows it to operate initially in a normal state, seamlessly transitioning to a HOST state in response to core overruns. This transition is critical for reallocating LC tasks to cores with spare capacity, ensuring uninterrupted system performance.

A key feature of this system is the implementation of the VD-EDF scheduling algorithm, which is pivotal in managing the execution of tasks on various cores. This algorithm plays a crucial role in upholding the system's integrity, particularly in ensuring the precise and timely execution of HC tasks, which are of utmost importance. Simultaneously, the system is designed to guarantee a minimum quality of service, acknowledging the essential nature of LC tasks.

This report aims to provide a comprehensive overview of the system's architecture, its operational mechanisms, and the benefits it offers in the context of real-time, multi-core task management. Through this analysis, we aim to demonstrate the system's efficacy in enhancing task execution and management in complex, critical-mixed multi-core environments.

3 UUnifast

The provided code snippet demonstrates the implementation of the UUniFast algorithm, a method used for generating synthetic task sets in real-time systems research. The UUniFast algorithm is particularly designed to distribute a specified total utilization across a set of tasks while maintaining a uniform distribution of utilization. This approach is crucial for simulating realistic scenarios in real-time systems where tasks have varying computational demands.

3.1 Key Components of the UUniFast Implementation:

1. **Utilization Distribution (generate_uunifast):** The core function `generate_uunifast` takes the total utilization (`u`), the number of task sets (`nsets`), and the number of tasks in each set (`num_tasks`). It generates multiple sets of task utilizations that sum up to the specified total utilization. The algorithm ensures that the utilization is split uniformly among the tasks in each set.

2. **Random Period Generation (generate_random_periods_uniform):** This function generates random periods for tasks using a uniform distribution. It allows specifying the number of periods (`num_periods`), the number of sets (`num_sets`), and the range (`min_period`, `max_period`). Optionally, periods can be rounded to integers.

3. **Task Set Creation (generate_tasksets):** This function combines the utilizations and periods to create task sets. Each task is represented as a tuple of computation time (`c`), period (`p`), and utilization (`u`). The computation times are truncated to avoid floating-point precision errors.

4. **Final Task Set Assembly (generate_task_set):** This function integrates all the components. It generates periods, utilizes the UUniFast algorithm to distribute utilizations, and then combines them to form task sets. Each task is assigned a priority, distinguishing between high and low criticality tasks based on a specified ratio (`hc_to_lc_ratio`).

3.2 Significance in Real-Time Systems:

The UUniFast algorithm's ability to uniformly distribute utilization across tasks makes it an invaluable tool in the study and simulation of real-time systems. By generating realistic task sets with varied computational demands, researchers and engineers can better understand and design systems that efficiently manage and schedule tasks according to their criticality and resource requirements. This code implementation offers a practical approach to generating such task sets, providing a foundation for further analysis and optimization in real-time system design.

4 TMR

The provided Python script, `tmr.py`, outlines an implementation of the Triple Modular Redundancy (TMR) technique in real-time systems. TMR is a fault-tolerance method that involves replicating tasks to ensure system reliability. The script defines a `TMRManager` class responsible for applying TMR to a set of tasks and performing majority voting to determine the final result.

4.1 Key Components of the TMR Implementation:

1. **Applying TMR to Task Sets (apply_tmr_to_taskset):** This method takes a set of tasks and applies TMR by creating three copies of each task. Each copy is assigned the same TMR group identifier, ensuring they are recognized as replicas of the same task. This process enhances fault tolerance by allowing the system to withstand single-point failures.
2. **Performing Majority Voting (perform_majority_voting):** After task execution, this method is used to perform majority voting on the results. It takes the results of the tasks, grouped by their TMR identifiers, and determines the final result for each group based on the majority vote. This step is crucial in mitigating the impact of erroneous task executions.

4.2 Example Usage:

The TMRManager class can be utilized as follows:

```
tmr_manager = TMRManager()
tmr_tasks = tmr_manager.apply_tmr_to_taskset(original_task_set)
... execute tasks ...
final_results = tmr_manager.perform_majority_voting(task_results)
```

4.3 Significance in Real-Time Systems:

The implementation of TMR in real-time systems is vital for ensuring reliability and fault tolerance, especially in critical applications where failure can have severe consequences. By replicating tasks and using majority voting to determine outcomes, the system can continue to operate correctly even in the presence of faults in individual tasks. This Python implementation provides a practical and straightforward approach to integrating TMR in real-time system applications.

