SHARIF UNIVERSITY OF TECHNOLOGY

COMPUTER ENGINEERING

REAL TIME SYSTEMS

Final Project

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

Instructor:
Dr. Safari

January 7, 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 peri-vedic Low Criticality (LC) and High Criticality (HC) tasks. The system is designed to operate under two core mapping policies: Worst Fit Decreasing (WFD) and First Fit Decreasing (FFD), with a priority on mapping both original tasks and their copies. Initially functioning in a normal state, the system is equipped to handle core overruns by transferring LC tasks to other cores with available capacity, thereby introducing a new operational state termed 'HOST'. This adaptive mechanism ensures the continuous and effective execution of tasks, particularly emphasizing the importance of HC tasks whose timely and accurate execution is vital. The scheduling of tasks across cores is governed by the VD-EDF (Variable Deadline Earliest Deadline First) algorithm, striking a balance between the criticality of HC tasks and the need to maintain a minimum quality of service across the system. This report delves into the intricacies of this system, highlighting its potential in enhancing task management in complex multi-core environments.

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