# On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors

Piyush Garyali

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

July 16, 2010
Blacksburg, Virginia

Keywords: Multiprocessor Real-Time Scheduling, Utility Accrual Scheduling, Time/Utility Functions, Real-Time Linux

# On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors

Piyush Garyali

(ABSTRACT)

We consider the problem of scheduling real-time tasks on a multiprocessor system. Our primary focus is scheduling on multiprocessor systems where the total task utilization demand, $U$, is greater than $m$, the number of processors on a multiprocessor system—i.e., the total available processing capacity of the system. When $U > m$, the system is said to be overloaded; otherwise, the system is said to be underloaded.

While significant literature exists on multiprocessor real-time scheduling during underloads, little is known about scheduling during overloads, in particular, in the presence of task dependencies—e.g., due to synchronization constraints. We consider real-time tasks that are subject to time/utility function (or TUF) time constraints, which allow task urgency to be expressed independently of task importance—e.g., the most urgent task being the least important. The urgency/importance decoupling allowed by TUFs is especially important during overloads, when not all tasks can be optimally completed. We consider the timeliness optimization objective of maximizing the total accrued utility and the number of deadlines satisfied during overloads, while ensuring task mutual exclusion constraints and freedom from deadlocks. This problem is NP-hard. We develop a class of polynomial-time heuristic algorithms, called the *Global Utility Accrual* (or GUA) class of algorithms.

The algorithms construct a directed acyclic graph representation of the task dependency relationship, and build a global multiprocessor schedule of the *zero in-degree* tasks to heuristically maximize the total accrued utility and ensure mutual exclusion. Potential deadlocks are detected through a cycle-detection algorithm, and resolved by aborting a task in the deadlock cycle. The GUA class of algorithms include two algorithms, namely, the *Non-Greedy Global Utility Accrual* (or NG-GUA) and *Greedy Global Utility Accrual* (or G-GUA) algorithms. NG-GUA and G-GUA differ in the way schedules are constructed towards meeting all task deadlines, when possible to do so. We establish several properties of the algorithms including conditions under which all task deadlines are met, satisfaction of mutual exclusion constraints, and deadlock-freedom.

We create a Linux-based real-time kernel called ChronOS for multiprocessors. ChronOS is extended from the `PREEMPT_RT` real-time Linux patch, which provides optimized interrupt service latencies and real-time locking primitives. ChronOS provides a scheduling framework for the implementation of a broad range of real-time scheduling algorithms, including utility accrual, non-utility accrual, global, and partitioned scheduling algorithms.

We implement the GUA class of algorithms and their competitors in ChronOS and conduct experimental studies. The competitors include G-EDF, G-NP-EDF, G-FIFO, gMUA, P-EDF and P-DASA. Our study reveals that the GUA class of algorithms accrue higher utility and satisfy greater number of deadlines than the deadline-based scheduling algorithms by as much as 750% and 600%, respectively. In addition, we observe that G-GUA accrues higher utility than NG-GUA during overloads by as much as 25% while NG-GUA satisfies greater number of deadlines than G-GUA by as much as 5% during underloads.

This work was supported by the US National Science Foundation CCF (Software and Hardware Foundations) under Grant 0915895.

# Dedication

I dedicate this thesis to my wife, Kalpana.

*Without your support this would not have been possible.*

# Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, for his help and guidance on both technical and personal topics. It has been an honor to work under him and I am highly indebted to him for his trust in me.

I would also like to thank Dr. Paul Plassmann and Dr. Robert Broadwater, for serving on my committee and providing their valuable feedback and direction. In addition, I would like to thank all of my colleagues at the real-time systems lab. I would particularly like to thank Matthew Dellinger and Sherif Fahmy for their support and encouragement. It was a pleasure to work with them and make ChronOS a reality.

Finally, I would like to thank my family and friends for all the love and support they have given me, without which this thesis would not have been possible.

All figures in thesis are the work of the author, unless specified otherwise.

# Contents

**1 Introduction** **1**

1.1 Timeliness Model . . . . . . . . . . 2

1.2 Multiprocessor Real-Time Scheduling . . . . . . . . . . 4

1.3 Multiprocessor Real-Time Scheduling During Overloads . . . . . . . . . . 6

1.4 Research Contributions . . . . . . . . . . 7

1.5 Thesis Organization . . . . . . . . . . 9

**2 Related Work** **10**

2.1 Classification Based on Degree of Task Migration . . . . . . . . . . 10

2.2 Classification Based on Utilization Load . . . . . . . . . . 11

2.3 Summary . . . . . . . . . . 12

**3 Models and Objective** **14**

3.1 Thread Model . . . . . . . . . . 14

3.2 Timeliness Model . . . . . . . . . . 14

3.3 Resource Model . . . . . . . . . . 15

3.4 Processor Model . . . . . . . . . . 15

3.5 Abort Model . . . . . . . . . . 15

3.6 Scheduling Objective . . . . . . . . . . 16

**4 The GUA Class of Real-Time Scheduling Algorithms** **17**

4.1 Ensuring Mutual Exclusion . . . . . . . . . . 17

4.2 Maximizing Accrued Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3 Deadlock Detection and Resolution . . . . . . . . . . . . . . . . . . . . . . . . 20
4.4 Assigning Zero In-degree Phases to Processors . . . . . . . . . . . . . . . . . . 20
4.5 Data Structures and Auxiliary Functions . . . . . . . . . . . . . . . . . . . . . 21
4.5.1 Data Structure for Timeliness Model . . . . . . . . . . . . . . . . . . . 21
4.5.2 Data Structure for Resource Model . . . . . . . . . . . . . . . . . . . . 22
4.5.3 Auxiliary Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.5.4 Creation of DAG with Detection/Resolution of Deadlocks . . . . . . . 25
4.6 Non-greedy Global Utility Accrual (NG-GUA) . . . . . . . . . . . . . . . . . . 27
4.7 Greedy Global Utility Accrual (G-GUA) . . . . . . . . . . . . . . . . . . . . . 30
4.8 Algorithm Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**5 ChronOS Real-Time Linux 37**

5.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.2.1 `PREEMPT_RT` Real-Time Patch . . . . . . . . . . . . . . . . . . . . . 39
5.3 ChronOS Real-Time Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.3.1 Priority Bit-map and Run-queues . . . . . . . . . . . . . . . . . . . . . 40
5.3.2 Scheduling Real-Time Tasks . . . . . . . . . . . . . . . . . . . . . . . . 41
5.4 Multiprocessor Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.4.1 Partitioned Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.4.2 Global Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.4.2.1 Application Concurrent Scheduling Model . . . . . . . . . . 46
5.4.2.2 Stop-the-World Scheduling Model . . . . . . . . . . . . . . . 47
5.4.3 Mapping Tasks to Processors . . . . . . . . . . . . . . . . . . . . . . . 49
5.5 Implementing Scheduling Algorithms in ChronOS . . . . . . . . . . . . . . . . 50
5.5.1 Single-processor Scheduling . . . . . . . . . . . . . . . . . . . . . . . . 50
5.5.2 Multiprocessor Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . 51

**6 Experimental Results** **52**
6.1 Experimental Setup . . . . . 52
6.1.1 Platform Specifications . . . . . 52
6.1.2 Test Application . . . . . 52
6.1.3 Performance Parameters . . . . . 53
6.2 Results Without Dependencies . . . . . 54
6.2.1 Comparison with Global Scheduling Algorithms . . . . . 54
6.2.1.1 Two-core Platform Results . . . . . 56
6.2.1.2 Four-core Platform Results . . . . . 67
6.2.1.3 Eight-core Platform Results . . . . . 78
6.2.2 Comparison with Partitioned Scheduling Algorithms . . . . . 85
6.3 Results With Dependencies . . . . . 90
6.3.1 Varying CPU utilization Load . . . . . 91
6.3.2 Varying Lock Critical Section Length . . . . . 97
6.3.3 Varying Number of Locks per Task . . . . . 100
6.4 Overhead Measurements . . . . . 103
6.5 Conclusions . . . . . 107

**7 Conclusions and Future Work** **108**
7.1 Contributions . . . . . 111
7.2 Future Work . . . . . 111
7.2.1 Parallelizing Schedule Creation . . . . . 111
7.2.2 Reducing Scheduling Overheads . . . . . 112
7.2.3 Cache-Aware Algorithms . . . . . 112
7.2.4 Approximate Algorithms . . . . . 112

**A ChronOS System Calls** **113**

**B ChronOS Kernel Data Structures** **114**

**C Sample ChronOS Scheduling Kernel Modules** **117**

**Bibliography** **120**

# List of Figures

1.1 Example TUF time constraints. (a) MITRE Airborne Warning and Control System (AWACS) *association* TUF [30]; (b-c) GD/CMU air defense *plot correlation*, *track maintenance*, and *missle control* TUFs [58]; (d) step TUFs . . 3

1.2 Sample EDF schedule on two processors – (a) that cannot be scheduled; and (b) that can be scheduled. . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 Schedulability bounds of multiprocessor scheduling algorithms from [49]. . . 6

2.1 Characterization of scheduling algorithms during underloads ($U \leq m$) and overloads ($U > m$) based on ability to meet all deadlines and bounded best-effort real-time time interval. . . . . . . . . . . . . . . . . . . . . . . . . . . 12

4.1 A phase and resource dependency chain with nine phases and three resources 18

4.2 Directed Acyclic Graph (DAG) for the dependency chain shown in Fig 4.1. The nodes with a zero in-degree are eligible for final schedule in order to preserve mutual exclusion. . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4.3 Computing the global value density for a graph . . . . . . . . . . . . . . . 19

4.4 A phase and resource dependency chain with a deadlock condition. . . . . . 20

4.5 Deadlock detection and resolution. . . . . . . . . . . . . . . . . . . . . . . 21

4.6 Data structure representation of the DAG . . . . . . . . . . . . . . . . . . 23

4.7 Sample schedule for NG-GUA . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.8 Sample schedule for G-GUA . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.1 ChronOS architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

5.2 Linux real-time priority bitmap and run-queues . . . . . . . . . . . . . . . 40

5.3 ChronOS real-time priority run-queue for a given real-time priority . . . . . 41

5.4 ChronOS scheduling approach on a single-processor system . . . . . . . . . . 43
5.5 ChronOS partitioned scheduling approach on a multiprocessor system . . . . 44
5.6 ChronOS global scheduling approach on a multiprocessor system . . . . . . . 45
5.7 "Application Concurrent" architecture model for global scheduling algorithms 47
5.8 "Stop-the-World" architecture model for global scheduling algorithms . . . . 48
5.9 Task mapping for global scheduling algorithms in ChronOS . . . . . . . . . . 50

6.1 Task-sets created with variable TUFs (a) Increasing Utility (IU); (b) Decreasing Utility (DU); (c) Random Utility(RU) . . . . . . . . . . . . . . . . . . . . 54
6.2 AUR vs. CPU utilization (5T, 2C, NL, DU) . . . . . . . . . . . . . . . . . . 57
6.3 DSR vs. CPU utilization (5T, 2C, NL, DU) . . . . . . . . . . . . . . . . . . 58
6.4 AUR vs. CPU utilization (5T, 2C, NL, IU) . . . . . . . . . . . . . . . . . . 59
6.5 DSR vs. CPU utilization (5T, 2C, NL, IU) . . . . . . . . . . . . . . . . . . 59
6.6 AUR vs. CPU utilization (5T, 2C, NL, RU) . . . . . . . . . . . . . . . . . . 60
6.7 DSR vs. CPU utilization (5T, 2C, NL, RU) . . . . . . . . . . . . . . . . . . 60
6.8 AUR vs. CPU utilization (12T, 2C, NL, DU) . . . . . . . . . . . . . . . . . 61
6.9 DSR vs. CPU utilization (12T, 2C, NL, DU) . . . . . . . . . . . . . . . . . 62
6.10 AUR vs. CPU utilization (12T, 2C, NL, IU) . . . . . . . . . . . . . . . . . 62
6.11 DSR vs. CPU utilization (12T, 2C, NL, IU) . . . . . . . . . . . . . . . . . 63
6.12 AUR vs. CPU utilization (12T, 2C, NL, RU) . . . . . . . . . . . . . . . . . 63
6.13 DSR vs. CPU utilization (12T, 2C, NL, RU) . . . . . . . . . . . . . . . . . 64
6.14 AUR vs. CPU utilization (27T, 2C, NL, DU) . . . . . . . . . . . . . . . . . 64
6.15 DSR vs. CPU utilization (27T, 2C, NL, DU) . . . . . . . . . . . . . . . . . 65
6.16 AUR vs. CPU utilization (27T, 2C, NL, IU) . . . . . . . . . . . . . . . . . 65
6.17 DSR vs. CPU utilization (27T, 2C, NL, IU) . . . . . . . . . . . . . . . . . 66
6.18 AUR vs. CPU utilization (27T, 2C, NL, RU) . . . . . . . . . . . . . . . . . 66
6.19 DSR vs. CPU utilization (27T, 2C, NL, RU) . . . . . . . . . . . . . . . . . 67
6.20 AUR vs. CPU utilization (5T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . 68
6.21 DSR vs. CPU utilization (5T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . 69

6.22 AUR vs. CPU utilization (5T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . . 69
6.23 DSR vs. CPU utilization (5T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . . 70
6.24 AUR vs. CPU utilization (5T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 70
6.25 DSR vs. CPU utilization (5T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 71
6.26 AUR vs. CPU utilization (12T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . 72
6.27 DSR vs. CPU utilization (12T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . . 72
6.28 AUR vs. CPU utilization (12T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . 73
6.29 DSR vs. CPU utilization (12T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . 73
6.30 AUR vs. CPU utilization (12T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 74
6.31 DSR vs. CPU utilization (12T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 74
6.32 AUR vs. CPU utilization (27T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . 75
6.33 DSR vs. CPU utilization (27T, 4C, NL, DU) . . . . . . . . . . . . . . . . . . . 75
6.34 AUR vs. CPU utilization (27T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . 76
6.35 DSR vs. CPU utilization (27T, 4C, NL, IU) . . . . . . . . . . . . . . . . . . . 76
6.36 AUR vs. CPU utilization (27T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 77
6.37 DSR vs. CPU utilization (27T, 4C, NL, RU) . . . . . . . . . . . . . . . . . . . 77
6.38 AUR vs. CPU utilization (27T, 8C, NL, DU) . . . . . . . . . . . . . . . . . . 79
6.39 DSR vs. CPU utilization (27T, 8C, NL, DU) . . . . . . . . . . . . . . . . . . . 79
6.40 AUR vs. CPU utilization (27T, 8C, NL, IU) . . . . . . . . . . . . . . . . . . . 80
6.41 DSR vs. CPU utilization (27T, 8C, NL, IU) . . . . . . . . . . . . . . . . . . . 81
6.42 AUR vs. CPU utilization (27T, 8C, NL RU) . . . . . . . . . . . . . . . . . . . 81
6.43 DSR vs. CPU utilization (27T, 8C, NL, RU) . . . . . . . . . . . . . . . . . . . 82
6.44 AUR vs. CPU utilization (50T, 8C, NL, DU) . . . . . . . . . . . . . . . . . . 82
6.45 DSR vs. CPU utilization (50T, 8C, NL, DU) . . . . . . . . . . . . . . . . . . 83
6.46 AUR vs. CPU utilization (50T, 8C, NL, IU) . . . . . . . . . . . . . . . . . . . 83
6.47 DSR vs. CPU utilization (50T, 8C, NL, IU) . . . . . . . . . . . . . . . . . . . 84
6.48 AUR vs. CPU utilization (50T, 8C, NL, RU) . . . . . . . . . . . . . . . . . . . 84
6.49 DSR vs. CPU utilization (50T, 8C, NL, RU) . . . . . . . . . . . . . . . . . . . 85

6.50 AUR vs. CPU utilization (12T, 4C, NL, RU, BF) . . . . . . . . . . . . . . . 88
6.51 DSR vs. CPU utilization (12T, 4C, NL, RU, BF) . . . . . . . . . . . . . . . 88
6.52 AUR vs. CPU utilization (27T, 4C, NL, RU, BF) . . . . . . . . . . . . . . . 89
6.53 DSR vs. CPU utilization (27T, 4C, NL, RU, BF) . . . . . . . . . . . . . . . 89
6.54 AUR vs. CPU utilization (12T, 4C, 1L, RU, 5% CS) . . . . . . . . . . . . . 92
6.55 DSR vs. CPU utilization (12T, 4C, 1L, RU, 5% CS) . . . . . . . . . . . . . 93
6.56 AUR vs. CPU utilization (12T, 4C, 1L, RU, 25% CS) . . . . . . . . . . . . 93
6.57 DSR vs. CPU utilization (12T, 4C, 1L, RU, 25% CS) . . . . . . . . . . . . 94
6.58 AUR vs. CPU utilization (12T, 4C, 4L, RU, 5% CS) . . . . . . . . . . . . . 95
6.59 DSR vs. CPU utilization (12T, 4C, 4L, RU, 5% CS) . . . . . . . . . . . . . 95
6.60 AUR vs. CPU utilization (12T, 4C, 4L, RU, 25% CS) . . . . . . . . . . . . 96
6.61 DSR vs. CPU utilization (12T, 4C, 4L, RU, 25% CS) . . . . . . . . . . . . 96
6.62 AUR vs. Critical section length (12T, 4C, 1L, RU, 400% CPU utilization) . 98
6.63 AUR vs. Critical section length (12T, 4C, 1L, RU, 800% CPU utilization) . 98
6.64 AUR vs. Critical section length (12T, 4C, 4L, RU, 400% CPU utilization) . 99
6.65 AUR vs. Critical section length (12T, 4C, 4L, RU, 800% CPU utilization) . 99
6.66 AUR vs. Number of locks (12T, 4C, RU, 5% CS, 400% CPU utilization) . . 101
6.67 AUR vs. Number of locks (12T, 4C, RU, 25% CS, 400% CPU utilization) . 101
6.68 AUR vs. Number of locks (12T, 4C, RU, 5% CS, 800% CPU utilization) . . 102
6.69 AUR vs. Number of locks (12T, 4C, RU, 25% CS, 800% CPU utilization) . . 102
6.70 Scheduling overheads for G-GUA under variable utilization load . . . . . . . 104
6.71 Scheduling overheads for NG-GUA under variable utilization load . . . . . . 104
6.72 G-GUA vs. NG-GUA, scheduling overheads with variable number of tasks . 105
6.73 Comparison of scheduling overheads at 100% utilization load . . . . . . . . . 105
6.74 Comparison of scheduling overheads at 800% utilization load . . . . . . . . . 106

# List of Algorithms

1 Creation of DAG with detection/resolution of deadlocks . . . . . . . . . . . . . 26
2 NG-GUA: Non-greedy Global Utility Accrual . . . . . . . . . . . . . . . . . . . . 28
3 G-GUA: Greedy Global Utility Accrual . . . . . . . . . . . . . . . . . . . . . . . . 31

# List of Tables

6.1 Specifications of the various platforms used for experimental evaluation . . . 53

6.2 Legend used in experimental results, based on TUF-based models, for comparison with other global scheduling algorithms . . . . . . . . . . . . . . . . 56

6.3 Legend used in experimental results for comparison with other partitioned scheduling algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

6.4 Legend used in experimental results for comparison with other global scheduling algorithms in the presence of dependencies . . . . . . . . . . . . . . . . . 91

# Nomenclature

| | |
|---|---|
| ACUA | Asynchronous Consensus-based Utility Accrual scheduling algorithm, page 38 |
| API | Application Programming Interface, page 38 |
| AUR | Accrued Utility Ratio, page 53 |
| CPU | Central Processing Unit, page 11 |
| CRT-RQ | ChronOS Real-Time Run-Queue, page 41 |
| CUA | Consensus-based Utility Accrual scheduling algorithm, page 38 |
| D-TPR | Distributed Thread Polling with bounded Recovery, page 38 |
| DAG | Directed Acyclic Graph, page 17 |
| DASA | Dependent Activity Scheduling Algorithm, page 7 |
| DMS | Deadline Monotonic Scheduling, page 2 |
| DSR | Deadline Satisfaction Ratio, page 53 |
| DT | Distributed Thread, page 37 |
| DU | Decreasing Utility model, page 55 |
| EDF | Earliest Deadline First, page 2 |
| G-EDF | Global Earliest Deadline First scheduling algorithm, page 28 |
| G-FIFO | Global First In First Out scheduling algorithm, page 38 |
| G-GUA | Greedy Global Utility Accrual, page 7 |
| G-NP-EDF | Global Non-Preemptive Earliest Deadline First scheduling algorithm, page 38 |
| gMUA | Global Multiprocessor Utility Accrual, page 6 |

| | |
|---|---|
| GNU | Gnu is Not Unix, page 37 |
| GPLv2 | GNU Public License version 2.0, page 37 |
| GVD | Global Value Density, page 19 |
| HT | Hyper-Threading, page 78 |
| HUA | Handler-assured Utility Accrual scheduling algorithm, page 38 |
| HVDF | Highest Value Density First scheduling algorithm, page 38 |
| IPI | Inter-Processor Interrupt, page 48 |
| IRQ | Interrupt Request, page 39 |
| IU | Increasing Utility model, page 54 |
| LLF | Least Laxity First, page 2 |
| LLREF | Largest Local Remaining Execution time First scheduling algorithm, page 5 |
| LVD | Local Value Density, page 19 |
| MOCA | Multiprocessor On-Line Competitive Algorithm, page 6 |
| NG-GUA | Non-Greedy Global Utility Accrual, page 7 |
| P-DASA | Parallel Dependent Activity Scheduling Algorithm, page 44 |
| P-EDF | Partitioned Earliest Deadline First scheduling algorithm, page 10 |
| PFair | Proportionate Fair Scheduling Algorithm, page 11 |
| PIP | Priority Inheritance Protocol, page 22 |
| POSIX | Portable Operating System Interface [for Unix], page 39 |
| PUD | Potential Utility Density, page 7 |
| RMS | Rate Monotonic Scheduling algorithm, page 2 |
| RU | Random Utility model, page 55 |
| SMP | Symmetrical Multiprocessor System, page 40 |
| STW | Stop-the-World architecture model, page 46 |
| TPR | Thread Polling with bounded Recovery, page 38 |
| TUF | Time/Utility Functions, page 3 |
| UA | Utility Accrual, page 3 |

# Chapter 1

# Introduction

Recently, there has been a shift in the computer industry from increasing clock rates to designing multi-core and hyper-threading architectures in a quest to produce faster computers [65]. Motivated by heat and power issues, most chip manufacturers have chosen the route of increasing system and chip level parallelism in preference to increasing clock rates in order to satisfy the need for improving performance. This trend extends to embedded systems [14] as well as to traditional computing systems. Consequently, the design of multiprocessor[1] real-time scheduling algorithms has become important in order to allow real-time applications to take advantage of these emerging architectures.

Scheduling of real-time multiprocessor systems has received increased attention recently [18]. However, most of these works target "underloaded" systems—i.e., systems where the total application task utilization demand, $U$, is always less than the total available processing capacity of the system, which is $m$ for an $m$-processor system. Majority of the research focus in multiprocessor real-time systems is on developing scheduling algorithms and understanding their *schedulability utilization bounds*—i.e., task utilization bounds below which all task deadlines are met. The premise of this approach is that it is possible to determine the worst-case execution-time behaviors of applications (e.g., task arrival behaviors, task worst-case execution times) and thereby determine the total task utilization demands. Once the task utilization demand is known, task schedulability—i.e., the ability to meet all task deadlines—can be ensured off-line by selecting the appropriate scheduling algorithm with a higher utilization bound.

For some applications (e.g., [68, 30, 66]), it is difficult to determine worst-case execution-time behaviors *a priori*, as they are subject to run-time exigencies, such as execution time overruns and unpredictable thread arrival patterns, causing transient and permanent overloads. When overloads occur, often, such applications desire graceful timeliness degradation—e.g., meeting as many deadlines of high importance tasks as possible, irrespective of task urgency.

[1]We will use multiprocessor and multi-core interchangeably in the rest of the thesis unless explicitly stated otherwise.

## 1.1 Timeliness Model

The state-of-the-real-time practice is to handle application time constraints using the concept of *priorities*. This approach is the basis for the vast majority of real-time OS, language, and middleware standards including POSIX [42], Real-time CORBA [61], Ada 95 [5], and RTSJ [71]. However, using priorities have inherent shortcomings, which include:

1. The process of assigning time constraints to priorities is generally difficult, and sometimes intractable. Often, there is significant loss of information during this process which makes it difficult to dependably satisfy time constraints.

2. Real-time systems usually comprise of various sub-systems. In order to assign priorities to express urgency, it is necessary to know the entire system and its various subsystems, along with the knowledge of the global assignment of priorities for the entire system. Such knowledge can sometimes be difficult to obtain due to organizational boundaries.

3. The urgency and importance of a real-time task can be orthogonal—e.g., the most important task can be the most urgent; the most important task can be the least urgent, etc. However, a priority cannot express both. This causes serious difficulties during overload situations, when not all task deadlines can be met, and applications desire to differentially allocate resources in the order of decreasing task importance, irrespective of task urgency—e.g., most important task first; second-most important task second, and so on.

One solution that overcomes these shortcomings is to provide system developers with abstractions for directly specifying time constraints (instead of mapping time constraints to priorities) and using those time constraint specifications to manage resources (instead of indirectly managing resources through the priority artifact). This is the basis for the traditional real-time theory [47, 54]. In that theory, time constraints (which are largely limited to deadlines) are mapped to fixed priorities in algorithms such as Rate Monotonic Scheduling (RMS) [53] and Deadline Monotonic Scheduling (DMS) [53], or are mapped to dynamic priorities as in Least Laxity First (LLF) [59], or are directly used for scheduling as in Earliest Deadline First (EDF) [41].

However, deadline and deadline-based scheduling suffer from some drawbacks. A deadline of a task can be expressed in two forms. First, as a binary-valued expression which evaluates to the deadline being met or not met; and second, a linear-valued expression for the penalty of lateness. However, the penalty of being late per unit time is constant irrespective of how late the task actually is. Thus, deadlines per se cannot be used to distinguish between task urgency and task importance, which is a limitation during overloads. Furthermore, classical deadline-based scheduling algorithms (e.g., EDF) suffer from the *domino* effect [55] during overloads. This is because, these algorithms always favor tasks with an earlier deadline, irrespective of how close such tasks are towards missing their deadlines. This results in those tasks missing their deadlines, and also causing others, which are now delayed, to miss their deadlines.