5 Sample Results

We can generate a task set for any utilization that we want.

python main.py

5.1 Result

task id	Utilization	Priority
5b989	0.136655	HIGH
0027b	0.136655	HIGH
d4683	0.136655	HIGH
dfd1b	0.041013	LOW
ac3a5	0.041013	LOW
d6b3d	0.041013	LOW
82d6c	0.277706	HIGH
8ed5d	0.277706	HIGH
2a3c3	0.277706	HIGH
1413f	0.408287	LOW
123bf	0.408287	LOW
231d0	0.408287	LOW
9cf1f	0.161456	HIGH
99e67	0.161456	HIGH
fb4d5	0.161456	HIGH
8794e	0.080161	LOW
4f436	0.080161	LOW
0cbbb	0.080161	LOW
8eb3c	0.164999	HIGH
c3fb9	0.164999	HIGH
0bd6c	0.164999	HIGH
482cf	0.120716	LOW
51d56	0.120716	LOW
2a764	0.120716	LOW
2633a	0.105024	HIGH
c945f	0.105024	HIGH
57afb	0.105024	HIGH
3a384	0.023847	LOW
99c6a	0.023847	LOW
6e7c6	0.023847	LOW
e83db	0.034302	HIGH
fadbd	0.034302	HIGH
23913	0.034302	HIGH
e929e	0.623831	LOW
df1cf	0.623831	LOW
f83bb	0.623831	LOW
c67ba	0.14211	HIGH
3a814	0.14211	HIGH
f76a6	0.14211	HIGH
2f61b	0.170024	LOW
32b83	0.170024	LOW
b0a8f	0.170024	LOW
739ae	0.018761	HIGH
08328	0.018761	HIGH
2a4eb	0.018761	HIGH
1956a	0.111256	LOW
35dbf	0.111256	LOW
222e1	0.111256	LOW
1db2f	0.971521	HIGH
4a2e9	0.971521	HIGH
faa1b	0.971521	HIGH
b3cb1	0.275892	LOW
99606	0.275892	LOW
06712	0.275892	LOW
16513	0.025777	HIGH
6b441	0.025777	HIGH
c85d3	0.025777	HIGH
60fd4	0.106651	LOW
9cbeb	0.106651	LOW
05555	0.106651	LOW

6 FFD

6.1 Introduction

This algorithm heads to distribute the tasks between processors core. In each step this algorithm find the first processor that task can be placed in and assign task to it.

6.2 Result

For the given test task set and assumptions we assign task like this using WFD.

processor number	Tasks
0	5b989, b3cb1
1	0027b, 99606
2	d4683, 06712
3	dfd1b, 1413f, 739ae
4	ac3a5, 123bf, 08328
5	d6b3d, 231d0, 2a4eb
6	82d6c, 8ed5d, 8794e
7	2a3c3, 9cf1f, 1db2f
8	fb4d5, 4f436, 0cbbb, 8eb3c, c3fb9, 1956a

6.3 Advance mode

In this mode we think for the low priority task the processors can fill up to 1.5 of their utilization.

processor number	Tasks
0	5b989, 82d6c, e929e
1	0027b, 8ed5d, df1cf
2	d4683, 2a3c3, f83bb
3	dfd1b, ac3a5, 8794e, 739ae
4	d6b3d, 1413f, 9cf1f, 99e67, 1956a
5	123bf, 231d0, fb4d5, 4f436, 0cbbb, e83db, fadbd, 08328
6	8eb3c, c3fb9, 0bd6c, 482cf, 51d56, 2a764, 23913 c67ba, 3a814, f76a6, 35dbf, 16513
7	2633a, c945f, 57afb, 3a384, 99c6a, 6e7c6, 2f61b, 2a4eb, 222e1, 6b441, c85d3
8	32b83, b0a8f, 1db2f, 4a2e9, faa1b, b3cb1, 99606, 06712, 60fd4, 9cbeb, 05555

7 WFD

7.1 Introduction

This algorithm heads to distribute the tasks between processors core. In each step this algorithm find the processor which has the lowest work load and assign the task to it.

7.2 Result

For the given test task set and assumptions we assign task like this using WFD.

processor number	Tasks
0	5b989 , b3cb1
1	0027b , 99606
2	d4683 , 06712
3	dfd1b , 9cf1f, 739ae, 60fd4
4	ac3a5 , 99e67, 08328, 9cbeb
5	d6b3d , fb4d5, 2a4eb, 05555
6	82d6c , 1413f, 8794e, 2633a
7	8ed5d , 123bf, 4f436, c945f
8	2a3c3 , 231d0, 0cbbb, 57afb

7.3 Advance mode

In this mode we think for the low priority task the processors can fill up to 1.5 of their utilization.

processor number	Tasks
0	5b989, e929e , 1956a
1	0027b, df1cf , 35dbf
2	d4683, f83bb , 222e1
3	dfd1b, 9cf1f, e83db, 739ae , 16513
4	ac3a5, 99e67, fadbd, 08328 , 6b441
5	d6b3d, fb4d5, 23913, 2a4eb , c85d3
6	82d6c, 1413f, 8794e, 2633a , c67ba
7	8ed5d, 123bf, 4f436, c945f , 3a814
8	2a3c3, 231d0, 0cbbb, 57afb , f76a6

8 Overrun and Migration

8.1 Introduction to Overrun

Overrun in real-time systems occurs when a task, typically of low criticality and initially estimated to have a short worst-case execution time, fails to complete its execution within the expected timeframe. This issue particularly manifests in online scheduling environments, where tasks are dynamically managed and executed. Unlike in offline scheduling, where the precise timing of an overrun is not predictable due to the lack of real-time execution data, in online scheduling, an overrun can be more readily identified. For analytical or simulation purposes, when dealing with offline scheduling, it's often necessary to select a random point in time to represent when an overrun might occur. Additionally, in the context of our project, the number of cores that suffer from overrun is specified in the project description, highlighting the extent and impact of overrun in multi-core processing environments.

8.2 Implementation Details

In the implementation of overrun handling in our project, the processor is designed to operate in two distinct modes: NORMAL and OVERRUN. To accommodate this, tasks are assigned two types of execution times: a high execution time and a low execution time.

- **Processor Modes**:
  - **NORMAL Mode**: In this mode, tasks execute under regular conditions.
  - **OVERRUN Mode**: This mode is activated when an overrun occurs. The time at which an overrun happens on a processor is determined randomly.
- **Task Execution Times**:
  - **High Execution Time**: Used for tasks that require longer processing time or have higher priority.
  - **Low Execution Time**: Assigned to tasks with lower criticality or priority.
- **Task Migration and Scheduling**:
  - When an overrun is detected, the system triggers a reschedule function. This function facilitates the migration of low criticality tasks to a different core, predetermined in the offline scheduling phase, and the rescheduling of high-priority tasks with their higher Worst Case Execution Time (WCET).

This approach ensures a dynamic response to overruns, prioritizing high-priority tasks and reallocating resources efficiently to maintain system stability and performance.

9 EDF-VD Scheduling Algorithm

The Earliest Deadline First with Virtual Deadlines (EDF-VD) scheduling algorithm is a fundamental approach used in real-time systems to manage the execution of tasks with different criticality levels and deadlines. EDF-VD extends the classic EDF algorithm to address the challenges of mixed-criticality systems, where tasks may have varying levels of importance and stringent scheduling requirements.

9.1 Overview

In EDF-VD, tasks are categorized into two primary criticality levels: ”high” and ”low.” The algorithm aims to prioritize high-criticality tasks to ensure their timely execution while providing a level of service to low-criticality tasks.