Timeliness optimality criteria that can be specified using deadlines fall into the following categories: (i) the criterion of meeting all deadlines; (ii) criteria that are based on number of missed deadlines—e.g., minimize the number of deadline misses; upper bound the number of deadlines missed; and (iii) the criteria expressed using lateness—e.g., minimize the maximum lateness; upper bound the lateness.

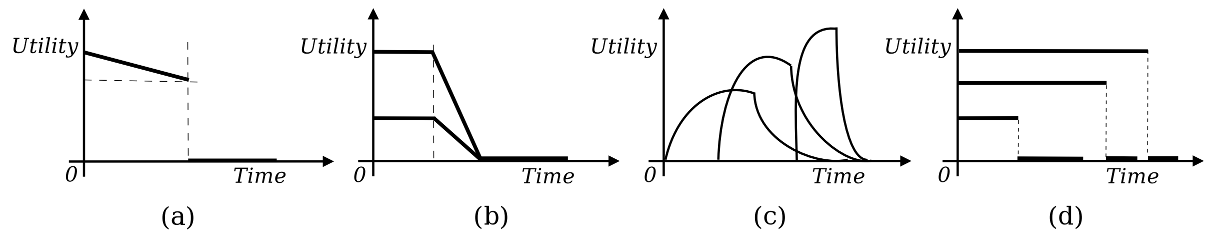


Figure 1.1: Example TUF time constraints. (a) MITRE Airborne Warning and Control System (AWACS) *association* TUF [30]; (b-c) GD/CMU air defense *plot correlation*, *track maintenance*, and *missle control* TUFs [58]; (d) step TUFs

The shortcomings of deadlines and deadline-based scheduling are overcome in the Time/Utility Function (TUF) model, first introduced in [43]. A TUF specifies the utility obtained by the completion of a task, as a function of that task's completion time. Figure 1.1 shows some example TUFs. Figure 1.1(a)-(c) show time constraints of some applications in the defense domain [30, 58]. A TUF decouples importance and urgency, with the urgency measured as a deadline on the X-axis, and importance measured as a utility (or value) on the Y-axis. The classical deadline is a special case of a TUF: a binary-valued, downward "step" TUF. Figure 1.1(d) shows an example.

When task time constraints are expressed using TUFs, the scheduling optimality criteria are often based on accrued utility—e.g., maximizing the sum of the tasks' attained utilities. Such criteria are called *Utility Accrual* (or UA) criteria and scheduling algorithms that optimize such criteria are called UA scheduling algorithms (e.g., [31, 55]).

UA scheduling algorithms that maximize total accrued utility under downward "step" TUFs, default to EDF during underloads, since EDF satisfies all deadlines during underloads. As a result, these algorithms obtain optimal total accrued utility during underloads. During overloads, UA scheduling algorithms favor tasks from whom greater utility can be accrued (as that generally tends to maximize the total accrued utility), irrespective of task urgency. This behavior of UA scheduling algorithms is called "best-effort" — i.e., the algorithms strive their best to feasibly complete as many high importance tasks as possible (where task importance is explicitly described using TUFs).[2] Note that the optimal timeliness behavior

[2]Note that the term "best-effort" as used in the networking context is intended to mean "least-effort."

of EDF is a special case of UA-scheduling.

A number of UA algorithms have been designed in the past. These cover a wide range of problem spaces: from processor scheduling [55, 31, 22, 50, 51, 52, 27]; to memory management and garbage collection [28, 23, 34]; to non-blocking synchronization [22, 24, 26]; to energy management [72, 73, 7, 74]; to packet scheduling [69, 70]; and to network routing [19, 20, 62]; They also cover step TUFs [31]; to non-step TUFs [22, 24, 26, 69, 72, 19, 62]; to abitrarily-shaped TUFs [50].

## 1.2 Multiprocessor Real-Time Scheduling

One unique aspect of multiprocessor real-time scheduling is the degree of run-time migration that is allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) *full migration*, where jobs are allowed to arbitrarily migrate across processors during their execution. This usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors and a processor-wide scheduling decision is made by a single (global) scheduling algorithm; (2) *no migration*, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled on their respective processors by processors' local scheduling algorithm, like single processor scheduling; and (3) *restricted migration*, where some form of migration is allowed—e.g., at job boundaries.

The partitioned scheduling paradigm has several advantages over the global approach. First, once tasks are allocated to processors, the multiprocessor real-time scheduling problem becomes a set of single processor real-time scheduling problems, one for each processor, which has been well-studied and for which optimal algorithms exist. Second, not migrating tasks at run-time means reduced run-time overhead as opposed to migrating tasks that may suffer cache misses on the newly assigned processor. If the task set is fixed and known a priori, the partitioned approach provides appropriate solutions [15].

The global scheduling paradigm also has advantages over the partitioned approach. First, if tasks can join and leave the system at run-time, then it may be necessary to reallocate tasks to processors in the partitioned approach [15]. Second, the partitioned approach cannot produce optimal real-time schedules — one that meets all task deadlines when task utilization demand does not exceed the total processor capacity — for periodic task sets [63], since the partitioning problem is analogous to the bin-packing problem which is known to be NP-hard in the strong sense. Third, in some embedded processor architectures with no cache and simpler structures, the overhead of migration has a lower impact on the performance [15]. Finally, global scheduling can theoretically contribute to an increased understanding of the properties and behaviors of real-time scheduling algorithms for multiprocessors.(See [40] for a detailed discussion on this).

Carpenter *et al.* [18] have catalogued multiprocessor real-time scheduling algorithms considering the degree of job migration and the complexity of priority mechanisms employed. The latter includes classes such as (1) *static*, where task priorities never change, e.g., rate-monotonic (RM); (2) *dynamic but fixed within a job*, where job priorities are fixed, e.g., earliest-deadline-first (EDF); and (3) *fully-dynamic*, where job priorities are dynamic.

The Pfair class of algorithms [11] that allow full migration and fully dynamic priorities have been shown to be theoretically optimal—i.e., they achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach [32, 63]. Under Pfair, tasks can be decomposed into several small uniform segments, which are then scheduled, causing frequent scheduling and migration.

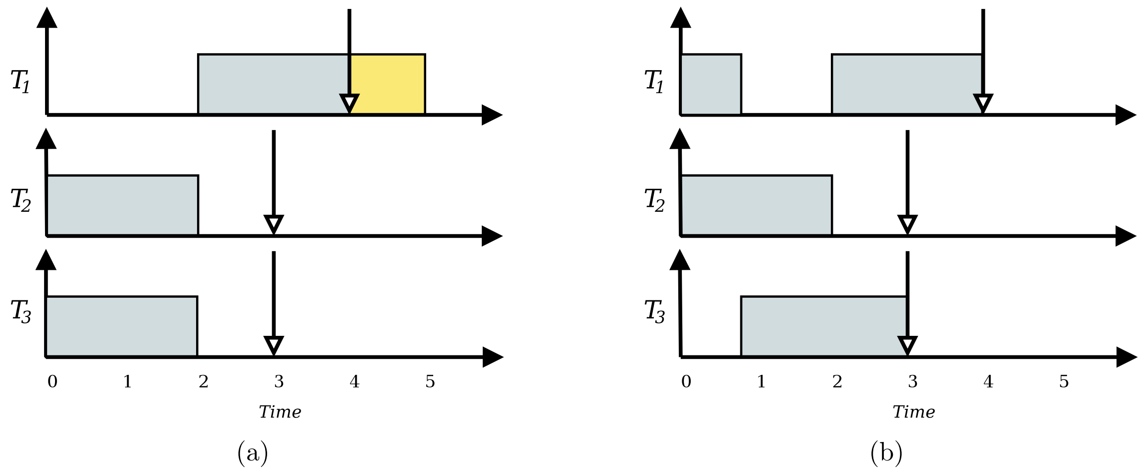


Figure 1.2: Sample EDF schedule on two processors – (a) that cannot be scheduled; and (b) that can be scheduled.

Thus, scheduling algorithms other than Pfair—e.g., global-EDF [15, 10, 2] have also been intensively studied though their schedulable utilization bounds are lower. Figure 1.2(a) shows an example task set that global-EDF cannot feasibly schedule. In Figure 1.2(a), task $T_1$ will miss its deadline when the system is given two processors. However, for the same task-set, there exists a schedule that meets all task deadlines, as shown in Figure 1.2(b).

There have also been efforts on designing optimal multiprocessor real-time scheduling algorithms that are not based on time quantum, unlike Pfair. Examples include the Largest Local Remaining Execution Time (or LLREF) algorithm and its derivatives [25, 21, 37].

Interestingly, all of these works exclude any run-time exigencies and consequent transient/permanent overloads—i.e., they presume that it is possible to determine the worst-case execution-time behaviors of applications (e.g., task arrival behaviors, task worst-case execution times), determine the total task utilization demands, and thus conduct off-line task schedulability, as mentioned before. Thus, the need for graceful timeliness degradation and best-effort timing assurances are outside their scope.

In [49], Lakshmanan *et al.* provide a classification of multiprocessor scheduling algorithms based on their schedulability bounds and we summarize the classification in Figure 1.3. In the global scheduling space, (at the time of writing this thesis), PFair and LLREF are the only known optimal multiprocessor scheduling algorithms that are able to meet all deadlines and have the optimal utilization bound of 100% (i.e., $U = m$) [11]. The dynamic priority algorithms that include deadline-based algorithms, such as G-EDF, and UA-based algorithms that default to G-EDF, such as gMUA, have the higher utilization bound of 50% [6, 22] (i.e., $U \approx m/2$). On the other hand, fixed priority algorithms, such as RMS, have a utilization bound of 33% [4] (i.e., $U \approx 3m/8$). In the partitioned scheduling space, both the dynamic priority algorithms, such as P-EDF and the fixed priority algorithms like P-DMS, have a utilization bound of 50% [56] (i.e., $U \approx m/2$). However, in [49], a version of P-DMS called PDMS-HPTS-DS, that uses high priority task-splitting (HPTS) with decreasing order of size (DS) is shown to have a utilization bound of 65%.

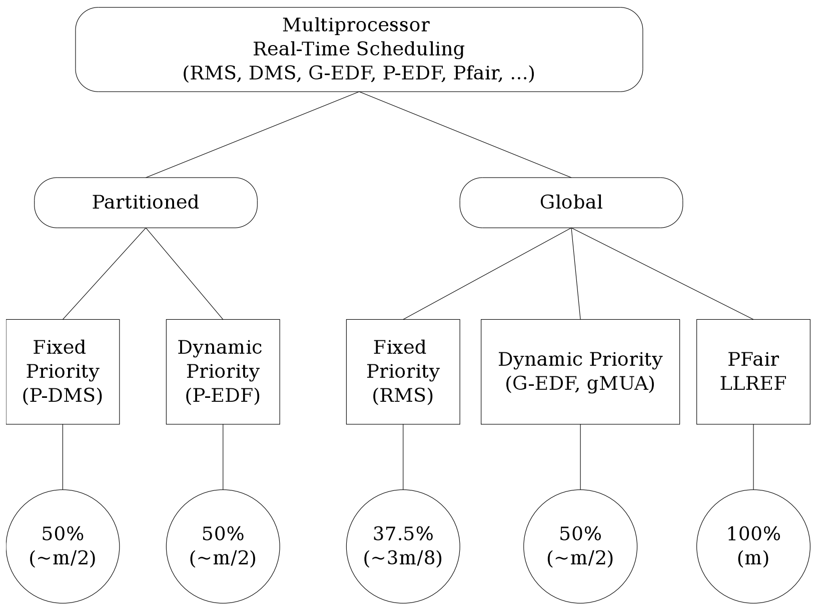


Figure 1.3: Schedulability bounds of multiprocessor scheduling algorithms from [49].

## 1.3 Multiprocessor Real-Time Scheduling During Overloads

Almost all of the past work on overload real-time scheduling [55, 31, 22, 50, 51, 52, 27], has focused on single processor systems, with very few exceptions. The only multiprocessor overload real-time scheduling algorithms that we are aware of (at the time of writing this thesis) include the Multiprocessor On-Line Competitive Algorithm (or MOCA) [48] and Global Multiprocessor Utility Accrual (or gMUA) [22] algorithms.

In [48], the authors determine the inherent upper bound on the best competitive ratio, in terms of accrued utility, that can be achieved by an online scheduler. They present a scheduling algorithm, MOCA, with a competitive ratio within a constant factor of the best competitive ratio found. MOCA divides the processors in a system into several "bands", each band being assigned a certain utility density. MOCA attempts to schedule an incoming task on a band that corresponds to its utility. If it is not possible to do so, the task is moved down the bands until one is found that can schedule it. As a result of this design, if a large number of low utility tasks arrive, they will quickly exhaust the lower bands of the system. This scenario may cause incoming low utility tasks to be rejected even though bands associated with higher utilities are idle.

gMUA is a global, best-effort real-time scheduling algorithm for multiprocessors. Like its single-processor counterparts [55, 31, 22, 50, 51, 52, 27], during underloads (i.e., $U \leq m$), gMUA defaults to global-EDF, and thus meets all deadlines (and accrues optimal total utility) up to a utilization bound of $m/2$. Also, like its single-processor counterparts, during overloads, the algorithm constructs a global multiprocessor schedule, favoring tasks from whom greater utility can be accrued, irrespective of task urgency, yielding graceful timeliness degradation and best-effort timeliness behavior.

Both MOCA and gMUA exclude task dependencies–i.e., they exclude task synchronization constraints (e.g., due to mutual exclusion) or precedence constraints.

## 1.4 Research Contributions

Thus, we observe a clear gap in the literature: how to schedule real-time tasks on multiprocessors that are subject to run-time uncertainties causing transient and permanent overloads, and task dependencies, such that optimal total utility can be accrued when possible and best-effort timeliness behavior, otherwise?

This problem is NP-hard because its one-processor version is shown to be NP-hard [31]. We solve this problem in the thesis by developing a class of polynomial-time heuristic algorithms, called the *Global Utility Accrual* (or GUA) class of algorithms. The algorithms construct a directed acyclic graph representation of the task dependency relationship, and build a global multiprocessor schedule of the *zero in-degree* tasks to ensure mutual exclusion. Potential deadlocks are detected through a cycle-detection algorithm, and resolved by aborting a task in the deadlock cycle. We use the heuristics of the Potential Utility Density (or PUD), which was defined by Clark in [31], to heuristically maximize the total accrued utility.

The GUA class of algorithms include two algorithms, namely, the *Non-Greedy Global Utility Accrual* (or NG-GUA) and *Greedy Global Utility Accrual* (or G-GUA) algorithms. NG-GUA and G-GUA differ in the way schedules are constructed towards meeting all task deadlines, when possible to do so. While NG-GUA constructs a global schedule that defaults to G-EDF's schedulability bound, G-GUA does not default to any algorithm. However, both

NG-GUA and G-GUA try to maximize the total accrued utility. The greediness in the name of these algorithms describes the tendency of the algorithms to accrue as much total utility as possible. As the names suggest, G-GUA is more greedy for utility accrual when compared to NG-GUA. We establish several properties of the algorithms including conditions under which all task deadlines are met, satisfaction of mutual exclusion constraints, and deadlock-freedom. We also establish the asymptotic cost of both the algorithms.

We create a Linux-based real-time kernel called ChronOS for multiprocessors. ChronOS is extended from the `PREEMPT_RT` real-time Linux patch [60], which provides optimized interrupt service latencies and real-time locking primitives. ChronOS provides a scheduling framework for the implementation of a broad range of real-time scheduling algorithms, including utility accrual, non-utility accrual, global, and partitioned scheduling algorithms. ChronOS provides a modular approach for development of the scheduling algorithms which can be implemented as kernel modules using the provided scheduler plugins.

We implement the GUA class of algorithms and their competitors in ChronOS and conduct experimental studies. The competitors include: EDF, G-EDF, G-NP-EDF, G-FIFO, gMUA, G-FIFO-PIP, G-NP-EDF-PIP, P-EDF and P-DASA. Our study reveals the following results:

(1) In the absence of dependencies, the GUA class of algorithms accrue higher utility and satisfy greater number of deadlines than the deadline-based algorithms (G-EDF, G-NP-EDF) by as much as 750% and 600%, respectively.

(2) In the absence of dependencies, the GUA class of algorithms accrue higher utility and satisfy greater number of deadlines compared to the partitioned algorithms by as much as 90% and 75%, respectively for P-DASA; and 450% and 600%, respectively for P-EDF.

(3) As gMUA defaults to NG-GUA without dependencies, the performance of NG-GUA and gMUA is similar.

(4) However, G-GUA outperforms NG-GUA and gMUA by accruing 25% more utility, while both NG-GUA and gMUA satisfy 5% more deadlines during underloads than G-GUA.

(5) In the presence of dependencies, both NG-GUA and G-GUA accrue higher utility and satisfy greater number of deadlines than G-NP-EDF-PIP by as much as 250% and 150%, respectively.

To summarize, the research contributions of the thesis include:

1. the GUA class of multiprocessor real-time scheduling algorithms that allow tasks to be subject to run-time uncertainties, overloads, and dependencies, and yield optimal total utility when possible and best-effort timeliness behavior otherwise — the first such multiprocessor real-time scheduling algorithms to do so.

2. the ChronOS multiprocessor real-time Linux kernel that provides optimized interrupt service latencies and real-time locking primitives (by virtue of `PREEMPT_RT` patch) and a scheduling framework that allows the implementation of a broad range of real-time

scheduling algorithms, including utility accrual, non-utility accrual, global, and partitioned scheduling algorithms – the first such multiprocessor real-time Linux kernel.

## 1.5 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 overviews past and related work in the multiprocessor real-time scheduling space, and contrasts them with the thesis's problem space. Chapter 3 describes our models and scheduling objective. Chapter 4 presents the GUA class of algorithms, including G-GUA and NG-GUA. The algorithm rationale, design, pseudo-code description, and algorithm properties are described in this chapter.

Chapter 5 describes the ChronOS real-time Linux. We report our experimental studies in Chapter 6. Finally, we conclude the thesis in Chapter 7.

# Chapter 2

# Related Work

In this chapter we survey the past and related work focusing on the problem of scheduling on multiprocessors. The work can be classified into two categories — (i) classification based on the degree of run-time migration; and (ii) classification based on the utilization load. We discuss these in Sections 2.1 and 2.2, respectively.

## 2.1 Classification Based on Degree of Task Migration

Carpenter *et al.* [18] have classified the scheduling algorithms based on the degree of run-time migration, considering three categories: (i) no migration; (ii) restricted migration; (iii) and full migration.

*No migration* refers to the approach of partitioned scheduling. In this approach, tasks are assigned to processors using an off-line assignment algorithm and each processor runs an instance of an independent single processor algorithm. Partitioned Earliest Deadline First (P-EDF) [2] is one example that uses the optimal uniprocessor scheduling algorithm EDF. As the partitioned approach involves partitioning the tasks among processors, it can be shown to be equivalent to the NP-Hard bin-packing problem. The tasks are usually partitioned using polynomial time heuristic algorithms such as, first-fit, worst-fit and best-fit. In [9], Baruah *et al.* present an optimized first-fit partitioning algorithm. The key idea of the algorithm is to assign tasks to processor bins such that the tasks assigned to the processors create a feasible schedule.

In the *restricted migration* category, jobs are executed entirely on a single processor but they are allowed to migrate on job boundaries – with different jobs of the same task allowed to execute on different processors. Each job's runtime context needs to be maintained only on a single processor. However, the task-level context is allowed to be migrated. In [2], for a restricted migration model, where migration is allowed only at job boundaries, Anderson *et al.* present an EDF-based partitioning scheme and scheduling algorithm that ensures bounded tardiness.

The *full migration* algorithms are the global algorithms that place no restriction upon inter-processor migration. The key idea of global scheduling is to maintain a single ready queue that has all the tasks released in the system which are eligible to be scheduled. The global queue is maintained according to some scheduling discipline and tasks are dispatched from this queue to any free processor.

Most of the global scheduling algorithms default to a single processor optimal algorithm such as Earliest Deadline First (EDF), or Rate Monotonic (RMS) algorithm. G-EDF [15] (*Global Earliest Deadline First*) orders the global queue based on the earliest deadline first order and the tasks are assigned to processors accordingly. However, once a scheduling event is triggered on one processor, a new schedule is created for which all tasks executing on other processors need to be preempted. In [10], Baruah derives the feasibility conditions for the non-preemptive version of G-EDF called the G-NP-EDF (*Global non-preemptive EDF*). In [32], Devi *et al.* derive the tardiness bounds for G-EDF and G-NP-EDF.

In [11, 12] Baruah *et al.* present a different global scheduling approach, called PFair [11] (*Proportionate Fair*) scheduling, which divides a task into quantum sized “sub-tasks” that are treated as the scheduleable entities. Each “sub-task” must execute within a “window” of time slots, where the last “window” is the deadline of the task. This approach has been shown to realize full system utilization and hence Pfair-based algorithms are optimal during underloads ($U \leq m$) [11, 12]. A particularly efficient version of Pfair scheduling, PD$^2$, was developed in [3] and its scalability was showcased in [16]. However, Pfair algorithms incur significant scheduling overheads due to their quantum-based scheduling approach as shown in [32]. Although, Pfair is optimal during underloads, its behavior during overloads follows an EDF-like pattern. This is because Pfair continues to use the notion of earliest deadline, even at the “sub-task” level. In [25] Cho *et al.* describe the LLREF scheduling algorithm. LLREF is a optimal real-time scheduling algorithm for multiprocessors which is not based on time quanta.

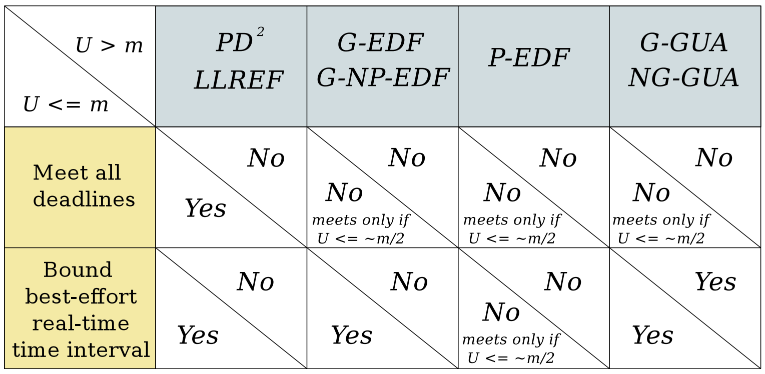
## 2.2 Classification Based on Utilization Load

Scheduling algorithms can also be classified based on those that assume the total utilization $U \leq m$, where $m$ is the number of CPUs and those that assume $U$ to be arbitrary. The former condition is referred to as *underload* while the later is referred to as *overload*. For multiprocessor systems, it has been shown that partitioned and non-Pfair global scheduling have similar schedulability bounds [18] and that these bounds are significantly below full system utilization. Algorithms that are not specifically designed to address overloads can suffer an unacceptable amount of degradation when they occur [13, 48]. What is needed is a set of algorithms that are designed with overload scheduling specifically in mind. This would allow these algorithms to successfully schedule as many high importance tasks as possible during overloads.

On single processor systems, Utility Accrual algorithms, such as DASA [31] and LBESA [55], that maximize accrued utility for downward “step” TUFs default to EDF during underloads, since EDF satisfies all deadlines during underloads, consequently maximizing possible accrued utility. During overloads, they favor more important activities, irrespective of urgency. Thus, deadline scheduling’s optimal timeliness behavior is a special case of utility accrual scheduling for single processor systems.

For a multiprocessor system, gMUA [22] and MOCA [48] scheduling algorithms provide best-effort utility accrual during overloads. gMUA defaults to global-EDF, a non-optimal scheduling algorithm, during underloads. During overloads, gMUA tries to maximize accrued utility. MOCA, on the other hand, divides the processors into “bands” where each band is assigned a certain utility density. MOCA attempts to schedule an incoming task on a band that corresponds to its utility. If it is not possible to execute the task on its utility band, MOCA moves the task down the bands until one is found that can schedule it. As a result of this design, if a large number of low utility tasks arrive, they will quickly exhaust the lower bands of the system and this scenario may cause incoming low utility tasks to be rejected even though bands associated with higher utilities are idle.

Both gMUA and MOCA suffer from lack of support for resources, such as locks, and their behavior in the presence of dependencies has not been studied.



| $U > m$ / $U <= m$ | $PD^2$ LLREF | G-EDF G-NP-EDF | P-EDF | G-GUA NG-GUA |
|---|---|---|---|---|
| Meet all deadlines | No / Yes | No / No meets only if $U <= \sim m/2$ | No / No meets only if $U <= \sim m/2$ | No / No meets only if $U <= \sim m/2$ |
| Bound best-effort real-time time interval | No / Yes | No / Yes | No / No meets only if $U <= \sim m/2$ | Yes / Yes |

Figure 2.1: Characterization of scheduling algorithms during underloads ($U \leq m$) and overloads ($U > m$) based on ability to meet all deadlines and bounded best-effort real-time time interval.

## 2.3 Summary

Figure 2.1 gives a summary of the classification discussed in this chapter. We observe the following:

1. During underloads (for $U \leq m$) only Pfair (PD$^2$) and LLREF scheduling algorithms can meet all deadlines. This is primarily because these algorithms have been proven to be optimal during underloads.

2. During underloads, G-EDF, G-NP-EDF and P-EDF scheduling algorithms can only meet their deadlines upto $U \approx m/2$.

3. During underloads, an optimal best-effort behavior can be seen in Pfair(PD$^2$) and LLREF. This is due to the fact that they are optimal during underloads.

4. During overloads, none of the scheduling algorithm can meet their deadlines.

5. During overloads, none of the scheduling algorithms provide a best-effort bound on the real-time interval.

6. There are no scheduling algorithms that provide best-effort behavior during overloads in the presence of dependencies.

There is a clear gap in the past work, that establishes a need for best-effort utility accrual real-time scheduling algorithms for multiprocessors, that provides best-effort behavior during overloads for dependent tasks. We bridge this gap by designing the NG-GUA and G-GUA scheduling algorithms which we discuss in detail in the subsequent chapters.

# Chapter 3

# Models and Objective

## 3.1 Thread Model

We consider the programming model where each task, $T_i$, can have a periodic or an aperiodic execution. Each such invocation of the task, $T_i$, is referred to as phase $J_i$. Each phase has an estimated best-faith[1] execution time $e_i$ and a deadline $d_i$. No specific task arrival pattern is assumed.

Tasks can be implemented using threads on an operating system. There are two models that can be used. A single thread can be created that represents the task and all its phases. At the end of the task's period, the thread goes to sleep and wakes up at the next period. Hence, each phase of the task can be executed on the same thread with a call to a function such as `sleep_until_next_period()` between phase invocations. We call this the "thread-is-a-task" model. On the other hand, an individual thread could represent each phase of a task. Hence, at the invocation of each phase of a given task, a separate thread is fired. We call this the "thread-is-a-phase" model.

## 3.2 Timeliness Model

We specify the time constraint of each task using a Time/Utility Function (TUF) [43]. A TUF allows us to decouple the urgency of a task from its importance. This decoupling is a key property allowed by TUFs since the urgency of a phase may be orthogonal to its importance. A task $T_i$'s TUF is denoted as $U_i\left(t\right)$. A classical deadline is unit-valued—i.e., $U_i(t) = \{1, 0\}$, since importance is not considered. Downward step TUFs generalize classical deadlines where $U_i(t) = \{\{m\}, 0\}$. We focus on downward step TUFs, and denote the maximum, constant utility of a TUF $U_i\left(t\right)$, simply as $U_i$.

[1]This is an estimate of execution time, not an upper bound, and may be violated at runtime

Each TUF has an initial time $I_i$, which is the earliest time for which the TUF is defined, and a termination time $X_i$, which, for the special case of downward "step" TUFs we consider, is its discontinuity point. Equations 3.1a-b describe the mathematical representation of a downward "step" TUF.

$$U_i(t) > 0, \forall t \in [I_i, X_i], \forall i \quad (3.1a)$$

$$U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i \quad (3.1b)$$

## 3.3 Resource Model

A resource $R_j$ has a critical section length, $C_j$. The length of the critical section can be variable. Any phases $J_i$ can request a resource $R_j$ and enter critical sections by invoking APIs such as `ResRequest()`. Until the request for $R_j$ is granted, the phase is said to be `blocked` on the resource. Once $R_j$ is granted to $J_i$, we refer to $J_i$ as the `Owner` of the resource. $J_i$ invokes `ResRelease()`[2] to complete the critical section. There is no restriction on the number of phases that may request a resource. However, a phase can only be blocked on a single resource which is being owned by another phase. We refer to this as the single-unit resource model.

## 3.4 Processor Model

We consider a multiprocessor/multi-core architecture with $M$ cores/processors. In the rest of the thesis, cores and processors will be used interchangeably unless explicitly stated otherwise.

## 3.5 Abort Model

We consider the model in which any phase that has blown its deadline can be aborted by the system. The phase is sent an abort signal which is handled by the phase using an `abort_handler`. The handler is used by the phase to release any resources that it may have requested or owned.

[2]These sample APIs have been used for the sake of explanation of the concept

## 3.6 Scheduling Objective

As mentioned earlier, the problem of scheduling real-time tasks on multiprocessors that are subject to run-time uncertainties causing transient and permanent overloads, is NP-Hard.

Our primary objective is to design a class of polynomial-time heuristic[3] algorithms that provide best-effort utility accrual during overloads in the presence of dependencies (such as locks) on a multiprocessor system. The algorithms should yield optimal total utility when possible and best-effort timeliness behavior otherwise.

[3]Another approach of solving this problem is by designing approximate algorithms. We chose the heuristics route because it is easier to design a heuristic algorithm as compared to an approximation. Also, the insights provided by a heuristic algorithm can be useful in the design of approximate algorithms.

# Chapter 4

# The GUA Class of Real-Time Scheduling Algorithms

In this chapter we present the *Global Utility Accrual* (GUA) class of algorithms. We discuss two algorithms in detail — *Non-Greedy Global Utility Accrual* (NG-GUA) and *Greedy Global Utility Accrual* (G-GUA). The non-greedy variant defaults to *Global Earliest Deadline First* scheduling algorithm with *Priority Inheritance Protocol* (G-EDF-PIP) during underloads, while maximizing the total accrued utility during overloads. On the other hand, the greedy algorithm uses the concept of Global Value Density (GVD) to maximize accrued utility during both underloads as well as overloads.

## 4.1 Ensuring Mutual Exclusion

When resources, such as locks, are used, it is important to ensure mutual exclusion. Consider Figure 4.1 which shows a dependency chain consisting of nine phases and three resources. Phases $T_1$ and $T_2$ require resource $R_1$, which is owned by $T_4$. Phases $T_4$ and $T_5$ require resource $R_2$, which is owned by phase $T_6$. Phase $T_3$ requires resource $R_3$, which is owned by phase $T_7$. Phases $T_8$ and $T_9$ do not require any resources and are independent. In order to ensure mutual exclusion, we can only schedule phases that do not have a dependency on other phases.

We construct a global precedence graph, which is a directed acyclic graph (DAG), of the phases with the nodes representing the phases and the edges representing the resources being requested. Figure 4.2 gives the graph representation of the dependency chain shown in Figure 4.1. Phase $T_6$ owns resource $R_2$, which is being requested by phases $T_4$ and $T_5$ and therefore they are dependent on phase $T_6$. This is represented in the graph as an edge $R_2$ from $T_6$ directed towards phases $T_4$ and $T_5$. In a similar fashion, phase $T_4$ currently owns resource $R_1$, which is being requested by phases $T_1$ and $T_2$. Hence, we see an edge from

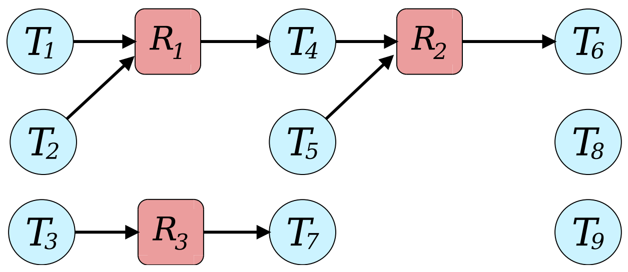


Figure 4.1: A phase and resource dependency chain with nine phases and three resources

phase $T_4$ towards phases $T_1$ and $T_2$. Similarly, we have an edge from phase $T_7$ to $T_3$, which requires resource $R_3$ that is being owned by phase $T_7$. Phases $T_8$ and $T_9$ do not have any resources requested and hence are represented as independent nodes.

All the nodes in the given DAG that do not have any edges from any other node are referred to as *zero in-degree* phases. In order to construct a schedule that respects mutual exclusion, we consider only the *zero in-degree* phases, as these represent phases that are not dependent on any other phase.

## 4.2 Maximizing Accrued Utility

In order to define a measure for the benefit that a given phases' completion can bring to the total accrued utility of the system, we use the concept of Potential Utility Density (or PUD), which was first defined by Clark in [31]. PUD is defined as the ratio of the remaining execution time of a phase at time $t$ to the utility of the phase defined by the TUF at time $t$, i.e., $U(t)$. The PUD of a phase is a dynamic value as it depends on the remaining execution time of a phase and hence at any given time it denotes the "return on investment" that can be gained for each unit of processing time assigned to that particular phase.

Clark defines and uses PUD in DASA [31] when considering phases that are dependent on each other. Clark considers the PUD of the entire group involved in a dependency relationship and uses the concept as an indicator of the urgency to execute a phase that blocks other phases; and also as a measure of the total benefit that the system will accrue if the entire dependency chain is executed while respecting mutual exclusion. However, as DASA is a uniprocessor scheduling algorithm, only a single phase can be executed at any given time $t$.

The problem with multiprocessor scheduling is, that it is possible for a scheduler to schedule a phase and one or more of its dependents on two or more processors concurrently. This does not benefit the total system utility accrual given the fact that the phase blocked on a resource is going to waste its CPU cycles. In order to prevent this, we use the DAG to find the set of *zero in-degree* phases that do not have any dependency relationship among each

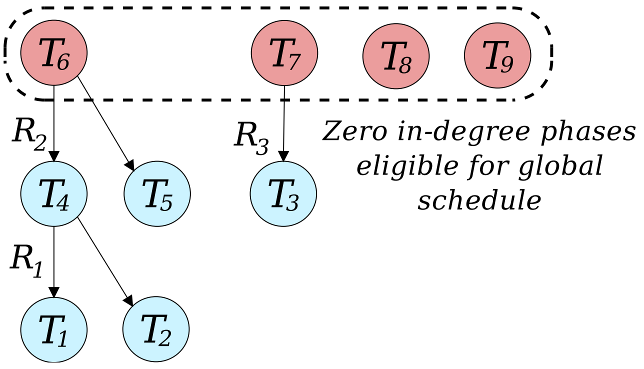


Figure 4.2: Directed Acyclic Graph (DAG) for the dependency chain shown in Fig 4.1. The nodes with a zero in-degree are eligible for final schedule in order to preserve mutual exclusion.

other and hence can be executed concurrently on any given $M$ processors (as discussed in Section 4.1).

However, given a set of $N$ *zero in-degree* phases, we require a method to select the $M$ phases that, when executed on the $M$ processors, would maximize the total accrued utility of the system. For this we define the term *Global Value Density* (or GVD). To avoid any confusion with the PUD of a phase, we define the term *Local Value Density* (or LVD) which is equal to the PUD of a given phase and is used instead of PUD in the rest of the thesis. GVD is computed only for the *zero in-degree* phases and it is defined as the sum of the LVD's of individual phases that are in a dependency relation with the given zero in-degree phase.

Consider Figure 4.3. We compute the GVD for phases $T_6$, $T_8$ and $T_9$ as these represent the current set of *zero in-degree* phases. The GVD for $T_6$ is computed as: `GVD(`$T_6$`) = GVD(`$T_4$`) + GVD(`$T_5$`) + LVD(`$T_6$`)`. This is equivalent to: `GVD(`$T_6$`) = LVD(`$T_6$`) + LVD(`$T_4$`) + LVD(`$T_5$`) + LVD(`$T_1$`) + LVD(`$T_2$`)`. The GVD for $T_8$ and $T_9$ is equivalent to their respective LVDs as these phases do not have dependents.

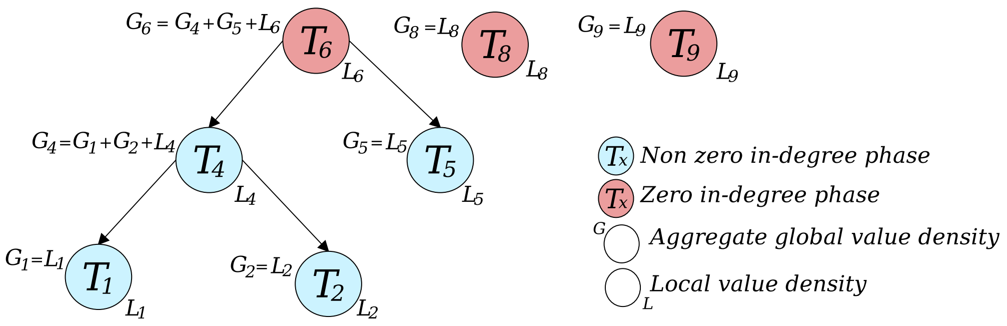


Figure 4.3: Computing the global value density for a graph

## 4.3 Deadlock Detection and Resolution

A deadlock can occur if a phase $T_1$, which owns a resource $R_1$, makes a request for another resource $R_2$, which is being owned by a phase currently blocked on a resource $R_i$ in the dependency chain of $T_1$. Figure 4.4 gives an example of a deadlock. Phase $T_4$ requires resource $R_2$ which is currently owned by phase $T_3$. Phase $T_3$ requires resource $R_3$ which is owned by phase $T_2$. However, phase $T_2$ is currently blocked on resource $R_1$ which is owned by $T_4$. Hence the deadlock.

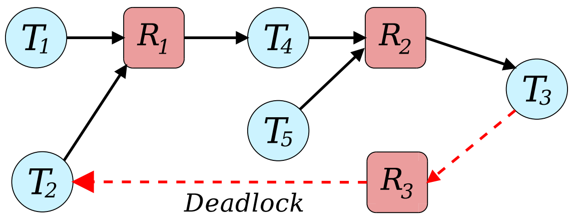


Figure 4.4: A phase and resource dependency chain with a deadlock condition.

A deadlock can be detected during the construction of the DAG. For the given set of phases that are ready for being scheduled, the construction of the DAG involves traversing the list of ready phases and creating a graph that establishes an edge relationship between nodes. For each phase $T_i$, we use API calls such as `ResRequested(`$T_i$`)` and `Owner(`$R_i$`)` to find the parent node in the DAG. We compute the relationship between all the dependent nodes in the chain. While the dependency chain is being computed, a list $\sigma_i$ is created with a reference to the nodes in the dependency chain of $T_i$ such that at every step in the construction of the DAG, we check $\sigma_i$ to verify if the node has already been inserted into the DAG. If the node is already in $\sigma_i$, it establishes the presence of a deadlock.

Fig 4.5 shows the process of DAG construction for the dependency chain of Figure 4.4. We see that there is deadlock between phases $T_6 \rightarrow T_2 \rightarrow T_4 \rightarrow T_6$. At this point we need to remove one of the phases from the deadlocked loop to resolve the deadlock. We remove the *Least Local Value Density* (or Least-LVD) phase amongst the phases that are currently in a deadlock. The rationale behind this approach is to ensure that we only abort a phase that provides the least utility to the total system accrued utility.

Aborting a phase involves marking the phase for immediate completion by sending it a signal. The phase can catch this signal and invoke its `abort_handler` to perform a cleanup; releasing resources such as locks.

## 4.4 Assigning Zero In-degree Phases to Processors

In order to assign the *zero in-degree* phases to the processors, we use the concept of the "least sum of total phase remaining execution time" on each processor, which was first described

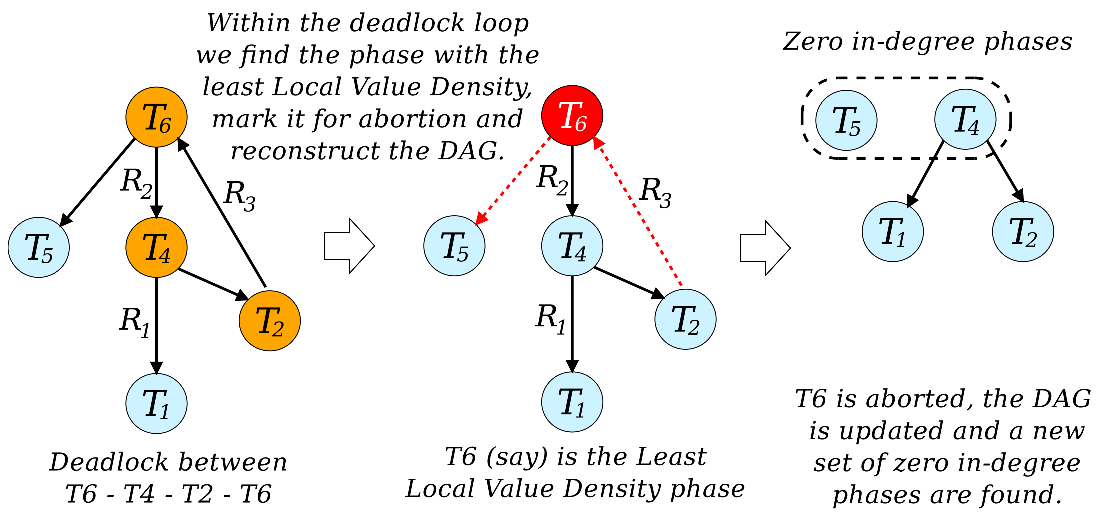

Within the deadlock loop we find the phase with the least Local Value Density, mark it for abortion and reconstruct the DAG.
Zero in-degree phases
$T_6$
$T_5$
$T_4$
$T_2$
$T_1$
$R_1$
$R_2$
$R_3$
Deadlock between T6 - T4 - T2 - T6
T6 (say) is the Least Local Value Density phase
T6 is aborted, the DAG is updated and a new set of zero in-degree phases are found.

Figure 4.5: Deadlock detection and resolution.

by Cho in [22].

Let us assume a set of *zero in-degree* phases along with their respective remaining execution times: $\{T_1 : 10\}, \{T_2 : 12\}, \{T_3 : 5\}, \{T_4 : 15\}$. Let us consider a multiprocessor system with $M = 2$. Initially, the sum of the remaining execution times for phases on each processor is set to zero: $\{p_1 : 0\}, \{p_2 : 0\}$. This is because no phase has been currently assigned to the processors. Now, $T_1$ can be assigned either to $p_1$ or $p_2$. Let us assume that $T_1$ is assigned to $p_1$. $T_1$'s remaining phase execution cost gets added to $p_1$. We have, $\{p_1 : 10\}, \{p_2 : 0\}$. $T_2$ gets assigned to $p_2$ and we get, $\{p_1 : 10\}, \{p_2 : 12\}$. At this point, $T_3$ needs to be assigned to the processors that has the least sum of total phase remaining execution time, which is $p_1$. The process continues until all the phases have been assigned to processors.

## 4.5 Data Structures and Auxiliary Functions

Before describing the algorithms in detail we list the member variables that can be added to the phase data structure to describe the timeliness model and the DAG representation of the resource model.

### 4.5.1 Data Structure for Timeliness Model

The following member variables can be used to define the timeliness model for each phase:

`J.ExecCost`
: The good faith execution cost of phase $J$.

`J.Deadline`
: The deadline of phase $J$.

`J.RemExec`
: The remaining execution time of a phase $J$. This represents the time that is left of the phase's execution cost. The value can be updated every time the phase is preempted by the scheduler or just before the new schedule is created. This value is used for the calculation of the Value Density for the phase.

`J.Utility`
: The utility of the phase $J$ as represented by its TUF.

`J.LocalValDen`
: The Local Value Density (LVD) of a phase $J$. The LVD is calcuated as $\frac{J.Utility}{J.RemExec}$.

### 4.5.2 Data Structure for Resource Model

The following member variables can be used to define the resource model for each phase represented as a DAG:

`J.AggUtil`
: The aggregate utility of a phase $J$. This is used for the calculation of the Global Value Density.

`J.AggExec`
: The aggregate remaining execution time of a phase $J$. This is used for the calculation of the Global Value Density.

`J.GlobalValDen`
: The Global Value Density (GVD) of a phase $J$. GVD is calculated as $\frac{\Sigma J.AggUtil}{\Sigma J.AggExec}$ for phase $J$ and all its dependents.

`J.PIPDeadLn`
: The PIP deadline of a zero in-degree phase $J$ is equal to the deadline of the phase in $J$'s dependency chain that has the earliest deadline. The PIP deadline is only calculated for zero in-degree phases. If a zero in-degree phase $J$ does not have any dependents, the PIP deadline is equal to `J.Deadline`.

`J.InDegree`
: The total number of links that come into a phase $J$ in the DAG. The in-degree of a phase $J$ can not be more than one.

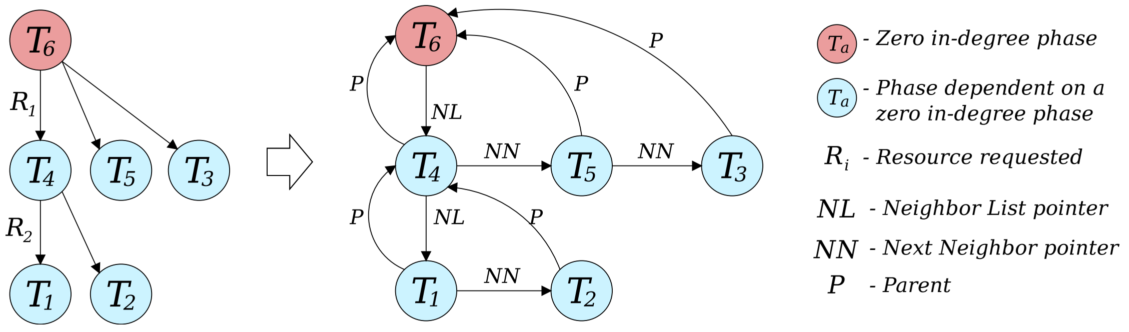
`J.OutDegree`
: The total number of links that go out of a phase in the DAG. A phase $J$ can have a value of zero or more for the out-degree.

`J.ResourceReq`
: The reference to the resource that has been requested by a phase $J$.

`J.NeighborList`
: The reference to the first dependent child node of a phase $J$ in the DAG.

`J.NextNeighbor`
: The reference to the next dependent child node of a phase $J$ in the DAG.

`J.Parent`
: The reference to the phase which owns the resource that the phase $J$ has requested. Figure 4.6 illustrates how the DAG can be represented using `J.NeighborList`, `J.NextNeighbor` and `J.Parent`.



$T_6$
$R_1$
$T_4$
$T_5$
$T_3$
$R_2$
$T_1$
$T_2$
P
NL
NN
$T_a$ - Zero in-degree phase
$T_a$ - Phase dependent on a zero in-degree phase
$R_i$ - Resource requested
NL - Neighbor List pointer
NN - Next Neighbor pointer
P - Parent

Figure 4.6: Data structure representation of the DAG

## 4.5.3 Auxiliary Functions

In this section, we define the auxiliary functions that are used by the NG-GUA and G-GUA scheduling algorithms. These methods perform small functions which are implementation independent.

`InsertList`$(J, \sigma)$
: Insert the phase $J$ into the list $\sigma$.

`RemoveList`$(U, \sigma)$
: Remove phase $J$ from the list $\sigma$.

`ComputeGVD(`$\sigma_{in}$`)`
: For the given list of zero in-degree phases in $\sigma_{in}$, compute the global value density, which is calculated as $\frac{\Sigma J.AggUtil}{\Sigma J.AggExec}$ for phase $J$ and its dependents represented as child nodes of the zero in-degree phase in the graph.

`SortByGVD(`$\sigma_{in}$`)`
: Sort the list $\sigma_{in}$ by the decreasing global value density and return the new ordered list in $\sigma_{out}$.

`SortByDeadLn(`$\sigma_{in}$`)`
: Sort the list $\sigma_{in}$ by the decreasing deadline (EDF order) and return the new ordered list in $\sigma_{out}$.

`HeadOf(`$\sigma$`)`
: Return the phase $J$ which is at the head of the list $\sigma$.

`InsDeadLnPos(`$J, \sigma$`)`
: Insert phase $J$ in the list $\sigma$ at its deadline position.

`FindZIDPhases(`$\sigma_t$`)`
: Return the list $\sigma_z$ of zero in-degree phases in $\sigma_t$.

`FindPIPDeadLn(`$\sigma_z$`)`
: For each zero in-degree phase $J \in \sigma_z$, assign the PIP deadline for $J$ equal to the deadline of the phase which has the earliest deadline amongst the dependents of $J$. If $J$ does not have any dependents, the PIP deadline of $J$ equals its deadline.

`RemoveLeastLVD(`$\sigma$`)`
: Remove the phase with the least LVD from $\sigma$.

`RemoveLeastGVD(`$\sigma$`)`
: Remove the phase with the least GVD from $\sigma$.

`IsPresent(`$J, \sigma$`)`
: Return `true` if the phase $J$ is present in list $\sigma$, else return `false`.

`IsFeasible(`$\sigma$`)`
: Return `true` if the given schedule in list $\sigma$ is feasible. For $\sigma$ to be feasible, the predicted completion time of each phase in $\sigma$ must never exceed its deadline.

`IsEmpty(`$\sigma$`)`
: Return `true` if the list $\sigma$ is empty, else return `false`.

`InsertEdge(`$J, DepJ$`)`
: Insert an edge between $DepJ$ and $J$.

`RemoveEdge(`$J$`)`
: Remove all in-degree and out-degree edges of $J$.

`Owner(`$R$`)`
: Return the phase $J$ that holds resource $R$. If there is no phase that holds resource $R$, return `NULL`.

`ResRequested(`$J$`)`
: Return the resource $R$ requested by phase $J$.

`FindProcessor()`
: Return the processor $p$ which has the least sum of total phase execution cost.

`FindProcessor(cpu_mask)`
: This version of `FindProcessor()` takes an argument called `cpu_mask`, which is a list of processors that have been checked before and need to be masked (avoided) while selecting the processor with the least sum of total phase execution cost. If all the processors on the system are included in the mask, the function returns `NULL`.

`AddCpuToMask(p, cpu_mask)`
: Add processor $p$ to the `cpu_mask`. The `cpu_mask` can be a bit array where each bit corresponds to each online processor, in which case, set the bit corresponding to $p$.

`FindLeastLVD(`$J$`)`
: For the given phase $J$'s dependency chain, find the phase with the least Local Value Density (LVD).

`UpdateCpuEC(`$p, J, b$`)`
: If $b$ is true, add the execution cost of $J$ to the total phase execution count for processor $p$, else subtract the execution cost of $J$ from the total phase execution count.

`AbortPhase(`$J$`)`
: Send an aborting signal to the phase $J$.

`IsPhaseAborted(`$J$`)`
: Return `true` if the phase $J$ has been marked to be aborted.

### 4.5.4 Creation of DAG with Detection/Resolution of Deadlocks

In this section, we give the details of the auxiliary function used to create the DAG along with detection and resolution of deadlocks. In Algorithm 1, we describe the pseudo-code for `CreateDAGwithDRD(`$\sigma_T$`)` that uses the list of phases, $\sigma_T$, which are ready and eligible to be scheduled. For the rest of the thesis, we will refer to the phase that has requested a resource as a *child* while the phase that owns the resource being requested as a *parent*.

**Algorithm 1**: Creation of DAG with detection/resolution of deadlocks

```
1: Procedure: CreateDAGwithDRD ($\sigma_T$)
2:
3: Input: $\sigma_T$            // List of released phases
4: Vars: $S, J, V, next$        // Phase pointers
5: Vars: $\sigma_J$             // Phase $J$'s list of dependents
6:
7: $next = \phi$;
8: for each phase $J$ in $\sigma_T$ do
9:     $\sigma_J = \phi$ ;
10:    $J.LocalValDen = \frac{J.utility}{J.RemExec}$;
11:
12:    InsertList($J, \sigma_J$);
13:    $next$ = Owner(ResRequested($J$));
14:    while $next \neq \phi$ do
15:        InsertEdge($J, next$) ;
16:        if IsPhaseAborted($next$) then
17:            break;
18:        if IsPresent($next, \sigma_J$) = false then
19:            InsertList($next, \sigma_J$);
20:            $J = next$;
21:            $next$ = Owner(ResRequested($next$));
22:        else
23:            $V$ = FindLeastLVD($next$);
24:            AbortPhase($V$);
25:            RemoveEdge($V$);
26:            break;
```

In lines 8-26, the algorithm iterates over the list, $\sigma_T$, and for each phase, $J$, checks if a *parent* node exists. If there exists a *parent* (line 13), the algorithm adds an edge from the *parent* to the *child* node (line 15). The main objective of the algorithm is to construct the complete dependency chain for a given phase. In lines 20-21, the algorithm sets the current *parent* node as the new *child* and checks if it has requested a resource. The steps are repeated for all the phases in the dependency chain of $J$.