9.2 Key Concepts

The EDF-VD algorithm introduces several key concepts:

10 WFD

1. **Virtual Deadlines**: Unlike the standard EDF algorithm, EDF-VD sets virtual deadlines for low-criticality tasks. These virtual deadlines are typically shorter than the actual deadlines of low-criticality tasks. Virtual deadlines allow for more flexibility in scheduling and enable high-criticality tasks to take priority when necessary.
2. **Deadline Calculation**: High-criticality tasks maintain their original deadlines based on their worst-case execution times (WCET). In contrast, low-criticality tasks have their deadlines calculated based on virtual deadlines.
3. **Quality of Service (QoS)**: EDF-VD considers Quality of Service (QoS) requirements for low-criticality tasks. It ensures that low-criticality tasks do not interfere with the execution of high-criticality tasks, meeting the system’s performance objectives.
4. **Deadline Miss Handling**: EDF-VD provides a mechanism to handle deadline misses differently for high and low-criticality tasks. Deadline misses for high-criticality tasks are treated with higher priority than those for low-criticality tasks, reflecting their relative importance.

10.1 Algorithm Steps

The EDF-VD scheduling algorithm involves several key steps:

1. **Initialization**: Initialize data structures for task tracking and execution.
2. **Task Selection**: Select the task with the earliest deadline for execution, considering both actual and virtual deadlines.
3. **Execution**: Execute the selected task for a specified time unit.
4. **Deadline Adjustment**: Adjust deadlines for tasks as they are executed, based on their criticality levels.
5. **Criticality Switch**: If a high-criticality task exceeds its low-criticality WCET, switch to high-criticality mode, altering scheduling priorities.
6. **Completion and Deadline Monitoring**: Continuously monitor task completion and deadlines, handling task completions and missed deadlines accordingly.

10.2 Advantages

EDF-VD offers several advantages:

- Effective Handling of Mixed-Criticality Systems: EDF-VD is well-suited for systems with tasks of varying criticality levels, ensuring that high-criticality tasks meet their deadlines while maintaining QoS for low-criticality tasks.
- Flexibility and Adaptability: The use of virtual deadlines provides flexibility in scheduling, allowing the system to adapt to changing conditions and prioritize critical tasks when needed.
- Improved QoS Reporting: EDF-VD allows for the monitoring and reporting of QoS metrics, making it suitable for systems with strict performance requirements.

10.3 Conclusion

The EDF-VD scheduling algorithm extends the capabilities of the classic EDF algorithm to meet the demands of mixed-criticality real-time systems. By introducing virtual deadlines and considering criticality levels, EDF-VD strikes a balance between meeting stringent deadlines and providing quality of service, making it a valuable tool in the design of reliable and adaptable real-time systems.

11 Our Algorithm implementation

In our implementation we look for an algorithm that tries to improve Quality of service

11.1 Overview

at offline phase we set which cores can be a HOST core, which means that core has utilization below 1. At onlie mode when overrun happens we move the low critical tasks to the not overruned cores that was labled HOST and assign each task to lowest utilization core available.

11.2 Example of Migration

initial task assignment

core number	Tasks
0	Task(20, LOW)
1	Task(15, HIGH), Task(18, LOW)
2	Task(4015, HIGH), Task(7, HIGH), Task(4008, HIGH)
3	Task(4016, HIGH), Task(4007, HIGH)
4	Task(17, HIGH), Task(10, LOW)
5	Task(4017, HIGH), Task(4, LOW)
6	Task(4018, HIGH), Task(16, LOW)
7	Task(1, HIGH), Task(5, HIGH)
8	Task(4001, HIGH), Task(4005, HIGH)
9	Task(4002, HIGH), Task(4006, HIGH)
10	Task(19, HIGH), Task(11, HIGH)
11	Task(4019, HIGH), Task(4011, HIGH)
12	Task(4020, HIGH), Task(4012, HIGH)
13	Task(14, LOW), Task(9, HIGH)
14	Task(2, LOW), Task(4014, HIGH)
15	Task(13, HIGH), Task(4013, HIGH)

overruned cores
2, 10, 15, 0, 5, 8, 11, 13
host cores
1, 3, 4, 6, 7, 9, 12, 14
remaining tasks before overrun

core number	Tasks
0	
1	Task(15, HIGH)
2	Task(4015, HIGH)
3	Task(4016, HIGH)
4	Task(17, HIGH)
5	Task(4017, HIGH)
6	Task(4018, HIGH)
7	Task(1, HIGH)
8	Task(4001, HIGH)
9	Task(4002, HIGH)
10	Task(19, HIGH)
11	Task(4019, HIGH)
12	Task(4020, HIGH)
13	Task(9, HIGH)
14	Task(2, LOW), Task(4014, HIGH)
15	Task(13, HIGH), Task(4013, HIGH)