To perform deadlock detection, for each phase, $J$, whose dependency chain is being analyzed, we create a temporary list $\sigma_J$ (line 9) and insert all the dependencies for phase $J$ into $\sigma_J$ (line 12). Before adding an edge between a *child* and a *parent*, we first check if the *parent* node is already present in $\sigma_J$ (line 18). The existence of the phase in $\sigma_J$ means that the phase has already been added to the graph. This implies that we have detected a circular dependency and hence a deadlock. To resolve the deadlock, we need to find the least local value density phase in $\sigma_J$, abort the phase and remove it from the graph (lines 23-26).

Once the DAG has been created, we need to compute the Global Value Density for the *zero in-degree* phases. Finding the *zero in-degree* phases involves iterating over the list of released phases $\sigma_T$ and creating another list $\sigma_z$ of phases which have `J.InDegree` equal to zero. For each of the *zero in-degree* phases, the global value density (or GVD) needs to be computed. The GVD of a *zero in-degree* phase $J$ is given as $\frac{J.AggUtility}{J.AggRemExec}$ where `J.AggUtility` is the aggregate utility of $J$ and all its child nodes in the graph and `J.AggRemExec` is the aggregate remaining execution time of $J$ and all its child nodes in the graph. The `ComputeGVD(`$\sigma$`)` auxiliary function describes one of the methods of computing GVD. As an optimization, the computation of the aggregate utility and the aggregate remaining execution time for *zero in-degree* phases can be coupled with the `InsertEdge()` or the `FindZIDPhases()` auxiliary functions.

## 4.6 Non-greedy Global Utility Accrual (NG-GUA)

Algorithm 2 describes the pseudo-code for the Non-Greedy Global Utility Accrual (NG-GUA) algorithm. The pseudo-code uses auxiliary functions that have been discussed earlier in Section 4.5.3.

In line 5, for a given list of phases $\sigma_T$ that are ready to be scheduled, we compute the global precedence graph (DAG) and detect/resolve deadlocks. This is done by calling `ComputeDAGwithDRD()`.

In line 6, after the DAG has been created, we find the list $\sigma_z$ of all the zero in-degree phases. A zero in-degree phase does not depend on any other phase in the system.

In line 7, for all the phases in $\sigma_z$, we compute the global value density. This is done using the `ComputeGVD()` function.

In order to ensure that NG-GUA defaults to G-EDF-PIP behavior (in the presence of dependencies), we compute the PIP deadlines for each of the zero in-degree phases. The PIP deadline of a zero in-degree phase $J_z$ can be found as the earliest deadline of a phase $J_i$ which is dependent on the given zero in-degree phase $J_z$ in the graph. In the absence of dependencies, the PIP deadline of $J_z$ is the same as the given deadline of $J_z$.

In lines 8-9, we compute the PIP deadline for all the zero in-degree phases and sort them based on those deadlines. The sorted list is stored in $\sigma_d$. The key idea here is to sort the zero in-degree phases by the deadlines of the phases which have the earliest deadline but are currently blocked on a resource that is being held by the zero in-degree phases.

In lines 11-13, we use the method defined in Section 4.4 to find the processor with the least sum of total remaining execution cost and start assigning the phases to the individual processor lists $\sigma_p$.

In lines 14-16, we check the individual processors lists $\sigma_p$ for schedule feasibility. This is done using the `IsFeasible()` function which returns true, if the given schedule in list $\sigma_p$ is feasible. For a schedule to be *feasible*, the predicted completion time for each phase in $\sigma_p$

**Algorithm 2**: NG-GUA: Non-greedy Global Utility Accrual

```
1: Input: $\sigma_T$          // List of released phases
2: Vars: $\sigma_1 \cdots \sigma_M$  // Per processor ready queues for M processors
3: Vars: $\sigma_z$           // Zero in-degree phase list
4:
5: ComputeDAGwithDRD($\sigma_T$);
6: $\sigma_z \leftarrow$ FindZIDPhases($\sigma_T$);
7: ComputeGVD($\sigma_z$);
8: $\sigma_z \leftarrow$ FindPIPDeadLn($\sigma_z$);
9: $\sigma_d \leftarrow$ SortByPIPDeadLn($\sigma_z$);
10:
11: for each phase $J$ in $\sigma_d$ do
12:     $p \leftarrow$ FindProcessor();
13:     InsertList(J, $\sigma_p$);
14: for each processor $p$ do
15:     while IsFeasible($\sigma_p$) = false do
16:         RemoveLeastGVD($\sigma_p$);
17: for each p processor's schedule $\sigma_p$ in $M$ do
18:     $Job_p \leftarrow$ HeadOf($\sigma_p$) ;
19: return { $Job_1, \cdots, Job_M$ };
```

must never exceed its deadline.

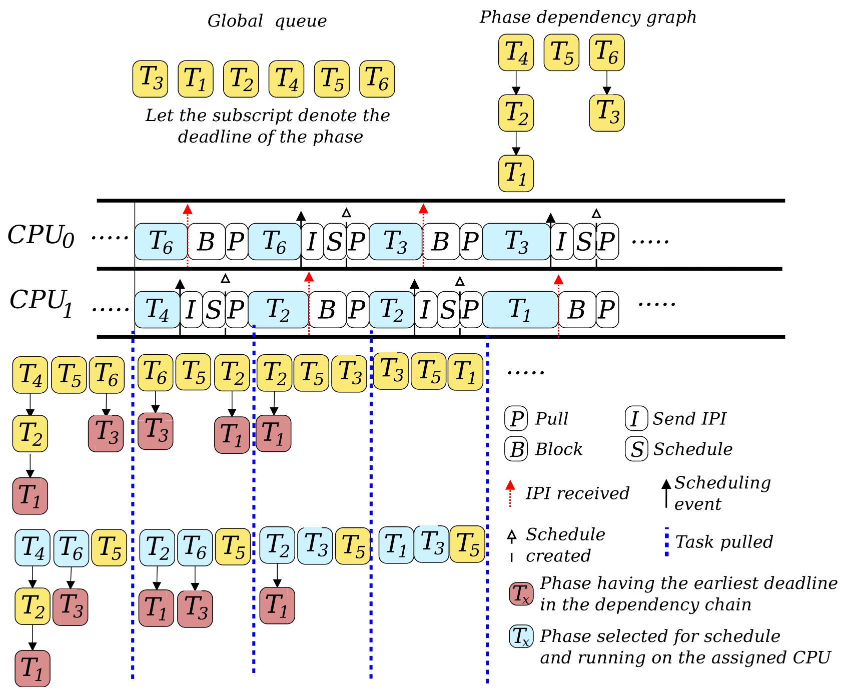
If the schedule is not feasible, the system is in an overload with $U > M$. During overloads, we need to attempt to maximize the total utility of the system by allowing phases that have a higher value density to be executed. In line 16, we find the phase in $\sigma_p$ that has the least GVD, remove it from $\sigma_p$ and check the schedule again for feasibility. Lines 15-16 are executed until we find a feasible schedule.

The reason for using GVD while removing the phase during an infeasible schedule is to ensure that we respect the dependency chain. It is possible that the zero in-degree phase having the highest GVD, has the least LVD. In such a case, using LVD to remove a phase would not be a correct representation of the dependency chain and hence would degrade the overall performance during overloads.

In lines 17-19, the head of the final feasible schedule $\sigma_p$ for each processor $p$ is dispatched on that processor.

Algorithm 2 is referred to as non-greedy because it defaults to a deadline order rather than a value density order along with support for priority inheritance protocol, thus following a G-EDF-PIP behavior during underloads and maximizing total accrued utility during overloads.

Figure 4.7 shows a sample schedule for NG-GUA. $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, and $T_6$ represent real-time phases in the global queue. We assume that the subscripts on each phase represent the relative deadlines. So, $T_1$ has an earlier deadline than $T_6$. The precedence graph shows the



Global queue
Let the subscript denote the deadline of the phase
Phase dependency graph
CPU0
CPU1
P Pull
I Send IPI
B Block
S Schedule
IPI received
Scheduling event
Schedule created
Task pulled
Phase having the earliest deadline in the dependency chain
Phase selected for schedule and running on the assigned CPU

Figure 4.7: Sample schedule for NG-GUA

relationship of phases with each other. $T_4$, $T_5$, and $T_6$ represent the zero in-degree phases. We find the PIP deadlines for these phases. For zero in-degree phase $T_4$, $T_1$ has the earliest deadline. Similarly, for phase $T_6$, $T_3$ has the earliest deadline. We sort the phases based on the PIP deadlines. Assuming the system is underloaded, the assignment of all the phases to the processors would result in a feasible schedule. $T_4$ and $T_6$ are the most eligible phases and they are assigned to processors CPU$_0$ and CPU$_1$, respectively.

Let us assume that $T_4$ finishes before $T_6$. As a result, a scheduling event is generated on CPU$_1$, which sends an `IPI` to all the processors in the system and enters the scheduler[1]. CPU$_0$ receives the `IPI` and enters the scheduler. However, as CPU$_1$ is already in the scheduler, CPU$_0$ blocks. CPU$_1$ prepares the global schedule. $T_6$, $T_5$, and $T_2$ represent the zero in-degree phases (as $T_4$ has finished execution). We find the PIP deadlines and sort the phases based on those deadlines. $T_2$ and $T_6$ are now the most eligible phases and they are assigned to the processors. As $T_6$ was running on CPU$_0$ before the scheduling event was generated, we assign $T_6$ to CPU$_0$ to preserve cache-coherence. The assigned tasks are mapped to the processors. The processors pull the assigned tasks and start executing them.

During overloads, after the phases have been sorted based on PIP deadlines, they are assigned to the processors with the least sum of remaining phase execution cost. For each of the processors, schedule feasibility is checked. During overloads, some (or all) of the processors might show infeasible schedules. At that time, for each of processors which have an infeasible

[1]The details on the `IPI` are discussed in detail in Chapter 5

schedule, we remove the phase with the least GVD (and continue doing this till the schedule is feasible). If there are no dependencies, all the phases are treated as zero in-degree and the PIP deadlines for all those phases are equivalent to the actual deadlines of the phases. Hence, the same mechanism follows.

## 4.7 Greedy Global Utility Accrual (G-GUA)

Algorithm 3 describes the pseudo-code for the Greedy Global Utility Accrual (G-GUA) scheduling algorithm. The pseudo-code uses auxiliary functions that have been discussed earlier in Section 4.5.3.

Lines 8-10 are similar to the NG-GUA algorithm, described in Section 4.6. We compute the DAG, find the list of zero in-degree phases and compute the GVD for all the zero in-degree phases in $\sigma_z$. As G-GUA does not default to G-EDF-PIP, we do not need to find the PIP deadlines.

G-GUA differs from NG-GUA in two ways — (i) the zero in-degree phases are sorted by GVD instead of the PIP deadlines (line 11), and (ii) instead of assigning phases to all the processors and then running the feasibility check, G-GUA follows a much more greedier approach to accrue total utility.

For all individual GVD sorted zero in-degree phases in $\sigma_d$ (lines 13-30), G-GUA assigns the phase to a processor which has the least sum of total remaining execution cost and checks for feasibility of schedule on that processor. If the schedule is feasible, only then does G-GUA move to the next phase in $\sigma_d$. However, if the schedule is not feasible after the phase was added to the first processor it was assigned to, G-GUA removes it from that processor and tries the same phase on all the other available processors. This is done to ensure that a phase is tried on all processors before being rejected. This behavior is greedy as it tries to ensure that a high GVD phase is not rejected and thus provides greater accrued utility.

G-GUA uses two variables called `cpu_mask` and `not_fes`. These are used to make sure that a phase which was not feasible on the first processor it was assigned to, is tried on all the available processors. For every phase, the `cpu_mask` is initially set to zero (line 14). If a phase is not feasible on a processor, it is removed from that processor (line 25) and the processor is added to the `cpu_mask` (line 28). The `cpu_mask` is used by `FindProcessor()` to find the processors that have the least sum of total remaining execution cost avoiding (masking) the processors which have been already checked (line 18).

In lines 24-30, G-GUA checks if the current phase added to a processor makes a feasible schedule. If it does, G-GUA moves to the next phase by setting the `not_fes` flag to `false`. If the schedule is not feasible, lines 25-28 remove the phase from the processor and add the processor to the `cpu_mask`. The `not_fes` continues to be true, due to which the loop 17-30 is activated again. The `FindProcessor` is called with the `cpu_mask` which either returns a new

**Algorithm 3**: G-GUA: Greedy Global Utility Accrual

```
1:  Input: $T$            // List of released phases
2:  Vars: $\sigma_1 \cdots \sigma_M$  // Per processor ready queues for M processors
3:  Vars: $\sigma_z$           // Zero in-degree phase list
4:  Vars: cpu_mask      // Mask of all processors on which feasibility check of a given phase failed
5:  Vars: not_fes  // Flag to check if the feasibility check failed for a given phase
6:                 // on last processor it was assigned to
7:
8:  ComputeDAGwithDRD($T$);
9:  $\sigma_z \leftarrow$ FindZIDPhases($T$);
10: ComputeGVD($\sigma_z$);
11: $\sigma_d \leftarrow$ SortByGVD($\sigma_z$);
12:
13: for each phase $J$ in $\sigma_d$ do
14:     cpu_mask = 0;
15:     not_fes = true;
16:
17:     while not_fes == true do
18:         $p \leftarrow$ FindProcessor();
19:         if $p$ == NULL then
20:             break;
21:         InsDeadLnPos(J, $\sigma_p$);
22:         UpdateCpuEC(p, J, true);
23:
24:         if IsFeasible($\sigma_p$) == false then
25:             RemoveList(J, $\sigma_p$);
26:             UpdateCpuEC(p, J, false);
27:             not_fes = true;
28:             AddCpuToMask(p, cpu_mask);
29:         else
30:             not_fes = false;
31: for each p processor's schedule $\sigma_p$ in $M$ do
32:     $Job_p \leftarrow$ HeadOf($\sigma_p$) ;
33: return { $Job_1, \cdots, Job_m$ };
```

processor (line 18), or returns NULL if the phase has been tried on all the processors (lines 19-20). In that case, the phase is rejected and the next phase is considered in line 13.

In lines 31-32, the head of the final feasible schedule ($\sigma_p$) for each processor $p$ is taken and dispatched to the individual processor for scheduling.

Figure 4.8 shows a sample schedule for G-GUA. $T_1$, $T_2$, $T_3$, $T_4$, $T_5$, $T_6$, $T_7$, $T_8$, and $T_9$ represent real-time phases in the global queue. We assume that the subscripts on each phase represent the relative deadlines while as the superscript represents the LVD. The precedence graph shows the relationship of phases with each other. $T_4$, $T_7$, and $T_6$ represent the zero in-degree phases. For all of the zero in-degree phases we compute the GVD. We find that

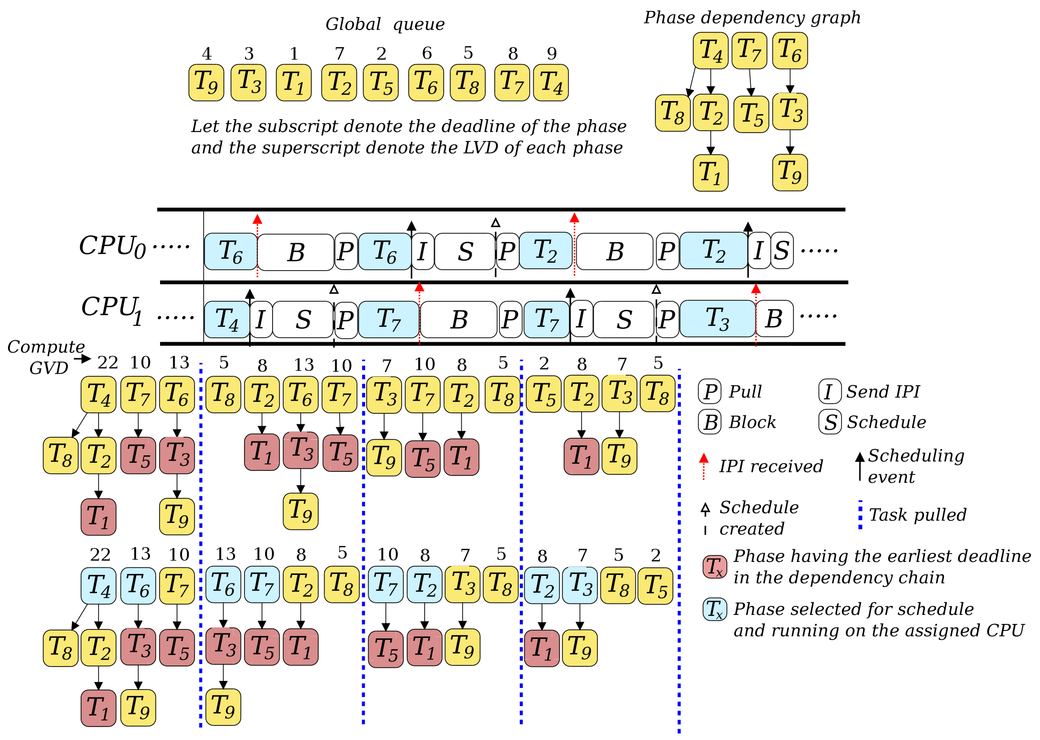

Global queue
Phase dependency graph
Let the subscript denote the deadline of the phase and the superscript denote the LVD of each phase
CPU0
CPU1
Compute GVD
P Pull
I Send IPI
B Block
S Schedule
IPI received
Scheduling event
Schedule created
Task pulled
Phase having the earliest deadline in the dependency chain
Phase selected for schedule and running on the assigned CPU

Figure 4.8: Sample schedule for G-GUA

$T_4$ has the highest GVD of 22 and $T_7$ has the least GVD of 10. We sort the phases based on the decreasing values of GVD. At this point, we take each phase, assign it to a processor with the least sum of total phase remaining execution cost and check for schedule feasibility. During underloads, the schedules would be feasible on all processors. As a result, $T_4$ and $T_6$ are found to be the most eligible phases and assigned to processors CPU$_0$ and CPU$_1$, respectively.

Let us assume that $T_4$ finishes before $T_6$. As a result, a scheduling event is generated on CPU$_1$, which sends an `IPI` to all the processors in the system and enters the scheduler[2]. CPU$_0$ receives the `IPI` and enters the scheduler. However, as CPU$_1$ is already in the scheduler, CPU$_0$ blocks. CPU$_1$ prepares the global schedule. As phase $T_4$ has finished execution, $T_8$, $T_2$, $T_6$, and $T_7$ represent the new zero in-degree phases. We compute the GVD for the phases. $T_8$ has the least GVD. We follow the same mechanism as discussed earlier and find that $T_6$ and $T_7$ are the most eligible phases.

During overloads, while the phases are being assigned to processors with the least sum of phase remaining execution cost, addition of a phase might result in an infeasible schedule on a given processor. At that point, the phase is tried on the other processors before being rejected. This ensures greedy behavior. If there are no dependencies, all the phases are

[2]The details on the `IPI` are discussed in detail in Chapter 5

treated as zero in-degree and the GVD for all those phases is equivalent to their LVD. Hence, the same mechanism follows.

## 4.8 Algorithm Properties

From now onwards in this thesis, "underload" is defined to exist when thread utilization demand satisfies one of the G-EDF schedulability conditions in [15], while "overload" is defined to exist when thread utilization demand exceeds the global EDF schedulability conditions in [15][3].

We define some of the properties of G-GUA and NG-GUA.

**Theorem 1.** *NG-GUA, with no dependencies, defaults to global-EDF scheduling during underloads.*

*Proof.* Global-EDF sorts the ready phases in deadline order and then dispatches the head of this queue to available processors. NG-GUA mimics this behavior by first sorting the ready phases in deadline order and then dispatching them to processors that have the least sum of remaining phase execution cost. The processor $p$ with the least sum of remaining execution cost would be the first processor to be available in the system. Therefore, the top of the ready queue would be assigned to processor $p$ in global-EDF and we assign the phase to $p$ in NG-GUA as well. Hence, NG-GUA mimics the scheduling semantics of global-EDF during underloads. □

**Theorem 2.** *NG-GUA, with dependencies, defaults to global-EDF with PIP scheduling during underloads.*

*Proof.* Global-EDF with PIP sorts the ready phases in deadline order. It then checks if the phase $J_a$ at the head of the queue is blocked by another phase $J_b$. Instead of executing $J_a$, G-EDF does a priority inheritance by allowing $J_b$ to be executed instead. NG-GUA mimics this behavior by finding out the PIP deadline of all the zero in-degree phases (line 8) and sorting the zero in-degree phases based on the PIP deadlines (line 9). This ensures that the phase $J_b$ which is now at the head of the queue after the sort, is a zero in-degree phase which is blocking another phase $J_a$, that has the earliest deadline in the system. Hence NG-GUA mimics the scheduling semantics of global-EDF with PIP scheduling during underloads, in the presence of dependencies. □

**Theorem 3.** *The gMUA scheduling algorithm is a special case of NG-GUA scheduling algorithms*

[3]As EDF is optimal on a uniprocessor, underload is defined when task utilization $U \leq 1$ and an overload when $U > 1$. However, for multiprocessors, G-EDF is not optimal and can meet deadlines for $U \approx m/2$, for $m$ processors. Hence, underload is defined equivalent to the utilization bounds of G-EDF.

*Proof.* At every scheduling event, gMUA [22] takes the phases that have been released in the system and are ready for execution and sorts them based on their deadlines. gMUA assigns the deadline sorted phases to processor queues. Each phase is assigned to a processor which has the least sum to total remaining execution cost. Once the phases have been assigned, gMUA runs the schedule feasibility check on individual processors queues. If the phases assigned to a processor result in an infeasible schedule, gMUA removes the phase with the least value density, until the schedule is feasible. Finally, the head of each of the processor queue is considered as the final schedule.

This behavior is similar to NG-GUA without dependencies. In the absence of any dependency relationship between phases, line 5 of NG-GUA treats all the phases as *zero in-degree* nodes in the graph. Thus in line 6, all the phases that were initially in the ready queue are eligible for the final schedule. As there are no dependencies, the PIP deadline of each phase is equal to its own deadline. Hence, in line 9, the `SortByPIPDeadLn()` function sorts the phases based on their actual deadlines. In lines 11-13, NG-GUA uses `FindProcessor()` function to assign phases to processors that have the least sum to total remaining execution cost. In lines 14-16 NG-GUA, runs a schedule feasibility check on individual processor queues and in case of an infeasible schedule, removes the phases with the least GVD. However, as there are no dependencies, the GVD of a phase is equal to its LVD.

Thus, gMUA is a special case of NG-GUA.

□

**Theorem 4.** *During overloads, NG-GUA attempts to accrue as much utility as possible by attempting to maximize the utility accrued at each scheduling step.*

*Proof.* NG-GUA, in lines 8-9, first constructs a global deadline ordered schedule. Once the entire schedule is constructed, the algorithm goes over the system to check for feasibility (lines 14-16), and removes the least GVD phase if the schedule is not feasible. Thus, during overloads, when phases first begin to miss deadlines, NG-GUA begins to shed phases in inverse proportion to their utility density. This method allows the algorithm to gracefully degrade in the presence of overloads by eliminating the threads that would be least beneficial to the system in terms of accrued utility. The algorithm therefore attempts to minimize the loss in accrued utility, or, equivalently, to maximize the remaining accrued utility, at each scheduling step. □

**Theorem 5.** *Both algorithms ensure mutual exclusion.*

*Proof.* In order to ensure mutual exclusion, it should not be possible for two or more phases that have critical sections, which access the same variables, to execute at the same time. Phases that access the same variables in their critical sections protect these sections using the same locks. The algorithms detect this dependency while constructing the DAG. Since both algorithms only execute *zero in-degree* phases, they ensure that a phase and its dependents do not execute at the same time, thus ensuring that mutual exclusion is guaranteed. □

**Theorem 6.** *For both algorithms considered, an application always makes progress, i.e., executes application specific code, if there is work offered and the application is not deadlocked.*

*Proof.* Both algorithms are based on the concept of *zero in-degree* phases. Given a DAG, line 5 in NG-GUA and line 8 in G-GUA, select a set of phases that do not depend on any other phases. These phases are then sorted according to deadline or GVD order and dispatched to the appropriate processor. If the application still has work offered, and it is not deadlocked, there will always be at least one *zero in-degree* node in the DAG. The phase represented by this node is dispatched to one of the processors in the system, thus making progress in accordance with application logic. □

**Property 1.** *In [39], Theorem 16.3.1 shows that when the schedule length is used as a criteria, a greedy scheduling algorithm that schedules the zero in-degree nodes in a DAG produces a schedule that is within a factor of two from being optimal. Further, for a multi-threaded application with $P$ threads, work $T_1$ and critical path length $T_\infty$, the length of the schedule is bounded by*

$$\frac{T_1}{P_A} + \frac{T_\infty(P-1)}{P_A}$$

*where $P_A$ is defined as the average number of threads executed at each scheduling interval.*

**Theorem 7.** *Property 1 applies for both NG-GUA and G-GUA.*

*Proof.* The "greedy" nature discussed in [39] refers to the ability of an algorithm to schedule as many ready phases as possible. More specifically, an algorithm is considered "greedy" in the context of [39] if the number of threads scheduled at each step is the minimum of the number of available processors and the number of threads that are ready to be executed.

Both G-GUA and NG-GUA have this property, since they both select the *zero in-degree* nodes in a DAG and schedule the threads they represent on the available processors – note that these are the phases that are ready for immediate execution. The main difference between them and the general greedy algorithm discussed in [39] is that they sort the *zero in-degree* phases according to some real-time criteria (whether deadlines or GVD) and select the first $n$ phases from this ordered list for immediate dispatching (where $n$ is the number of available processors). Therefore, NG-GUA and G-GUA are in the class of algorithms discussed in [39] and the theorem follows. □

**Theorem 8.** *For $m$ processors and $n$ phases, the asymptotic cost of NG-GUA is $O(mn \log n)$.*

*Proof.* In line 5 we take the list of released phases and compute the dependency relationship for each phase, which has a cost of $O(n)$. In line 6 we find the zero in-degree phases from the DAG, which can be completed in $O(1)$. For computing the GVD, the worst case cost is

$O(n)$, assuming all the $n$ phases in the ready queue are zero in-degree. This is possible if there are no dependencies.