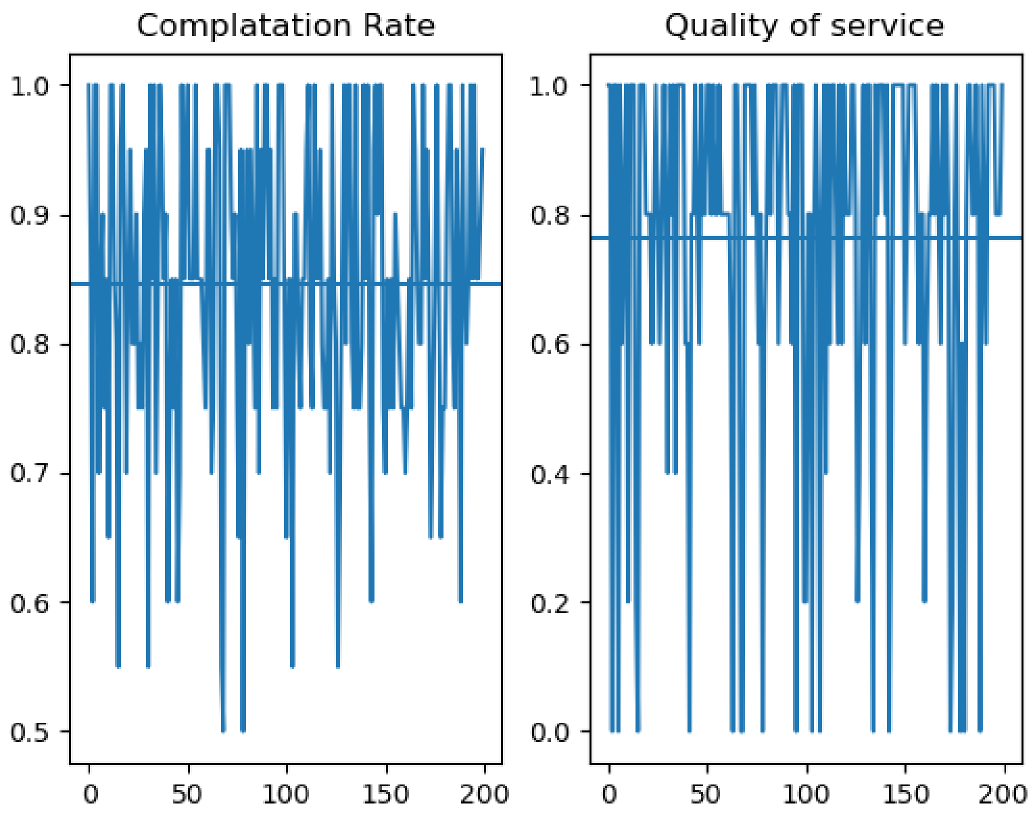
task assignment after overrun

core number	Tasks
0	
1	Task(15, HIGH)
2	Task(4015, HIGH)
3	Task(4016, HIGH)
4	Task(17, HIGH)
5	Task(4017, HIGH)
6	Task(4018, HIGH)
7	Task(1, HIGH), Task(14, LOW)
8	Task(4001, HIGH)
9	Task(4002, HIGH)
10	Task(19, HIGH)
11	Task(4019, HIGH)
12	Task(4020, HIGH), Task(20, LOW)
13	Task(9, HIGH)
14	Task(2, LOW), Task(4014, HIGH)
15	Task(13, HIGH), Task(4013, HIGH)

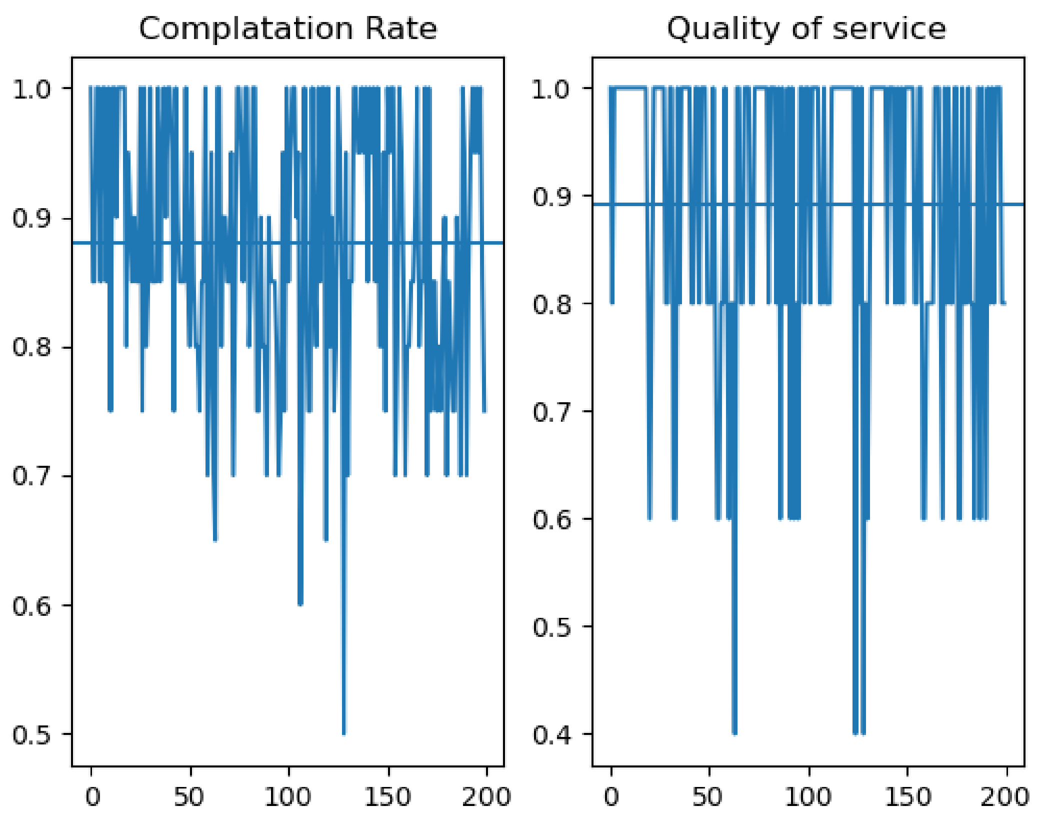
11.3 Results

This is the Quality of service and Complatation Rate from edf-vd in our implementation.

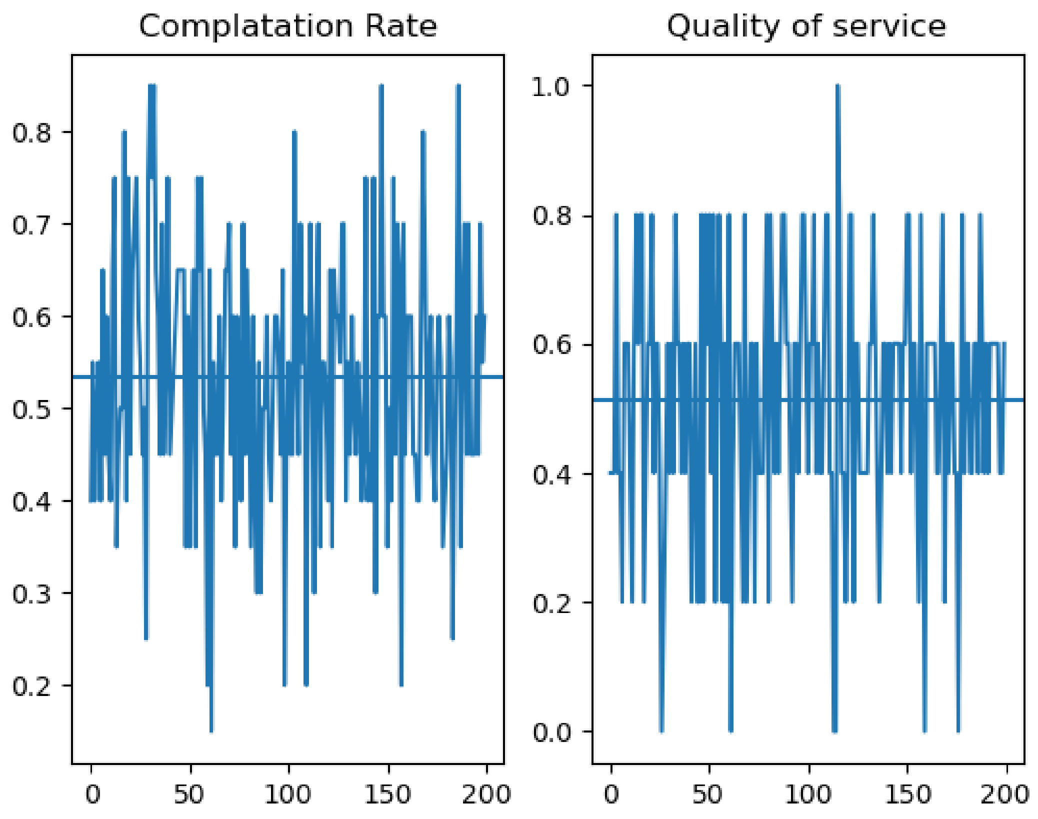
1. **8 Core, 50 percent Utilization**
   Edf-vd