In line 8 we find the PIP deadline for all the zero in-degree phases, which has a cost of $O(n)$. In line 9 we sort the phases by the PIP deadline. The worst-case execution cost of the most efficient sorting methods is $O(n \log n)$. Lines 11-16 have a cost of $O(n)$, so with $m$ processors, the overall cost is $O(mn)$.

Hence, the total asymptotic cost of NG-GUA is: $O(n) + O(1) + O(n) + O(n) + O(n \log n) + O(mn) = O(mn \log n)$. □

**Theorem 9.** *For $m$ processors and $n$ phases, the asymptotic cost of G-GUA is $O(mn \log n)$.*

*Proof.* In line 8 we take the list of released phases and compute the dependency relationship for each phase, which has a cost of $O(n)$. In line 9 we find the zero in-degree phases from the DAG, which can be completed in $O(1)$. For computing the GVD, the worst case cost is $O(n)$, assuming all the $n$ phases in the ready queue are zero in-degree. This is possible if there are no dependencies.

In line 11 we sort the phases by the GVD. The worst-case execution cost of the most efficient sorting methods is $O(n \log n)$. Lines 17-30 can be repeated for $m$ processors, with a cost of $O(m)$. Lines 13-30 are computed for $n$ phases, with a total cost of $O(mn)$.

Hence, the total worst-case asymptotic cost of G-GUA is: $O(n) + O(1) + O(n) + O(n \log n) + O(mn) = O(mn \log n)$. □

# Chapter 5

# ChronOS Real-Time Linux

## 5.1 Background

In order to implement and evaluate our scheduling algorithms we developed ChronOS [1], which provides real-time extensions to the Linux kernel. ChronOS is derived from the 2.6.31.12 version of the Linux kernel and uses Ingo Molnar's `PREEMPT_RT` real-time patch [60] which enables complete preemption in Linux and improves interrupt latencies. The patch is released under GPLv2 [36] and makes it suitable for academic research. ChronOS provides a set of APIs and a scheduler plugin infrastructure that can be used to implement and evaluate various single-processor and multiprocessor scheduling algorithms.

The two main objectives of implementing ChronOS are — (i) to provide a first-class abstraction of Distributed Threads; and (ii) to provide a real-time scheduling framework. The Distributable Thread (also know as DT) programming abstraction first appeared in the Alpha OS [29] and subsequently in Mach 3.0 [35], MK7.3 [67], Real-Time CORBA 2.0 [61], and the emerging Distributed Real-Time Specification for Java (DRTSJ) [45]. A Distributable Thread is a single thread of execution with a globally unique identity that transparently extends and retracts through local and remote objects. A real-time Distributed Thread carries along its timing constraints (deadline, TUF, worst-case execution cost) as it makes remote invocations and travels through multiple nodes. On each node, the local instance of the Distributed Thread is scheduled with these timing constraints[1].

The rest of the chapter outlines the architecture of ChronOS and the modifications that we have made to the Linux kernel to achieve the real-time behavior. We primarily focus on the scheduling infrastructure concentrating on the design and implementation of multiprocessor scheduling algorithms.

[1]The DTs will not be discussed in detail as the topic is outside the scope of this thesis.

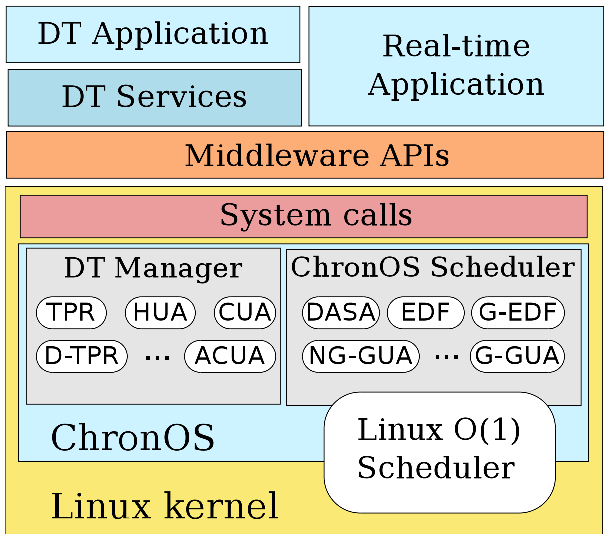


Figure 5.1: ChronOS architecture

## 5.2 Architecture

Figure 5.1 shows the overall architecture of ChronOS real-time Linux. ChronOS is derived from the Linux kernel and we rely on all the kernel primitives for basic operating system objectives such as process and memory management, timers, interrupts, drivers, file-system, and networking. As shown in Figure 5.1, Linux provides the basic foundation of ChronOS.

We extend the Linux `O(1)` scheduler and implement the ChronOS real-time scheduler where various single-processor scheduling algorithms, such as `EDF`, `DASA`, `LBESA`, `HVDF`, `RMA` and multiprocessor algorithms, such as `G-EDF`, `G-NP-EDF`, `G-FIFO`, `NG-GUA`, `G-GUA` have been implemented. At the same level, the Distributed Thread manager is implemented which provides first-class kernel level abstraction for Distributed Threads. Various Distributed Thread related algorithms, such as `TPR`, `D-TPR`, `HUA`, `CUA`, `ACUA` [33] have been implemented.

The Distributed Thread manager and the scheduling framework extensions are exposed to the user-space through system calls. ChronOS adds a set of new system calls to Linux that are used to enable the DT and scheduling extensions. At the user-space level, a middleware API abstraction layer is provided that maps the system calls to the user-space. ChronOS provides the application developer the freedom to write real-time applications directly using the middleware APIs without using any of the Distributed Thread services or to create a distributed applications using DTs without using any of the real-time functionality.

### 5.2.1 `PREEMPT_RT` Real-Time Patch

The stock Linux kernel provides soft real-time capabilities, such as the POSIX primitives and the ability to set priorities. The kernel provides two real-time scheduling policies - `SCHED_FIFO` and `SCHED_RR` and a "nice" based scheduling policy called `SCHED_NORMAL`. In `SCHED_FIFO` processes are given CPU for as long as they want, until the arrival of a higher priority task. This is primarily a POSIX-specified feature. `SCHED_RR`, on the other hand, schedules processes of the same priority in a round robin fashion. This is done while favoring higher priority tasks. The ChronOS real-time scheduler uses the `SCHED_FIFO` as the underlying base scheduling algorithm and builds up the scheduling framework on that. The scheduler is described in detail in the subsequent sections.

In order to bring in real-time semantics to the kernel, it is required to have a preemptable kernel. The stock Linux kernel does not allow "complete" kernel preemption. However, in order to implement scheduling algorithms such as `G-EDF` or `G-GUA`, it is necessary to be able to preempt the kernel. To achieve this we use the `PREEMPT_RT` patch. The patch enables complete kernel preemption along with a generic clock event layer with high resolution support, thus providing hard real-time capabilities in the Linux kernel.

With the `PREEMPT_RT` patch most parts of the kernel, except for a few small regions (which are inherently unpreemptible, such as the task scheduler), can be preempted. In order to achieve complete preemption the following changes were made to the Linux kernel.

1. All the in-kernel locking primitives, such as spinlocks have been re-implemented using rtmutexes. As a result, all critical sections that are protected with `spinlock_t` or `rwlock_t` are preemptable. However, the patch still enables the creation of non-preemptable sections in the kernel, if that is required.

2. Priority Inheritance has been implemented for all in-kernel spinlocks and semaphores.

3. All interrupt handlers in the kernel have been converted into kernel threads. All the soft interrupt handlers are treated in the kernel thread context (i.e., they have a `struct task_struct` associated) such that they can be treated as threads with higher priority and scheduled accordingly. However, it is still possible to register an IRQ in the kernel context.

4. The existing Linux timer APIs have been converted into separate infrastructure for high resolution kernel timers, including those for timeouts, which enable the use of user-space POSIX timers with high resolution.

Overall, the `PREEMPT_RT` patch improves the interrupt latencies and provides a completely preemptable kernel.

## 5.3 ChronOS Real-Time Scheduler

Since version 2.6, the Linux kernel features an `O(1)` scheduler [57]. Every scheduling algorithm provided in the default Linux kernel (`SCHED_NORMAL`, `SCHED_FIFO`, `SCHED_RR`) completes in constant-time, regardless of the number of processes in the system that are in the running state. The `O(1)` scheduler also implements SMP scalability where each processor has its own locking and individual run-queues. The scheduler also implements SMP affinity which enables processes to be assigned to a specific CPU.

### 5.3.1 Priority Bit-map and Run-queues

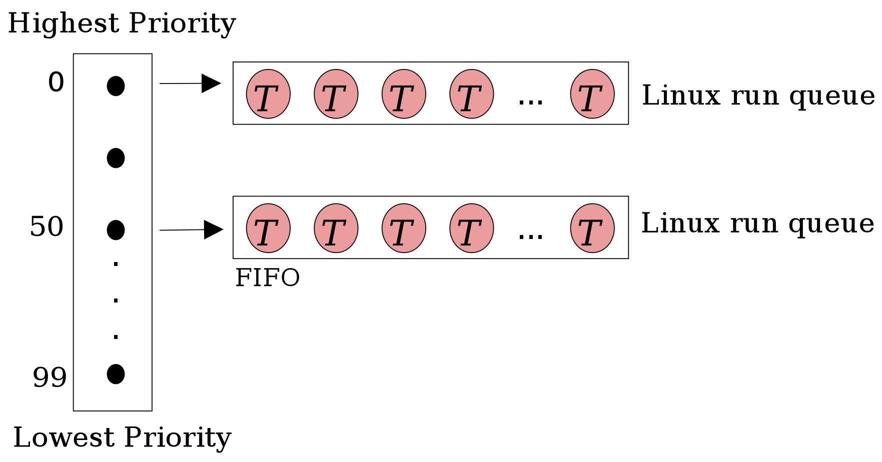
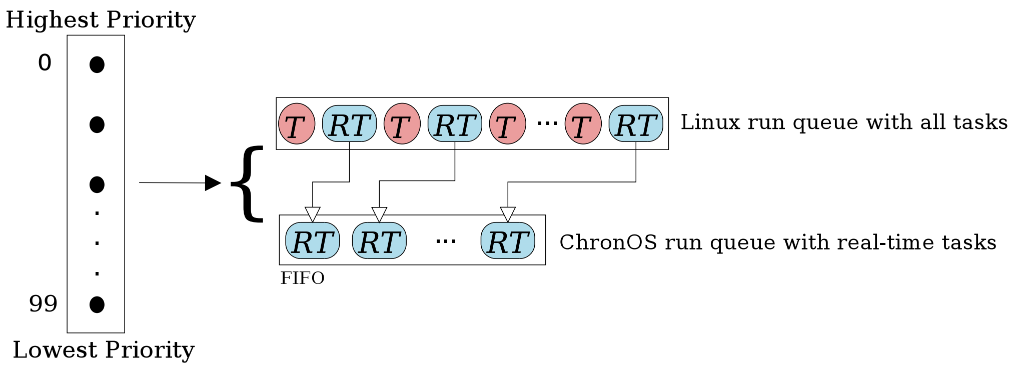


Figure 5.2: Linux real-time priority bitmap and run-queues

In order to achieve an `O(1)` scheduler, Linux implements a bit-map for each priority. There are 140 priority levels. [0 . . . 99] are referred to as real-time priorities while [100 . . . 140] are called "nice" priorities. In the kernel-space, "0" is the highest real-time priority while "99" is the least (which is opposite to that in the user-space). Figure 5.2 shows the section of the priority bit-map which maps to the real-time priorities inside the kernel. As shown in the figure, each priority has a run-queue of active tasks on the system. The default scheduling algorithm used in Linux is `SCHED_NORMAL` in which the amount of CPU that each process consumes, and the latency that it will get, is determined by the "nice" values, which are calculated by the kernel over time in an interactive fashion looking at the consumption patterns of processes in the system. The kernel starts from the highest priority bit in the bit-map, looks for tasks at that priority level and executes them before going to the next level. The key idea is to give preference to higher priority tasks.

The ChronOS scheduler extends the Linux `O(1)` scheduler. However, to differentiate between normal tasks and real-time tasks created by the ChronOS middle-ware, we add additional parameters to the `struct task_struct` to specify the real-time properties of a task, such as the task's worst case execution cost, deadline, period and TUFs. The ChronOS real-time tasks are tagged so that they stand out in the run-queue. In order to facilitate working on the real-time tasks, for every priority level in the bit-map we create another queue called the ChronOS real-time run-queue (`CRT-RQ`) which holds a reference to the real-time tasks in the Linux run-queue. This is illustrated in Fig 5.3. As shown in the figure, the tasks in the `CRT-RQ` are references of the real-time tasks in the default Linux run-queue.



Highest Priority
0
99
Lowest Priority
T
RT
Linux run queue with all tasks
ChronOS run queue with real-time tasks
FIFO

Figure 5.3: ChronOS real-time priority run-queue for a given real-time priority

The working of the `CRT-RQ` is similar to the Linux run-queue. When a task enters the system, it is added to the run-queue corresponding to its priority. If the new task is tagged as a ChronOS real-time task, a reference to the task is also added in the `CRT-RQ` for the specific priority level. When the ChronOS scheduler is enabled, it looks at the `CRT-RQ`, orders the run-queue based on the scheduling algorithm selected, and picks up the first task at the head of `CRT-RQ`. When the ChronOS scheduler picks up a real-time task, it is removed from both the default Linux run-queue and the `CRT-RQ`.

### 5.3.2 Scheduling Real-Time Tasks

A real-time application in ChronOS needs to specify the start and end of a real-time segment. A real-time segment is defined as a portion of the thread, which needs to be executed with real-time time constraints. This can be done using the following system calls provided by ChronOS. The detailed description of the system calls is provided in Appendix A.

`begin_rt_seg()`
: This system call is used to indicate the start of a real-time scheduling segment and also to provide the real-time timing constraints for a given task.

`end_rt_seg()`
: This system call is used to indicate the end of a real-time scheduling segment.

`set_scheduler()`
This system call is used to enable a scheduling algorithm.

The real-time scheduler is invoked at various scheduling events. A scheduling event is defined as a trigger that forces the system into a scheduling cycle resulting in a call to the scheduler where a new task is picked based on the scheduling algorithms. In ChronOS we define the following scheduling events.

**A task entering the system**
When a new task is added to the system, the scheduler is invoked. At that time, the scheduling algorithm looks at the `CRT-RQ`, orders the queue and picks the task at the head of the queue for scheduling.

**A task leaving the system**
When a task finishes its scheduling segment and leaves the system, the scheduler is invoked. The scheduling algorithm looks at the `CRT-RQ`, orders the queues and picks the task at the head of the queue for scheduling.

**A resource being requested**
When a task requests for a resource, ChronOS tags the task as `RESOURCE_REQUESTED` and invokes the scheduler. This is done to let the scheduling algorithm look at the dependency chain based on the resource requested and pick the task that is best suited for execution.

**A resource being released**
When a task releases a resource, ChronOS invokes the scheduler in order to allow a new task to be picked which might be blocking on the resource that was just released. The decision to choose the new task is done by the scheduling algorithm.

Using the `set_scheduler()` system call, a scheduling algorithm can be selected. All scheduling algorithms are created as Linux modules in ChronOS which provides the flexibility to add or remove any scheduling algorithm from a running kernel without restarting the system. The scheduling algorithms are implemented in a modular fashion using a set of functions that we refer to as the "scheduler plugin". Once a scheduler is selected for a set of processors, all the real-time tasks that are added to the system on those processors are scheduled using the the selected scheduling algorithm. The scheduler plugin is described in detail in the subsequent section.

Figure 5.4 illustrates an example of scheduling on a single processor machine. As the scheduling algorithms are written as modules in ChronOS, they can be loaded into a running kernel using `modprobe`, which registers the scheduling algorithms, adding them to the list of available schedulers in ChronOS. In Figure 5.4, the call to `set_scheduler()` system call is made from the real-time application to select `EDF` as the real-time scheduler. ChronOS checks if

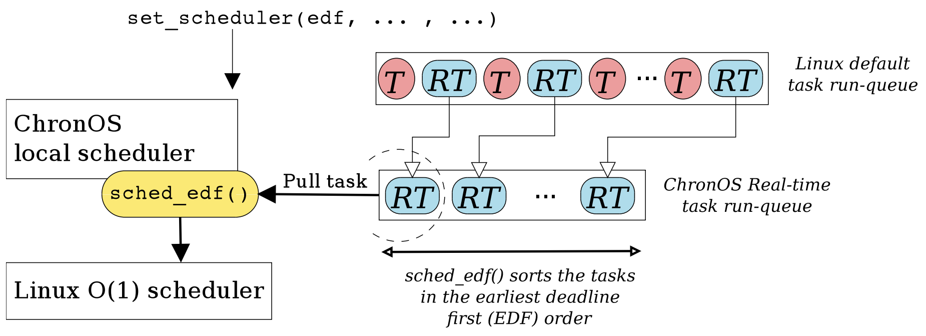

set_scheduler(edf, ... , ...)
ChronOS local scheduler
sched_edf()
Pull task
Linux O(1) scheduler
T
RT
Linux default task run-queue
ChronOS Real-time task run-queue
sched_edf() sorts the tasks in the earliest deadline first (EDF) order

Figure 5.4: ChronOS scheduling approach on a single-processor system

`EDF` kernel module is available. If the scheduler is found, ChronOS loads the plugin and makes it the default ChronOS local scheduler for running real-time tasks. All the real-time tasks are now added to the `CRT-RQ`. At every scheduling event, the ChronOS scheduler invokes `sched_edf()`, which sorts the `CRT-RQ` in Earliest Deadline First order. The head of the queue now represents the earliest deadline task which is pulled by the ChronOS local scheduler and given to the Linux `O(1)` scheduler for execution.

## 5.4 Multiprocessor Scheduling

Scheduling on multiprocessors can be mainly categorized into two forms – partitioned scheduling and global scheduling. ChronOS supports both these variants. In this section we describe the details of both these architectures, and discuss their design and implementation.

### 5.4.1 Partitioned Scheduling

Partitioned scheduling can be described as uniprocessor scheduling done on multiprocessors. The key idea of partitioned scheduling is to divide the task-set using an off-line heuristic, as partitioning a set of tasks on $M$ processors has been shown to be equivalent to the bin-packing problem [46] and hence NP-hard in the strong sense. Baruah *et al.* present a polynomial-time algorithm to partition collection of sporadic tasks onto a $M$ processors in [8].

Figure 5.5 illustrates the partitioned scheduling approach used in ChronOS. The task-set is partitioned off-line using polynomial-time heuristics such as first-fit, worst-fit, and best-fit. In Figure 5.5 we simulate a two processor system. The heuristic divides the task-set into two processor bins as shown. Once all the tasks have been divided, the real-time application sets the affinity of each of the tasks to the processors they have been assigned to. This is done to ensure that the tasks are added to the run-queue of their respective assigned proces-

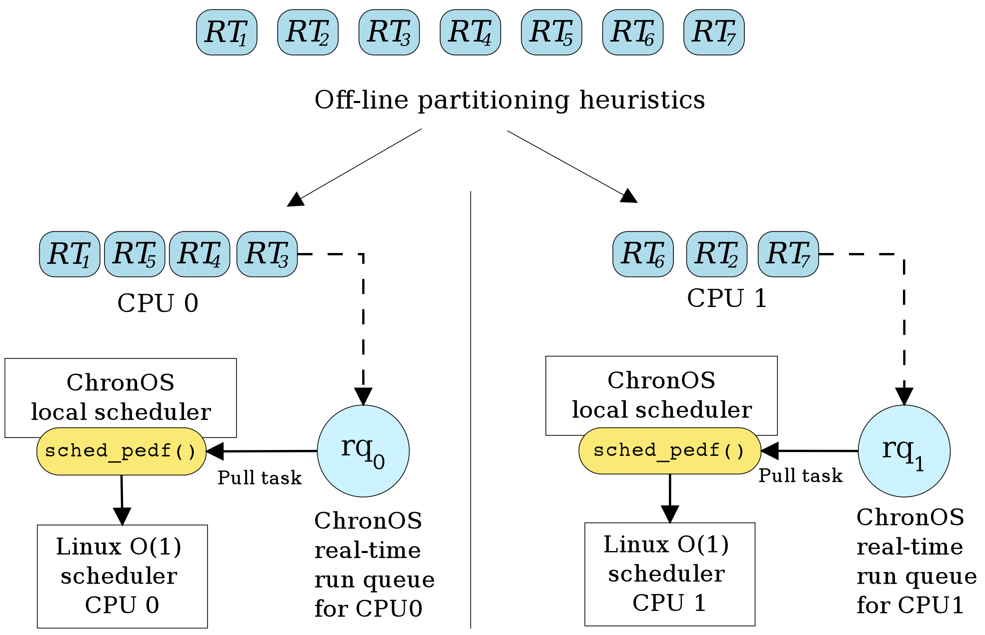

$RT_1$ $RT_2$ $RT_3$ $RT_4$ $RT_5$ $RT_6$ $RT_7$
Off-line partitioning heuristics
$RT_1$ $RT_5$ $RT_4$ $RT_3$
CPU 0
$RT_6$ $RT_2$ $RT_7$
CPU 1
ChronOS local scheduler
sched_pedf()
Pull task
$rq_0$
$rq_1$
Linux O(1) scheduler CPU 0
Linux O(1) scheduler CPU 1
ChronOS real-time run queue for CPU0
ChronOS real-time run queue for CPU1

Figure 5.5: ChronOS partitioned scheduling approach on a multiprocessor system

sors. The reference to these real-time tasks is also added to the `CRT-RQ` of their respective assigned processors. As partitioned scheduling is an extension of uniprocessor scheduling, we set the partitioned scheduler as the local ChronOS scheduler on all processors using the `set_scheduler()` system call. Each processor runs its scheduling algorithm independently. At every scheduling event, the processor enters its local scheduler, looks at the local run-queue, and using the selected scheduling algorithm (in the figure shown as `P-EDF`), picks the next task to be executed. As the tasks have already been partitioned, we disable Linux's load balancing mechanism to prevent tasks from being migrated between processors. Some of the algorithms that have been implemented using this approach in ChronOS are `P-EDF` and `P-DASA`.

### 5.4.2 Global Scheduling

Most of the multiprocessor scheduling algorithms, such as `G-EDF`, `G-NP-EDF`, `Pfair`, `gMUA`, `G-GUA`, and `NG-GUA` are based on global scheduling. The main idea behind global scheduling is that the tasks are assigned to a global queue instead of individual local queues. The scheduling algorithm on each processor looks at the global queue and either makes a scheduling decision for itself and every other processor in the system (such as `G-EDF`, `Pfair`, `G-GUA`, `NG-GUA`) or picks a task only for itself (such as `G-NP-EDF`).

Figure 5.6 illustrates the global scheduling approach used in ChronOS. In order to implement global scheduling inside ChronOS, we create another level of scheduling abstraction. At

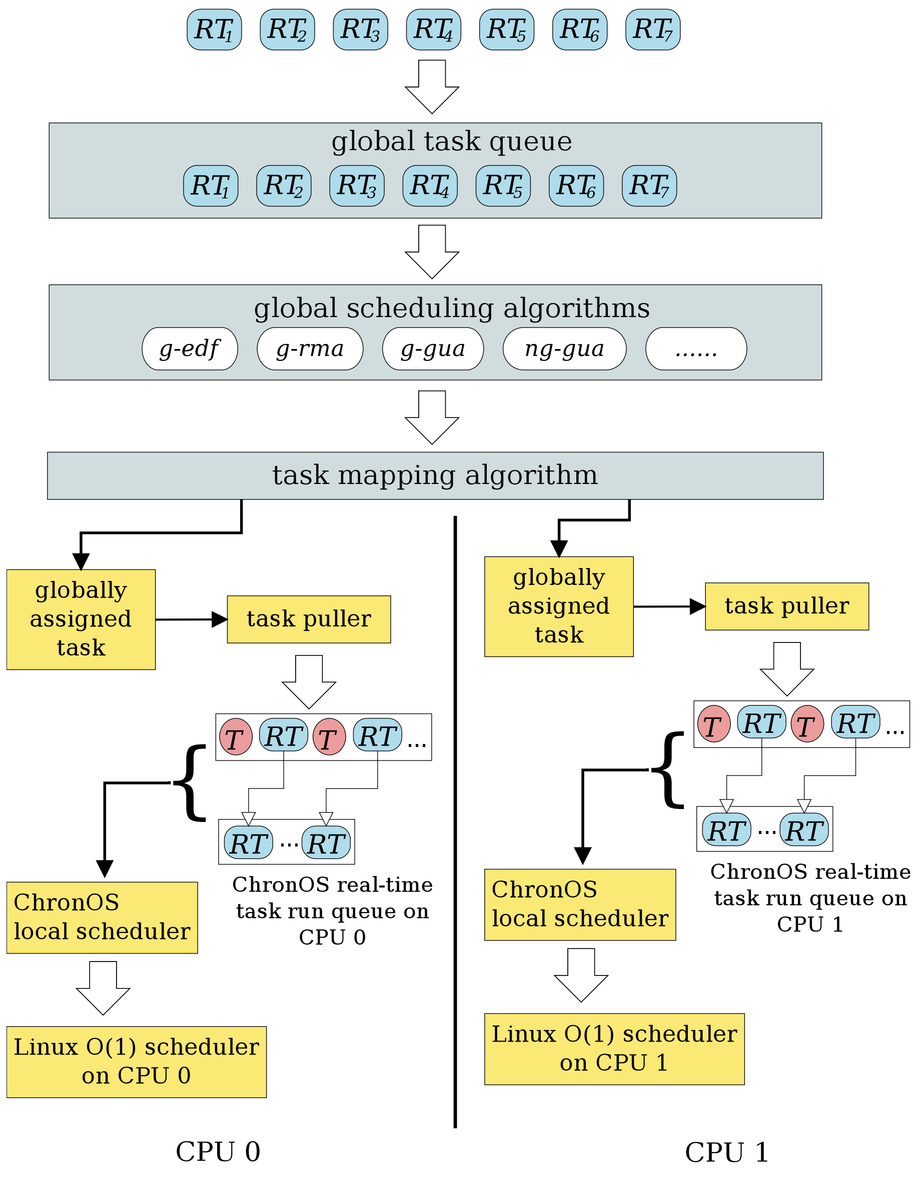

$RT_1$ $RT_2$ $RT_3$ $RT_4$ $RT_5$ $RT_6$ $RT_7$
global task queue
global scheduling algorithms
g-edf
g-rma
g-gua
ng-gua
......
task mapping algorithm
globally assigned task
task puller
T RT T RT ...
RT ... RT
ChronOS real-time task run queue on CPU 0
ChronOS local scheduler
Linux O(1) scheduler on CPU 0
CPU 0
ChronOS real-time task run queue on CPU 1
Linux O(1) scheduler on CPU 1
CPU 1

Figure 5.6: ChronOS global scheduling approach on a multiprocessor system

the top we have the "global scheduler" which looks at the "global task queue". The global scheduler maps to a "local scheduler" on individual processors which extends from the Linux `O(1)` scheduler. The global scheduler (invoked on a processor) can either pick a task for itself or decide for all the processors on the system (depending on the scheduling policy used). If the global scheduler needs to choose tasks for all available processors on the system (such as `G-EDF, G-GUA or NG-GUA`), it picks the top $M$ tasks. These tasks are given to the task mapping algorithm which maps these tasks on $M$ underlying processors. The tasks assigned by the task mapping algorithm are pushed into the "globally assigned task" block from where the "task puller" on each CPU picks up the task and moves it to the head of its local queue. The default local scheduling algorithm for global scheduling algorithms on each processor is `SCHED_FIFO`, which picks the head of the `CRT-RQ` queue and gives the task to the Linux `O(1)` scheduler for execution.

In global scheduling under ChronOS, tasks can be created and assigned to any processor. The tasks continue to reside on the Linux run-queue of the processor they were created on. The global queue has a reference to all the real-time tasks from all the processors.

As mentioned earlier, there are two ways in which global scheduling can be achieved. In the first approach, the global scheduler picks a task for itself from the global queue, such as `G-NP-EDF`. We refer to this as the "Application Concurrent Scheduling Model". In the second approach, the global scheduler picks the tasks for all $M$ available processors, such as `Pfair, NG-GUA, G-GUA`. We refer to this as the "Stop-the-World Scheduling Model" (STW). These architecture models are described in detail in Sections 5.4.2.1 and 5.4.2.2, respectively.

#### 5.4.2.1 Application Concurrent Scheduling Model

Figure 5.7 illustrates the application concurrent scheduling model for global scheduling in ChronOS. For the sake of explanation of the model, we will assume that the scheduling algorithm picks the first task that is at the head of the queue. At the beginning, task $T_6$ is running on processor $P_0$ and task $T_8$ is running on processor $P_1$. As shown in the figure, $T_8$ finishes before $T_6$. However, $T_6$ is not preempted on $P_0$. It continues to run. After $T_8$ finishes, it generates a scheduling event. $P_1$ enters the global scheduler, picks the first task $T_3$ from the global queue and assigns it to the "globally assigned task" block. The local scheduler on $P_1$ pulls the task $T_3$ and starts executing it. While $P_1$ is pulling the task, $T_6$ finishes on processor $P_0$ and generates a scheduling event. It pulls $T_1$ from the global queue and starts executing it without preempting $T_3$ on $P_1$. The same procedure is repeated for other scheduling events.

There might be a scenario when both processors finish their tasks at the same time. As shown in the Figure 5.7, when $T_2$ finishes on $P_0$, $T_4$ finishes on $P_1$ around the same time. As the global task queue is common between all the processors, while $P_0$ enters the global scheduler, $P_1$ blocks. The moment $P_0$ is done with the schedule, $P_1$ unblocks and enters the scheduler. The `G-NP-EDF` and `G-FIFO` are some of the algorithms that use such an architecture model.

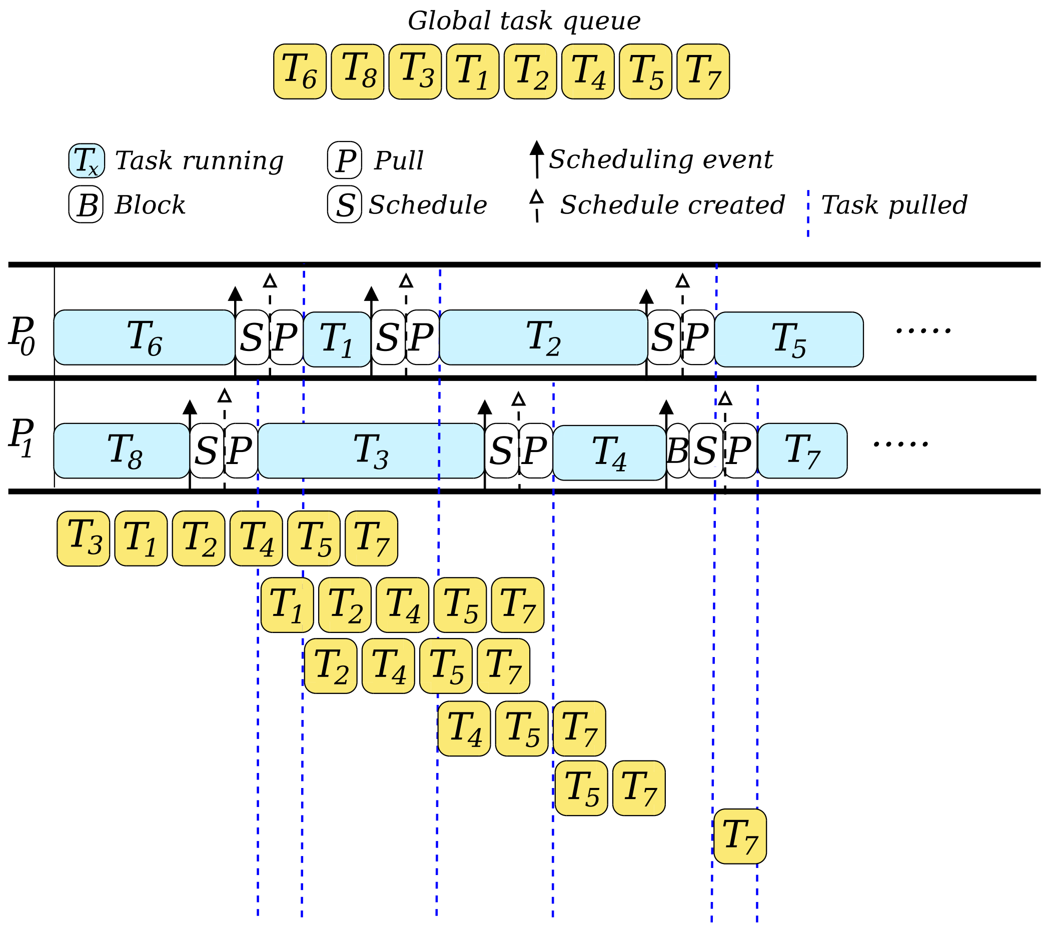


Figure 5.7: "Application Concurrent" architecture model for global scheduling algorithms

The downside of the application concurrent model is that it can only be used by scheduling algorithms that do not have resource dependencies (such as locks). This is because when resources are used, we need to look into more complicated scheduling mechanisms to figure out the best task to be executed that respects resource boundaries.

### 5.4.2.2 Stop-the-World Scheduling Model

In order to allow global scheduling algorithms with resource management, ChronOS implements the Stop-the-World scheduling model. Figure 5.8 provides an illustration of the model. For the sake of explanation of the model we will assume that the scheduling algorithm selects the tasks for execution that are not dependent on any other tasks in the system. Let us assume that the global task queue has the tasks eligible for the final schedule. The figure shows the dependency relation of the tasks in the global task queue with each other. Task $T_4$ needs a resource which is owned by task $T_2$ which in turn requires a resource that is

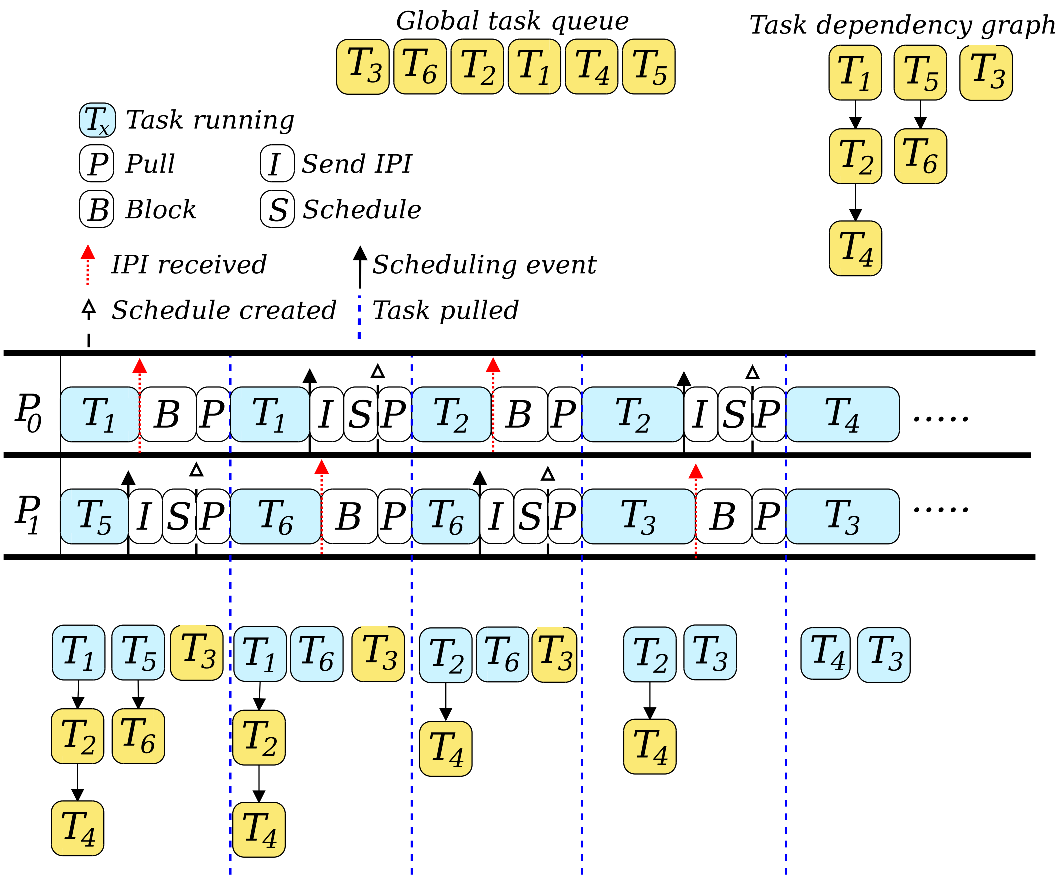


Figure 5.8: “Stop-the-World” architecture model for global scheduling algorithms

being held by task $T_1$. In a similar fashion, task $T_6$ needs a resource which is owned by task $T_5$. Task $T_3$, on the other hand, does not have any dependents. Let us assume that the scheduling algorithm considers the tasks that have the maximum dependents as the most eligible tasks for the final schedule.

Given all the assumptions, Figure 5.8 shows that tasks $T_1$ and $T_5$ are the current executing tasks on processors $P_0$ and $P_1$, respectively. Task $T_5$ finishes first and generates a scheduling event. In the Stop-the-World model, once a scheduling event is generated, the schedule needs to be created for all the processors. This requires a processor to be able to force a scheduling event on all other processors. When task $T_5$ triggers the scheduling event, processor $P_1$ sends an “Inter-processor Interrupt” (`IPI`) to all the processors on the system. In Linux, at the end of every interrupt handler, the call to the scheduler is made. This is done in order to pick up the next task for execution after the interrupt has been handled. The `IPI` used by the scheduler is a dummy interrupt. The interrupt handler for the scheduling `IPI` does not do anything but call the `schedule()` function at the end, which forces the processor to enter the scheduler.

In the example shown in Figure 5.8, processor $P_1$ sends an `IPI` to all the available processors on the system. After sending the `IPI`, processor $P_1$ enters its global scheduler and looks at the available tasks in the run-queue to create the schedule. In the meanwhile, processor $P_0$ receives the `IPI` and is forced into the scheduler. However, as processor $P_1$ is already in the global scheduler, processor $P_0$ blocks. The global scheduler on processor $P_1$ picks two eligible tasks (assuming a two-processor system) and hands these tasks to the mapping algorithm. The task mapper pushes these tasks into the "globally assigned task" block of each processor. Once the mapper is done, each individual processor pulls its globally assigned task to the head of their local queue. The local scheduler (which is `SCHED_FIFO`) on each processor, picks the task at the head of the local queue and gives it to the Linux `O(1)` scheduler for execution.

Even if the tasks do not have a dependency relationship, the Stop-the-World model works the same way as mentioned above. The scheduling algorithm looks at the global queue and picks the $M$ tasks that are eligible for final schedule and hands them to the mapping algorithm, which is explained in detail in the next section.

### 5.4.3 Mapping Tasks to Processors

Figure 5.9 illustrates the default mapping algorithm used in ChronOS for global scheduling. The job of the mapping algorithm is to take the $M$ most eligible tasks that have been selected by the scheduling algorithm and map them to the $M$ available processors on the system. The key idea is to be able to reduce task migrations, thus preserving cache coherence. The algorithm shown in Figure 5.9 is selected as the default for all global scheduling algorithms. However, the scheduler plugin infrastructure of ChronOS allows users to override the default mapping algorithm and provide their own implementation.

In ChronOS the mapping is done using a three-pass algorithm. In Figure 5.9 tasks $RT_5$, $RT_8$, $RT_1$ and $RT_4$ represent four real-time tasks that have been selected by the global scheduling algorithm at the end of a scheduling event. In order to understand the mapping algorithm we give a snapshot of the per-processor run-queues. Each run-queue shows the tasks that belong to the individual processors. We also highlight the current running task on each of the processors before the scheduling event was triggered, which led to the creation of the new global schedule. Task $RT_2$ is the current running task on processor $P_0$, task $RT_5$ on $P_1$, task $RT_9$ on $P_3$ while task $RT_7$ is the current running task on $P_3$.

In the first pass, the algorithm goes over the final schedule and maps tasks to processors that are the current running tasks on that processor. This provides cache coherence. As shown in Figure 5.9, task $RT_5$ is the current running task on $P_1$. Hence, it is mapped to processor $P_1$. In the second pass, the algorithm goes over the final schedule and maps tasks to processors that belong to that processor's run-queue. This prevents tasks from being unnecessarily migrated. As shown in the figure, task $RT_4$ belongs to processor $P_0$ and hence it is mapped to processor $P_0$. In the similar fashion, task $RT_8$ belongs to processor $P_2$ and

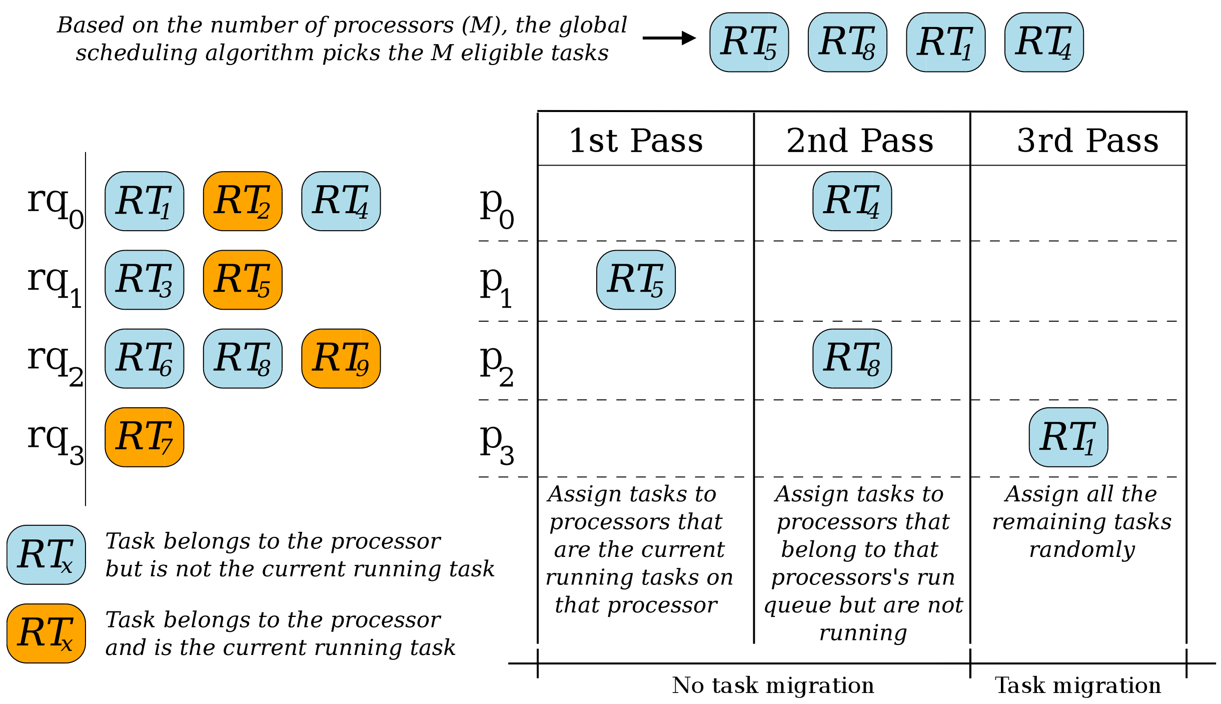

Based on the number of processors (M), the global scheduling algorithm picks the M eligible tasks
RT5 RT8 RT1 RT4
rq0 RT1 RT2 RT4
rq1 RT3 RT5
rq2 RT6 RT8 RT9
rq3 RT7
RTx Task belongs to the processor but is not the current running task
RTx Task belongs to the processor and is the current running task
1st Pass
2nd Pass
3rd Pass
p0
p1
p2
p3
Assign tasks to processors that are the current running tasks on that processor
Assign tasks to processors that belong to that processors's run queue but are not running
Assign all the remaining tasks randomly
No task migration
Task migration

Figure 5.9: Task mapping for global scheduling algorithms in ChronOS

it gets mapped to the same processor. In the last pass, the mapping algorithm randomly assigns the leftover tasks to the remaining processor(s). As shown in the figure, task $RT_1$ is assigned to processor $P_3$. However, task $RT_1$ actually belongs to processor $P_0$. As a result, this step results in the migration of the mapped tasks.

The key idea of the mapping algorithm is to reduce task migrations. However, if the final schedule consists of $M$ tasks that all belong to the same processor, the worst case migration cost is $M - 1$ task migrations. In such a case, cache-aware scheduling algorithms can be used to create a cache conscious schedule, and thus handle their own mapping, overriding the default mapping algorithm. Guan *et al.* [38] and Calandrino *et al.* [17] present different cache-aware real-time scheduling algorithms. Stenstrm provides a survey of various cache coherence schemes on multiprocessors in [64].

## 5.5 Implementing Scheduling Algorithms in ChronOS

### 5.5.1 Single-processor Scheduling

```
struct rt_sched_local
{
  char *name;
  int number;
  struct rt_info* (*schedule) (struct list_head *head, int flags);
};
```

Listing 5.1: ChronOS Single-processor Scheduler Plugin

The code listing 5.1 shows the ChronOS plugin for single-processor scheduling. When implementing a new scheduler, the user needs to specify a name and a unique number for the scheduler. The `*schedule` function pointer provides the function syntax for the main scheduling function where the algorithmic logic for the scheduler needs to be implemented. The function provides the `head` pointer to the `CRT-RQ` which has the list of all the real-time tasks released and eligible for schedule.

### 5.5.2 Multiprocessor Scheduling

```
struct rt_sched_global
{
  char *name;
  int number;
  struct rt_info* (*schedule) (struct list_head *head, int flags, int cpus);
  struct rt_info* (*preschedule) (struct list_head *head, int flags);
  struct rt_sched_arch *arch;
};
```

Listing 5.2: ChronOS Multiprocessor Scheduler Plugin

The code listing 5.2 shows the ChronOS plugin for multiprocessor scheduling. When implementing a new scheduler, the user needs to specify a name and a unique number for the scheduler. The `*schedule` function pointer provides the function syntax for the main scheduling function where the algorithmic logic for the scheduler needs to be implemented. The function provides the `head` pointer to the global real-time queue which has the list of all the real-time tasks released across the system and eligible for schedule. The function also provides the number of processors on which multiprocessor scheduling has been enabled. The `*preschedule` function pointer provides the function syntax for pre-scheduling. This is an architecture feature provided by ChronOS for scheduling algorithms that need to do some preprocessing before entering the main scheduler (such as checking if the tasks have missed their deadlines and in such a case aborting them).

Appendix C shows a sample implementation of the kernel modules for single-processor and multi-processor scheduling.

# Chapter 6

# Experimental Results

In order to evaluate the performance of NG-GUA and G-GUA, we implement the scheduling algorithms in ChronOS along with their state-of-the-art competitors, such as G-EDF, G-NP-EDF, G-FIFO, gMUA, P-EDF, P-DASA. We conduct experiments and evaluate the results on a two-core, four-core and an eight-core platform. In this section, we discuss the experimental setup in detail and present the result and the analysis.

## 6.1 Experimental Setup

### 6.1.1 Platform Specifications

We conduct our experiments on three different platforms, each with an increasing number of processing units. Table 6.1 gives the detailed specifications of each platform used.

The first platform is based on the Intel dual-core P8600 processor with a CPU frequency of 2.4 GHz and a 3 MB L2 cache. The second platform is a quad-core machine based on AMD Phenom 9650 processor with a CPU frequency of 2.3 GHz and a 2 MB L3 cache. Finally, we use an eight-core platform using two Intel Xeon E5520 quad-core processors, each having a CPU frequency of 2.8 GHz and an 8 MB L3 cache. These platforms provide a rich environment with different processing speeds and cache footprints, thus allowing us to verify the viability of multiprocessor scheduling algorithms under different platforms.

### 6.1.2 Test Application

The NG-GUA and G-GUA algorithms are flexible about the task arrival pattern. We do not assume any specific model (such as periodic, aperiodic, sporadic). Tasks can arrive at any time in the system and create scheduling events. However, in order to evaluate our algorithms

Table 6.1: Specifications of the various platforms used for experimental evaluation

| **Specifications** | **Two-core** | **Four-core** | **Eight-core** |
|---|---|---|---|
| *Processor Type* | Intel Core 2 Duo P8600 | AMD quad-core Phenom 9650 | Intel quad-core Xeon E5520 |
| *No of processors* | 1 | 1 | 2 |
| *Total physical cores* | 2 | 4 | 8 |
| *Total logical cores* | 2 | 4 | 16 |
| *CPU Frequency* | 2.4 GHz | 2.3 GHz | 2.8 GHz |
| *Memory* | 2 GB | 2 GB | 8 GB |
| *Cache* | 3 MB L2 | 128 KB L1, 512 KB L2, 2 MB L3 | 4x256 KB L2, 8 MB L3 |

against the other state-of-the-art algorithms, which specify a particular task arrival pattern, we default to a periodic model. This helps to easily quantify the schedulability criteria of the algorithms and allows us to compare the performance with other scheduling algorithms.

We create a real-time test application using ChronOS APIs. The test application periodically fires real-time tasks with specified time-constraints. For each task, we use a `burn_cpu(wcet)` function, which takes the `wcet` (worst case execution time) as an input and burns processor cycles for that amount of time. This allows us to simulate a real-time task which executes until its worst case execution time. In our test application we use the “thread-is-a-phase” model, where each periodic instance of the task is represented as a separate thread. Using the task’s period as the relative deadline, we fire threads periodically.

### 6.1.3 Performance Parameters

We measure the Deadline Satisfaction Ratio (DSR) and the Accrued Utility Ratio(AUR). Equations 6.1 and 6.2 give the formulas for finding the DSR and AUR.

$$DSR_U = \frac{Tasks\ that\ met\ their\ deadlines\ at\ utilization\ load\ U}{Total\ tasks\ in\ the\ system} \tag{6.1}$$

$$AUR_U = \frac{Accrued\ utility\ of\ tasks\ that\ met\ their\ deadlines\ at\ utilization\ load\ U}{Total\ possible\ accrued\ utility} \tag{6.2}$$

At a given utilization load $U$, the DSR is measured as the ratio of the tasks that met their deadlines to the total number of tasks in the system. In the similar fashion, the AUR is measured as the total accrued utility of the tasks that met their deadlines to the total possible accrued utility in the system.

We create various task-sets using a random task-set generator. We discuss the task-sets used in detail in the individual sections. All the results are presented as an average of ten runs. We also present the standard deviation variation for each data point. Our focus is primarily on two types of experiments — (1) we evaluate the performance of NG-GUA and G-GUA without dependencies (locks), whose results are discussed in Section 6.2; and (2) we evaluate the performance of the algorithms in the presence of dependencies (locks), whose results are discussed in Section 6.3.

## 6.2 Results Without Dependencies

In this section we evaluate the performance of NG-GUA and G-GUA without dependencies (such as locks). We compare the results with other state-of-the-art global scheduling algorithms, such as G-EDF, G-NP-EDF and G-FIFO. As mentioned earlier, gMUA is a special case of NG-GUA without dependencies. As a result, for the non-dependent case, the implementation of gMUA in ChronOS is similar to NG-GUA (as we skip the function to create the DAG and treat all the tasks as zero in-degree). Hence, for the non-dependent case, the results for gMUA are similar to NG-GUA. In order to avoid confusion, we do not show the gMUA results on the plots but it is considered in the final summary. We also compare the results of our algorithms with partitioned algorithms, such as P-EDF and P-DASA.

### 6.2.1 Comparison with Global Scheduling Algorithms

In order to compare with global scheduling algorithms, we consider downward "step" TUF based model. We create random task-sets with three types of TUF assignment policies. Figure 6.1 illustrates the model used.

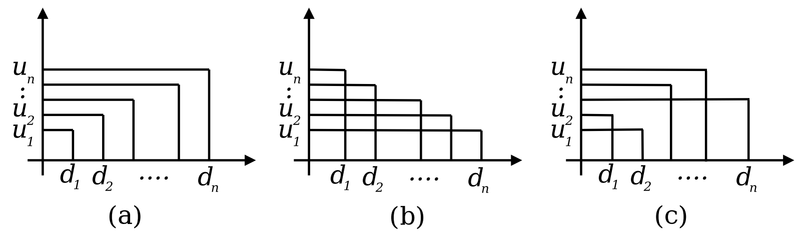


Figure 6.1: Task-sets created with variable TUFs (a) Increasing Utility (IU); (b) Decreasing Utility (DU); (c) Random Utility(RU)

In the Increasing Utility (IU) model, the utilities assigned to the tasks are proportional to their deadlines. The task with the earliest deadline has the least utility while the task with a later deadline has the highest utility. In the Decreasing Utility (DU) model, the utilities assigned to a task are inversely proportional to their deadlines. The task with the earliest deadline has the highest utility while the task with a later deadline has the least utility. In the Random Utility (RU) model, the tasks are assigned random utilities with no two tasks having the same utility. These models are used to accomplish two main goals — (i) to ascertain whether, irrespective of the TUF ordering, our algorithms perform comparable to the competitors; and (ii) to ensure that we do not create a bias based on the TUF assignment against the deadline-based algorithms. Hence, the model covers the worst-case to the best-case scenarios.