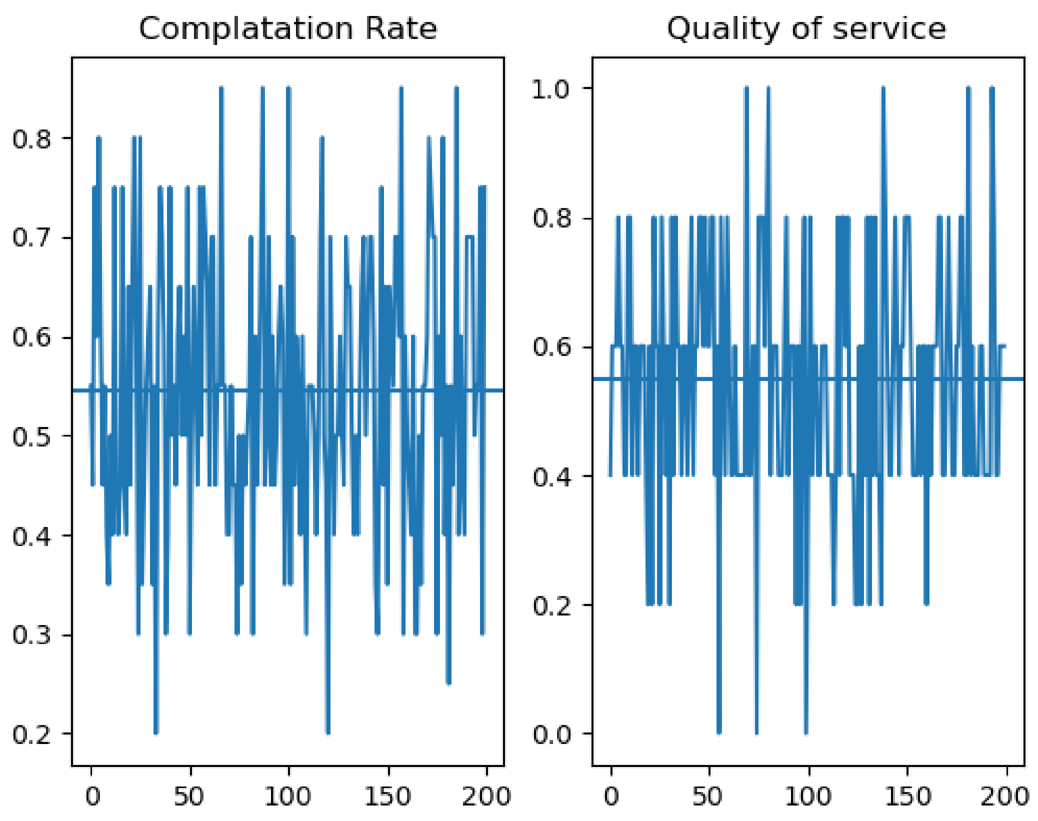
Our implemenation



2. **16 Core, 75 percent Utilization**
   Edf-vd



Our implemenation



11.4 Conclusion

As we can see in both cases this algorithm has improved the Qos.