On the two-core and four-core platform, we consider three types of task-sets with an increasing number of tasks.

1. The first task-set uses 5 tasks with deadlines/periods in the range of $[500ms - 5000ms]$, with the utilization load for each task in the range of $[0.1 - 0.4]$.
2. The second task-set uses 12 tasks with deadlines/periods in the range of $[300ms - 20000ms]$, with the utilization load for each task in the range of $[0.01 - 0.4]$.
3. The third task-set uses 27 tasks with deadlines/periods in the range of $[50ms - 7500ms]$, with the utilization load for each task in the range of $[0.01 - 0.3]$.

On the eight-core platform, we consider two types of task-sets.

1. The first task-set uses 27 tasks with deadlines/periods in the range of $[50ms - 7500ms]$, with the utilization load for each task in the range of $[0.01 - 0.3]$.
2. The second task-set uses 50 tasks with deadlines/periods in the range of $[50ms - 10000ms]$, with the utilization load of each task in the range of $[0.001 - 0.3]$.

Using these task-sets we cover a total deadline/period range from $[50ms - 20s]$. The tasks are assigned utilities in the range of $[50 - 10000]$. In the IU model, the task with the least period is assigned a utility of 50 and the task with the largest period is assigned a utility of 10000, and vice-versa for the DU model. In the RU model, tasks are assigned utilities randomly. As we are using a utility based model, we ensure that we cover a wide deadline/period range in order to avoid any bias against deadline-based scheduling algorithms.

Table 6.2 shows the legend used in the experimental results. All the experimental result are shown as an average of ten runs. The plots cover a utilization load in the range of $[0 - 8]$ for the two-core platform, $[0 - 10]$ for the four-core platform and $[0 - 12]$ for the eight-core platform. Each data point shows the standard deviation as a vertical error bar.

In the subsequent sections we discuss the results on two-core, four-core and eight-core platforms.

Table 6.2: Legend used in experimental results, based on TUF-based models, for comparison with other global scheduling algorithms

| Symbol | Description |
|---|---|
| `IU` | Increasing Utility model |
| `DU` | Decreasing Utility model |
| `RU` | Random Utility model |
| `NL` | No locks used |
| `xT` | Task-set used with $x$ number of tasks |
| `xC` | Experiment using $x$ number of processors |

#### 6.2.1.1 Two-core Platform Results

On the two-core platform, we ran the experiments using the 5-task (5T), 12-task (12T) and 27-task (27T) task-sets using IU, RU and DU models. Figures 6.2 and 6.3, respectively, show the AUR and DSR results for the 5T using the DU model. We observe that both NG-GUA and G-GUA outperform other deadline-based algorithms.

We observe the following:

(1) G-EDF meets all deadlines for $U \leq 1(m/2)$, as $m = 2$ for a two-core platform, which is the lower utilization bound for G-EDF. However, during overloads, G-EDF suffers from the domino effect and starts losing deadlines.

(2) NG-GUA and G-GUA provide a better deadline satisfaction ratio during overloads. The main benefit of the algorithms can be observed in Figure 6.2 for the accrued utility. At 250% utilization load we see an improvement of around 300% in AUR over that of G-EDF, G-NP-EDF and G-FIFO. At 400% utilization load, there is an improvement of around 350% in AUR.

(3) Finally, the task-set uses a DU model, which gives the highest utility to the tasks that have the earliest periods/deadlines. We observe that NG-GUA and G-GUA perform better in the DU model as compared to the deadline-based algorithms.

When compared with each other, we see that G-GUA performs better than NG-GUA, which is an expected behavior. G-GUA is optimized to accrue more utility during overloads. On the other hand, NG-GUA performs well during underloads by trying to meet as many deadlines as possible, thus following a G-EDF behavior, but it accrues higher utility during overloads.

We observe some variability in the results (shown as the standard deviation). This amount of variance is acceptable given the fact that it is impossible to have the same operating system environment for every test execution. Due to the `PREEMPT_RT` patch, ChronOS is able to withstand jitter from non real-time based applications. However, there are critical operating system primitives that have higher priority in ChronOS which are required for the proper functioning of the operating system. These include (to name a few), the timer interrupts and the I/O interrupts, which are used while our test application fires periodic threads using timers and also when it writes the test results to a file after every data-point is captured. This also explains why most of the variability is observed during the overload scenarios and not during underloads. This is because during underloads, the total task utilization is still below the processing capacity of the system[1].

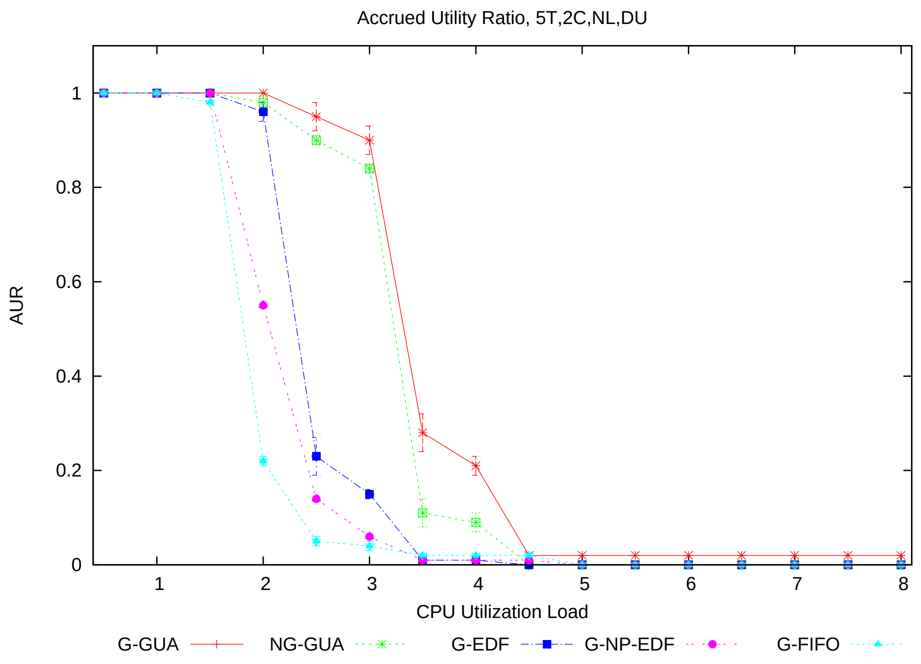


Figure 6.2: AUR vs. CPU utilization (5T, 2C, NL, DU)

The results for the AUR and DSR using the IU model for 5T are shown in Figures 6.4 and 6.5, respectively. We observe a performance improvement over the deadline-based scheduling algorithms. Both NG-GUA and G-GUA consistently accrue high utility during overloads with 700% improvement in AUR for 250% utilization load. NG-GUA performs better at meeting deadlines during underloads and its performance is comparable to G-EDF. G-NP-EDF falls behind G-EDF in meeting deadlines, which is an expected behavior [10]. In Figure 6.4, we observe a constant utility from 450% to 800% utilization load for G-GUA. This is primarily because of the smaller task-set used. The task-set has only five tasks; during high overloads, G-GUA finds at least one task that is feasible. This highlights the concept of importance over urgency. For a two-processor system, 450% utilization load is

[1]Unless stated otherwise, we will assume that the variability shown by the standard deviation in all plots is due to the same reason, as mentioned above.

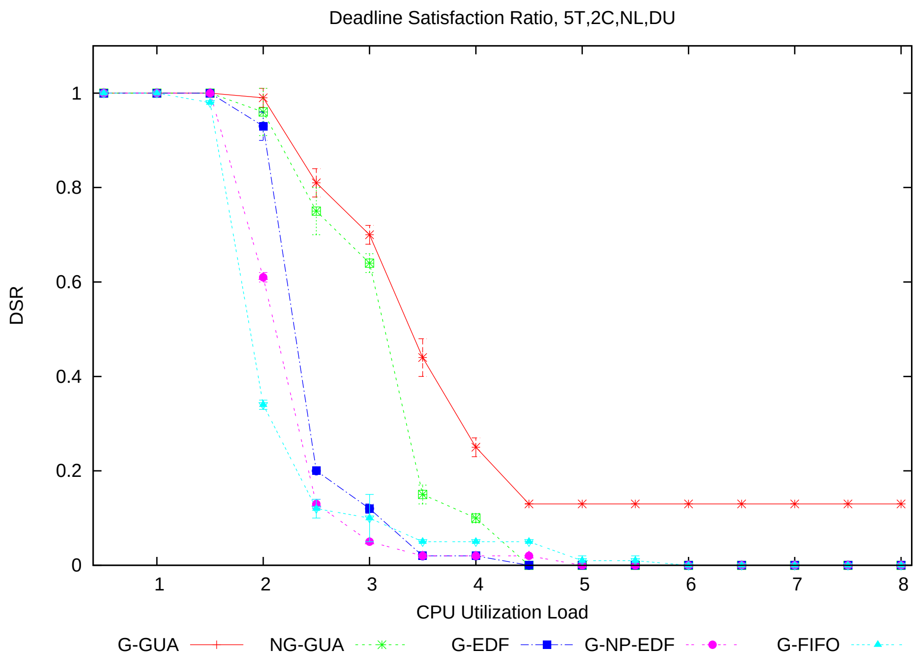


Figure 6.3: DSR vs. CPU utilization (5T, 2C, NL, DU)

twice the normal load. We observe that none of the deadline-based scheduling algorithms consider any task eligible to be scheduled. On the other hand, the UA based scheduling algorithms find at least one "important" task to be scheduled that can accrue total system utility.

Figures 6.6 and 6.7 show the AUR and DSR results, respectively, for the 5T task-set using the RU model. In the RU model, tasks are assigned random utilities. We continue to observe similar performance as seen in the earlier results for IU and DU.

In order to see the effect of increase in the number of tasks on the performance of NG-GUA and G-GUA, we use the 12T and 27T task-sets on the two-core platform. Figures 6.8, 6.9, 6.10, 6.11, 6.12 and 6.13 show the DSR and AUR results for the DU, IU and RU models, respectively, for the 12T task-set. The AUR results show that both NG-GUA and G-GUA perform better during overloads even when the number of tasks in the system are increased three-fold. Figure 6.8 shows the AUR result for the DU model. As the deadline-based schedulers are only able to meet task deadlines for $U \approx m/2$, there is a sharp plunge in the number of deadlines met (as seen in Figure 6.9). As a result, the AUR falls down drastically. This behavior is common among all the models under 12T.

We notice that the AUR results for G-GUA do not represent a constant line as was seen with the 5T results. This happens because the number of tasks in the system have increased

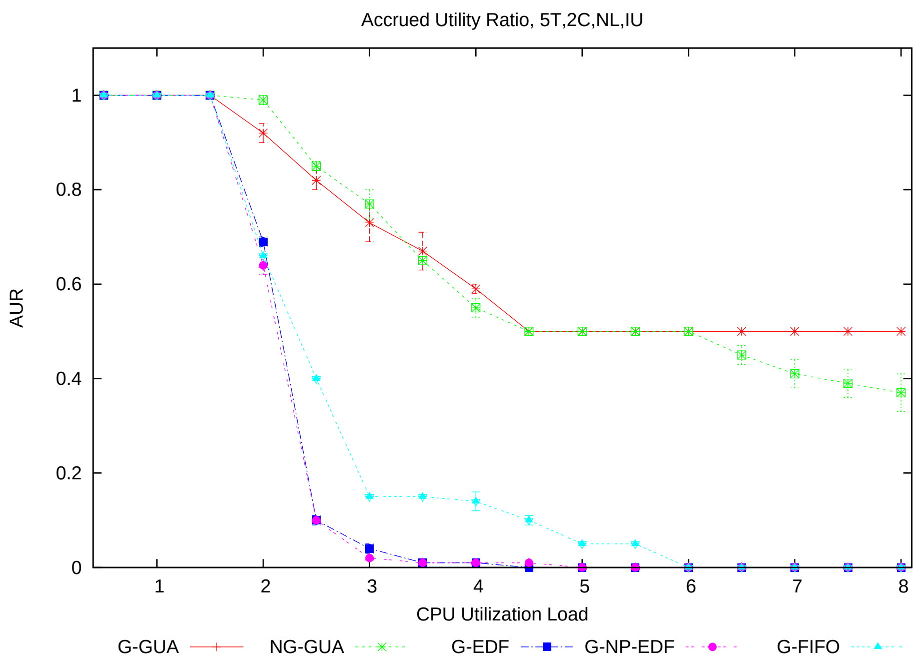


Figure 6.4: AUR vs. CPU utilization (5T, 2C, NL, IU)

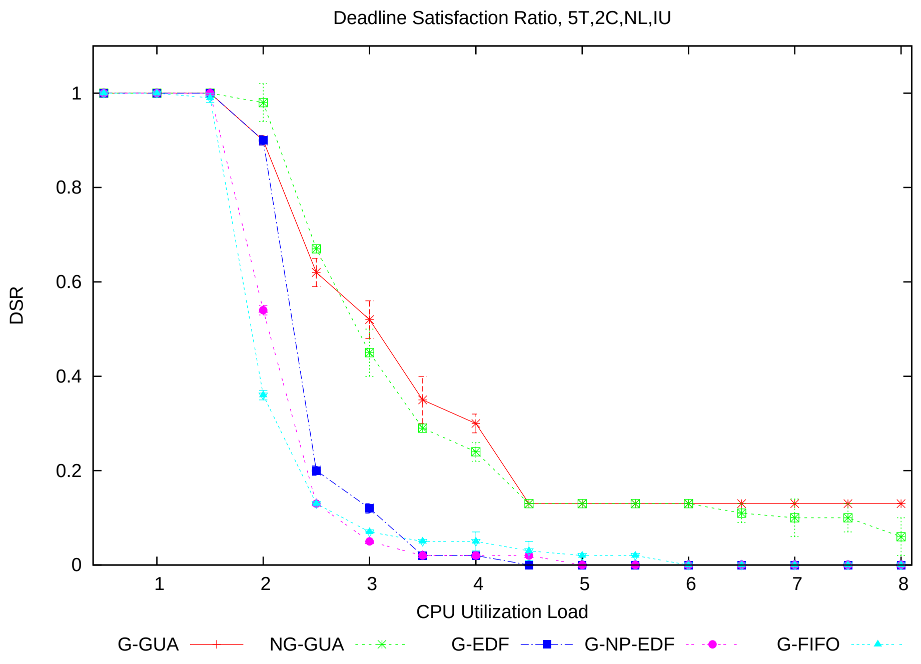


Figure 6.5: DSR vs. CPU utilization (5T, 2C, NL, IU)

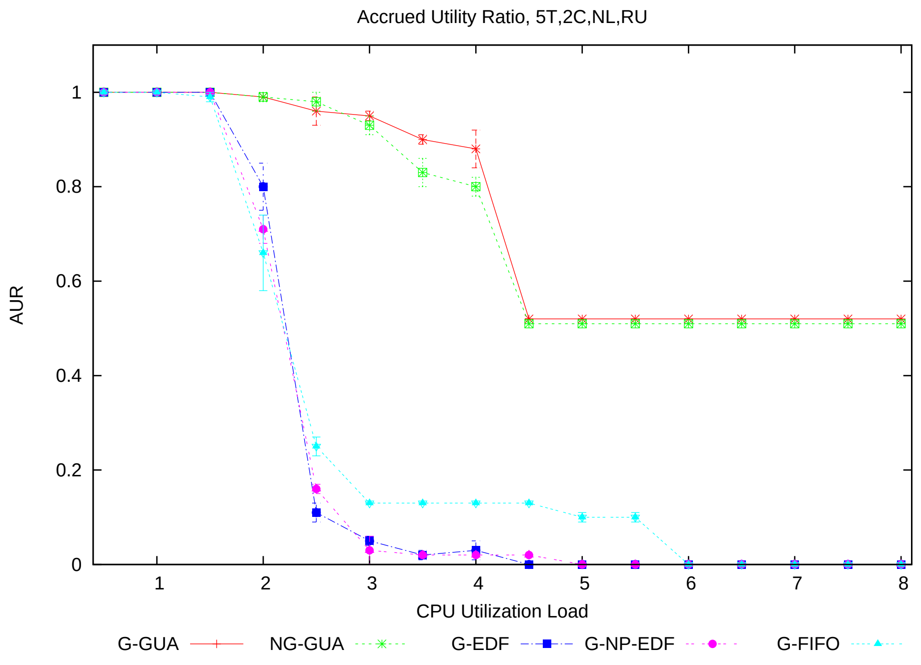


Figure 6.6: AUR vs. CPU utilization (5T, 2C, NL, RU)

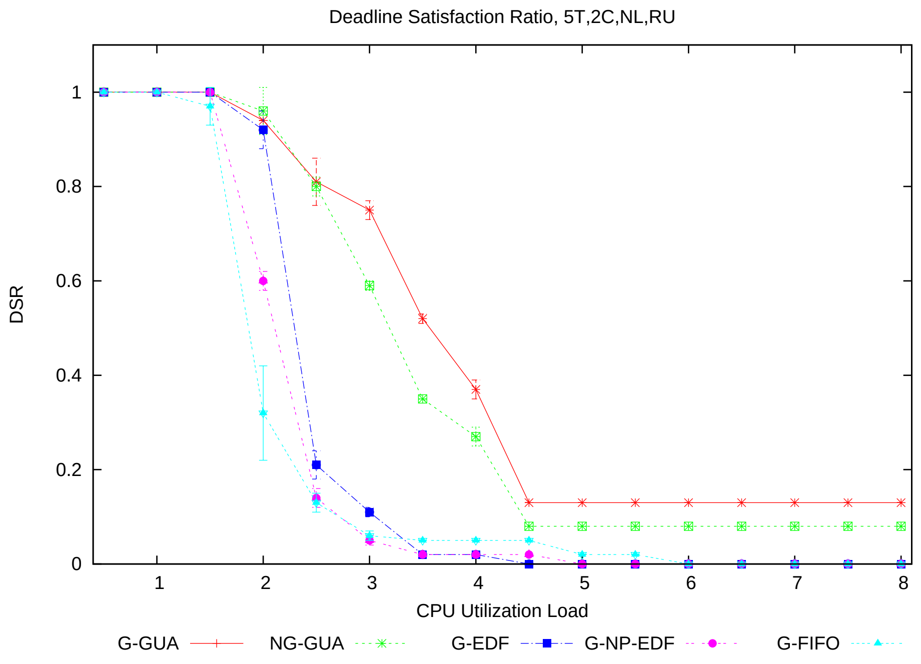


Figure 6.7: DSR vs. CPU utilization (5T, 2C, NL, RU)

and as a result G-GUA is able to schedule more high utility tasks. Also, the average AUR accrued between 250% utilization load and 800% utilization load is around 85%, which is higher than the average AUR accrued in the same range for the 5T task-set.

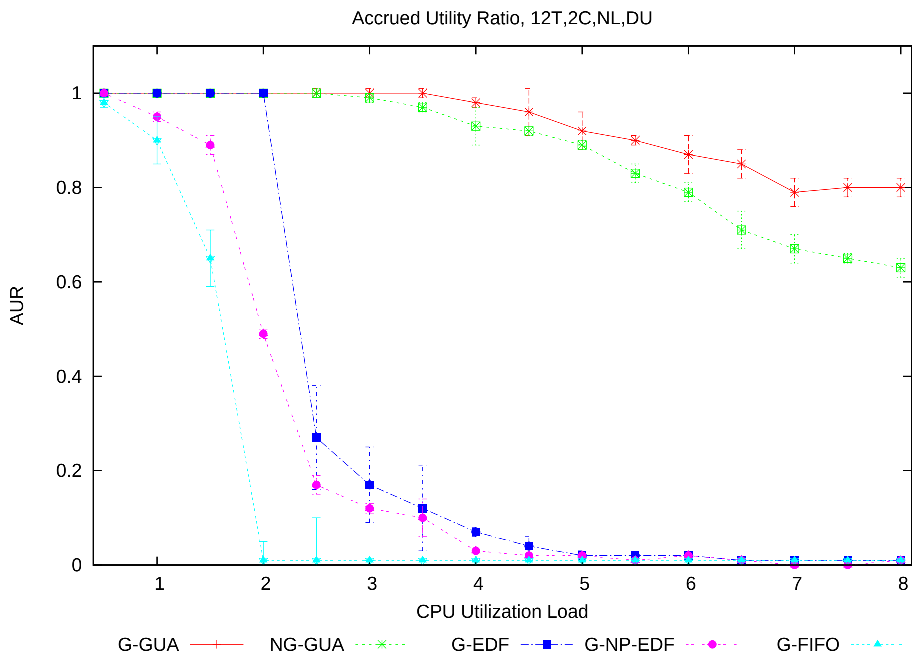


Figure 6.8: AUR vs. CPU utilization (12T, 2C, NL, DU)

We further increase the number of real-time tasks in the system and use the 27T task-set. The results are shown in Figures 6.14, 6.15, 6.16, 6.17, 6.18, and 6.19.

Figures 6.18 and 6.19 show the AUR and DSR results, respectively, for the RU model using a 27T task-set. With an increase in the number of tasks in the system, we do not see much improvement in the performance of the deadline-based scheduling algorithms. Both G-EDF and G-NP-EDF are able to meet all deadlines up until 200% ($m = 2$). During overloads, the DSR of all the deadline-based algorithms is close to zero. For 250% utilization load ($m = 2.5$), which represents a system with a light overloads, G-EDF meets around 5% of task deadlines, while NG-GUA is able to meet around 95% task deadlines. The improvement in AUR at 250% CPU utilization load is around 1800%. We notice that G-GUA performs better than NG-GUA. This is an expected behavior. However, we observe that between 250% to 650%, NG-GUA performs lower than expected. As we are using a RU model, this can be attributed to the way the utilities are assigned to the tasks. This is confirmed in the IU and DU models for the same task-set. Figure 6.16 shows the AUR results for the IU model using 27T task-set. We observe that the performance of NG-GUA is in sync with G-GUA. The same is true for the DU model as shown in Figure 6.14.

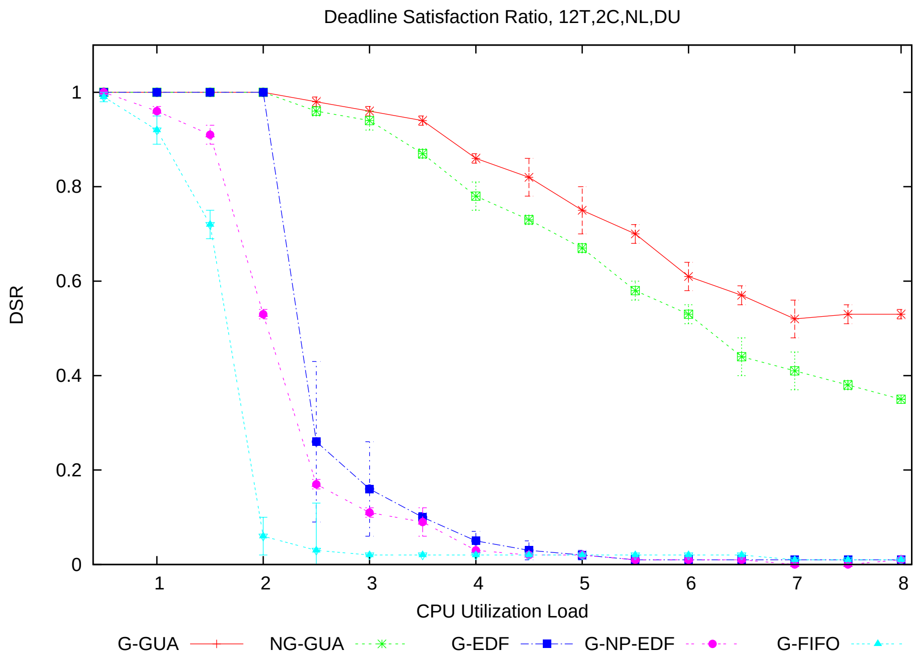


Figure 6.9: DSR vs. CPU utilization (12T, 2C, NL, DU)

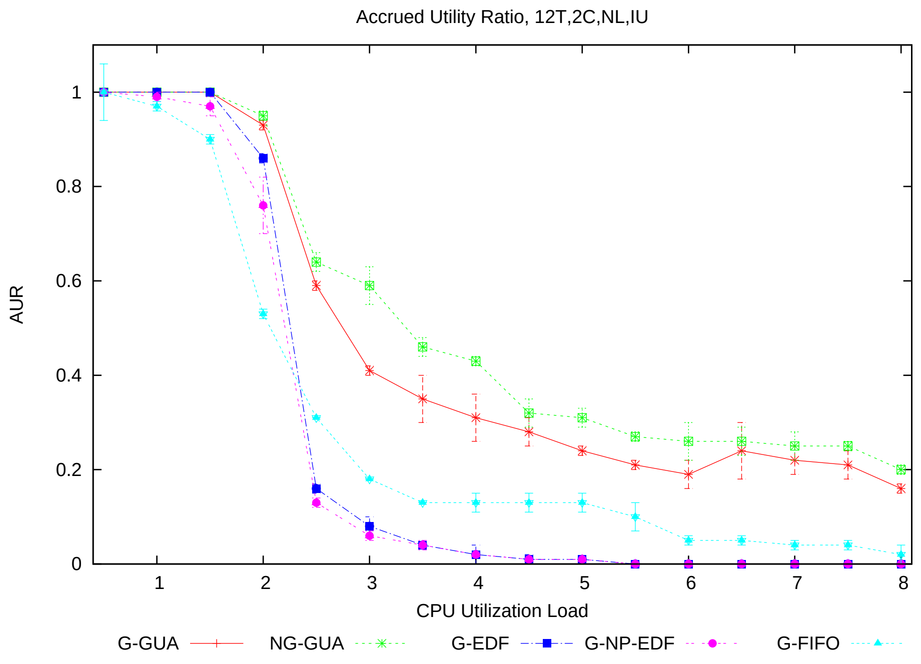


Figure 6.10: AUR vs. CPU utilization (12T, 2C, NL, IU)

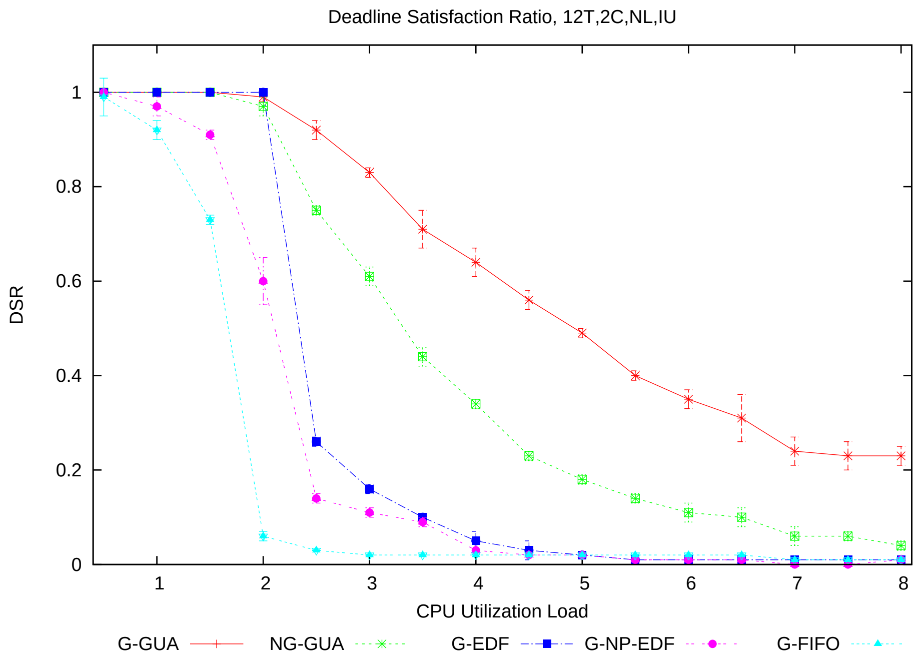


Figure 6.11: DSR vs. CPU utilization (12T, 2C, NL, IU)

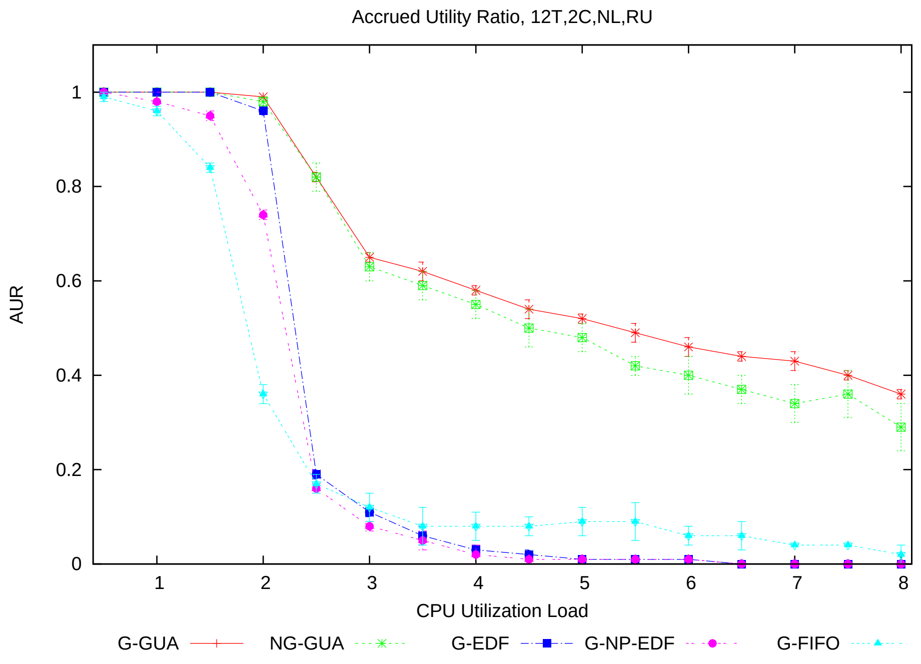


Figure 6.12: AUR vs. CPU utilization (12T, 2C, NL, RU)

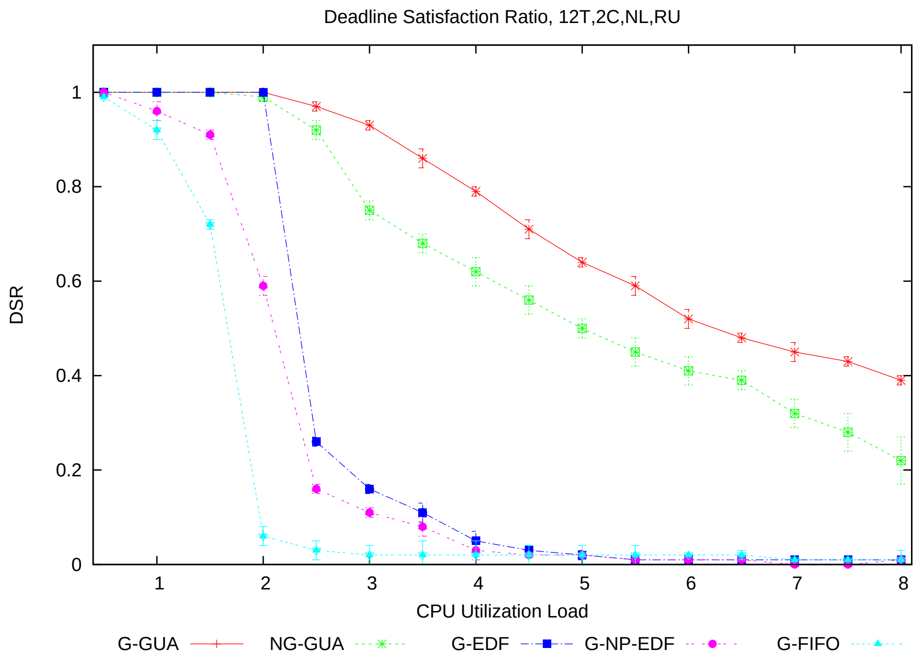


Figure 6.13: DSR vs. CPU utilization (12T, 2C, NL, RU)

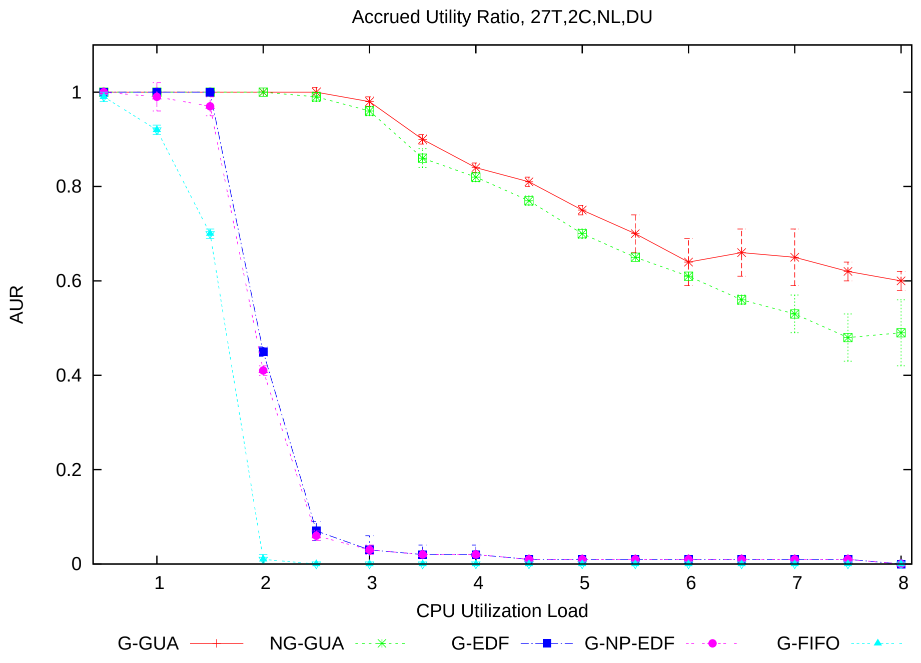


Figure 6.14: AUR vs. CPU utilization (27T, 2C, NL, DU)

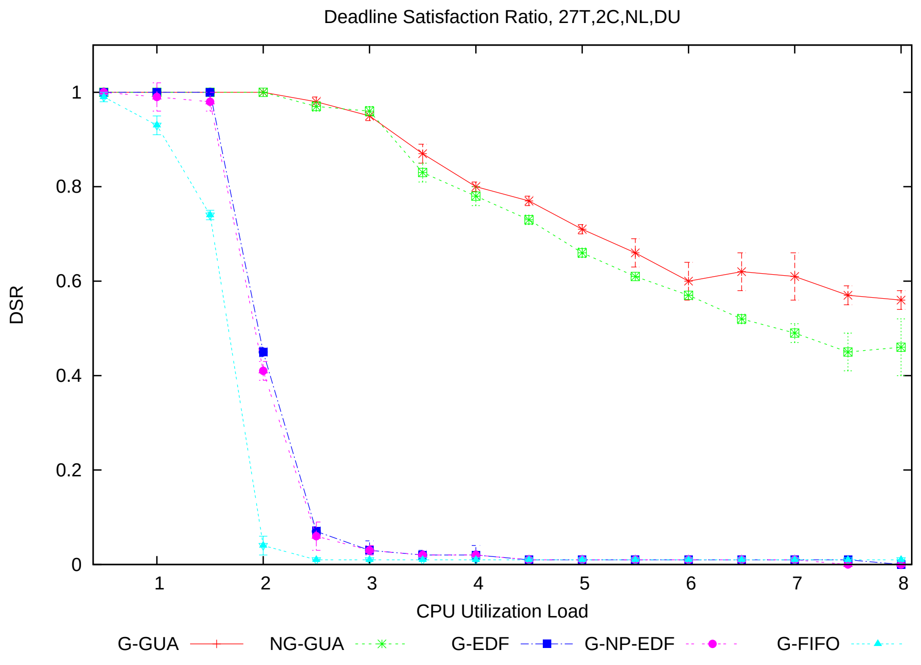


Figure 6.15: DSR vs. CPU utilization (27T, 2C, NL, DU)

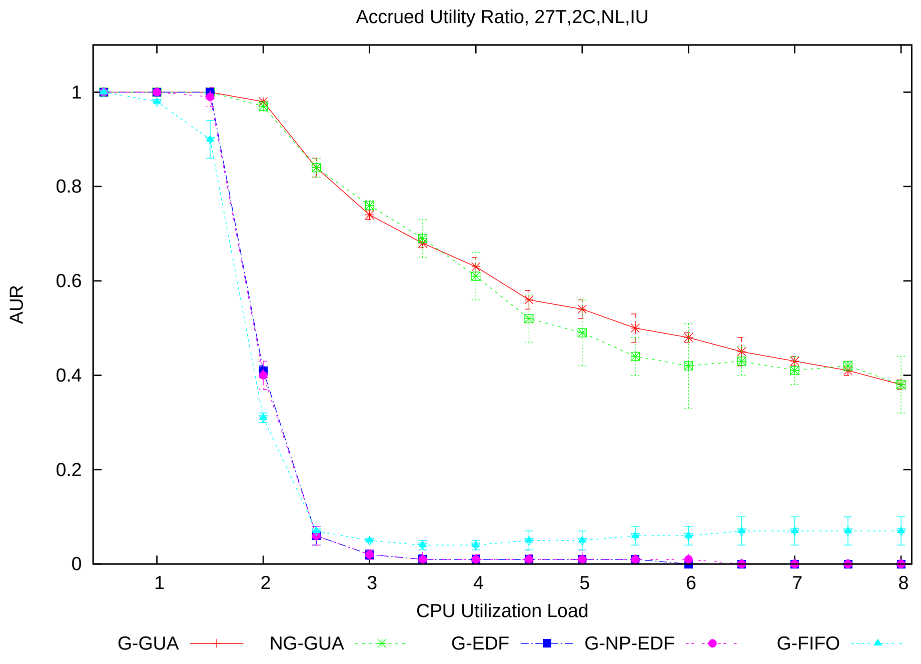


Figure 6.16: AUR vs. CPU utilization (27T, 2C, NL, IU)

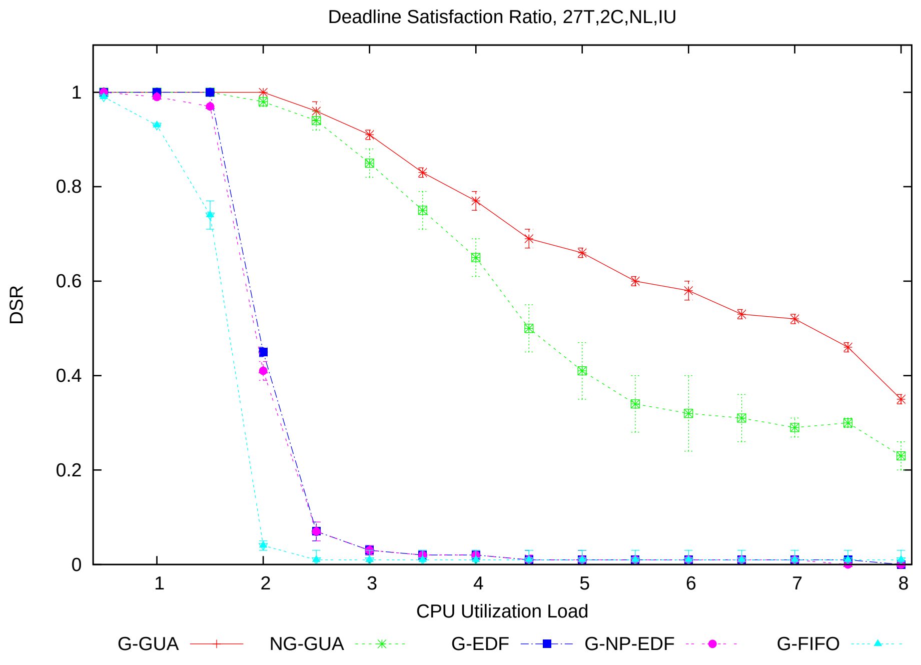


Figure 6.17: DSR vs. CPU utilization (27T, 2C, NL, IU)

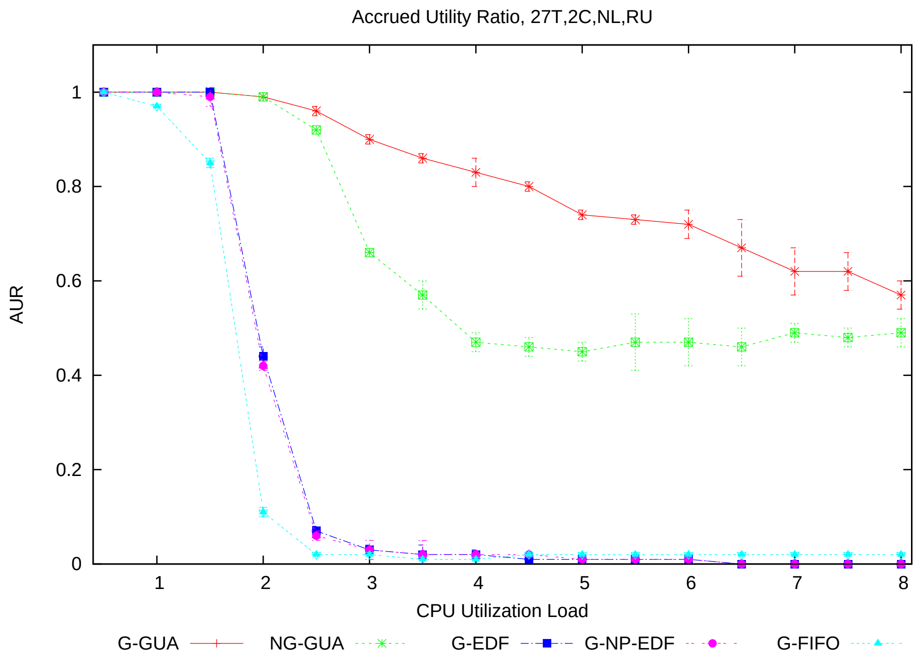


Figure 6.18: AUR vs. CPU utilization (27T, 2C, NL, RU)

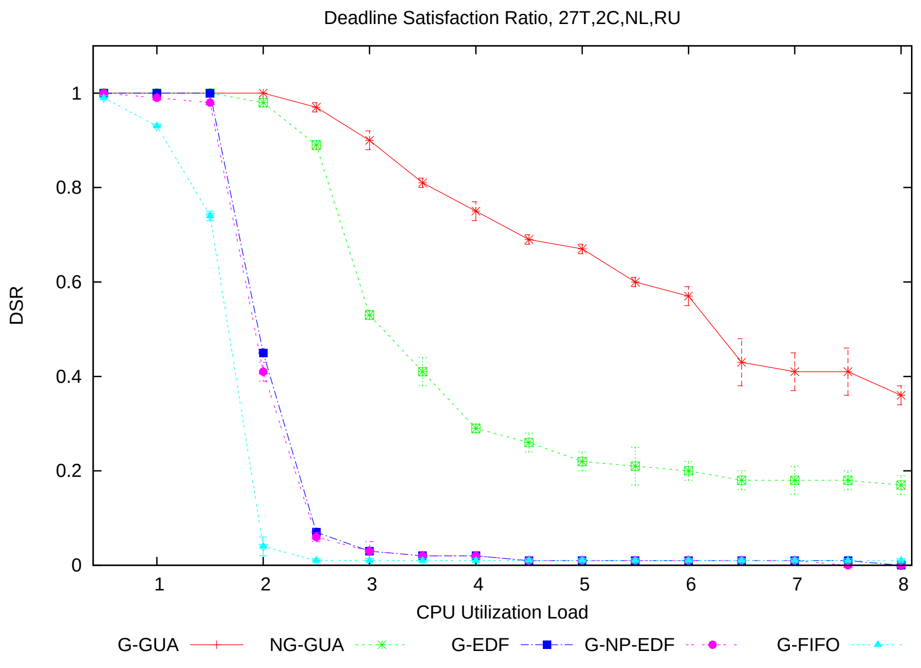


Figure 6.19: DSR vs. CPU utilization (27T, 2C, NL, RU)

#### 6.2.1.2 Four-core Platform Results

In this section, we discuss the experimental results on the four-core platform. We use the 5T, 12T and 27T task-sets using all three models. Figures 6.20 and 6.21 show the AUR and DSR results, respectively, for the DU model using the 5T task-set. Although, the total accrued utility appears to be lower for this task-set on the four-core platform when compared to the two-core results for the same task-set in Figure 6.2, the behavior can be explained. The reason for this is the relationship between the number of cores and real-time tasks used. There are 4 processors in the system and we are using a 5T task-set. This means that a deadline-based scheduling algorithm is able to assign four out of the five tasks to processors during underloads. However, during overloads, there is a much sharper decline in the DSR (and as a result the AUR) because there are not many real-time tasks left in the system to be executed when other tasks start missing deadlines. As a result, after 350% utilization load, the DSR for the deadline-based scheduling algorithms is zero. NG-GUA and G-GUA, on the other hand, are still able to find at least one task to execute.

At 400% utilization load, G-EDF, G-NP-EDF and G-FIFO meet zero deadlines. On the other hand, NG-GUA is able to schedule at least one task, while G-GUA schedules two. During underloads both G-GUA and NG-GUA try to meet deadlines and accrue as much utility as possible. NG-GUA is able to meet all the deadlines till 350% and performs better

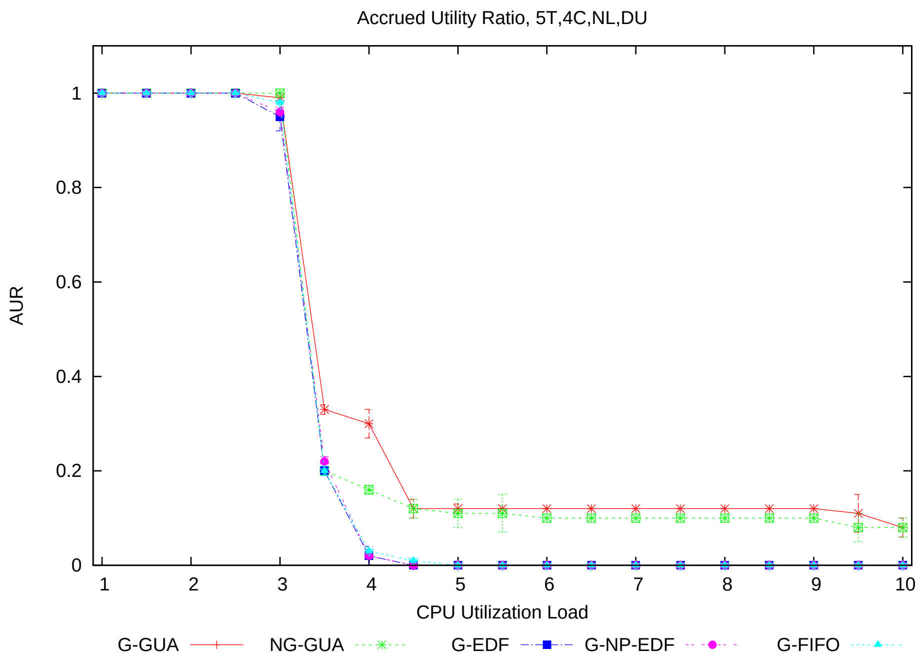


Figure 6.20: AUR vs. CPU utilization (5T, 4C, NL, DU)

than G-GUA while the latter provides better utility accrual during overloads. Figures 6.20 and 6.21 provide a good example of utility based scheduling during overloads and we have included the 5T results on the four-core platform for the same reason.

The DSR results for the IU and RU models for the 5T task-set are shown in Figures 6.23 and 6.25, respectively, while the AUR results for the IU and RU models for the 5T task-set are shown in Figures 6.22 and 6.24, respectively. As the utility assignments in IU and RU favor UA scheduling, we observe that for a similar DSR curve for IU and RU (when compared with the DU model for 5T in Figure 6.21), the system accrues better utility. This is because tasks with earlier deadlines are given smaller utilities in the IU model while the assignment of utilities in RU is totally random. This explains that given a TUF ordering such as DU, NG-GUA and G-GUA still outperforms other deadline-based scheduling algorithms. The average AUR improvement of G-GUA between the DU and the IU model is around 250%. G-GUA continues to perform better than NG-GUA in light overloads. In Figure 6.24, at 350% utilization load, G-GUA performs 5% better than NG-GUA. However at 400% utilization load, G-GUA performs 50% better than NG-GUA.

We now consider the behavior of our algorithm when the number of real-time tasks in the system are increased. We use the 12T and 27T task-set with all three TUF assignment models. Figure 6.26 and 6.27 show the AUR and DSR results for the DU model using the 12T task-set. Both G-GUA and NG-GUA perform better than the deadline-based schedulers.

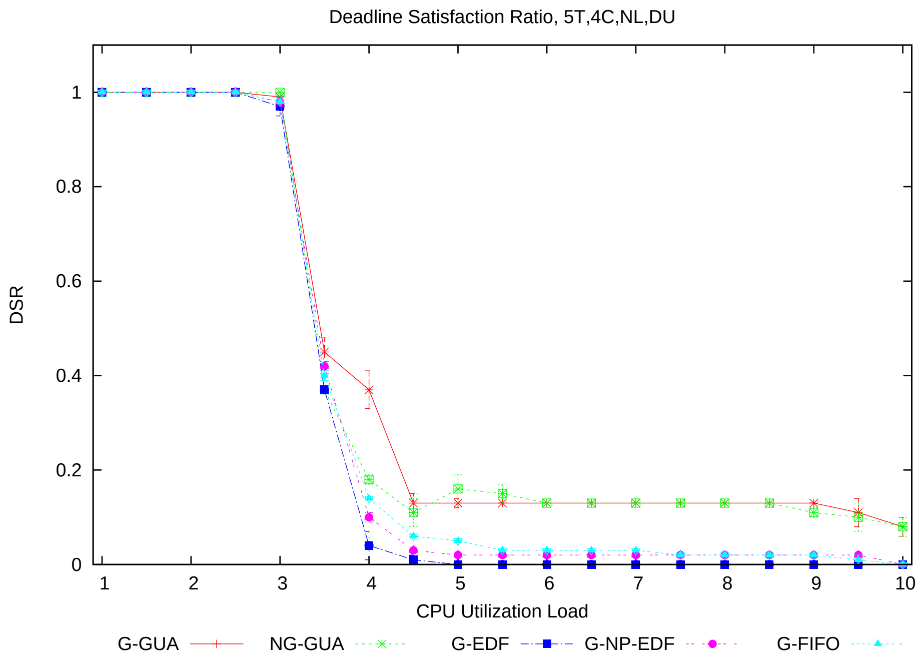


Figure 6.21: DSR vs. CPU utilization (5T, 4C, NL, DU)

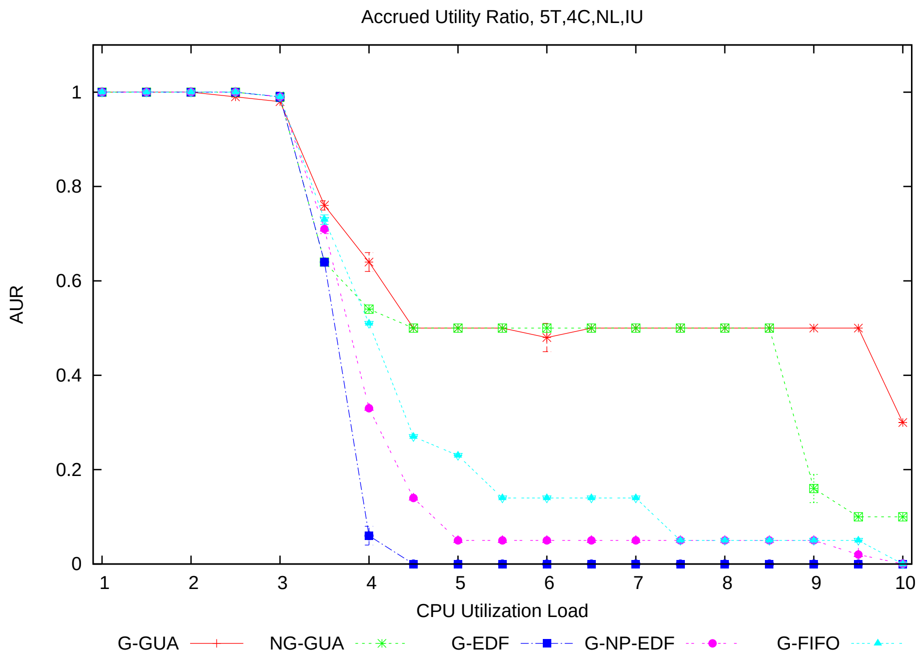


Figure 6.22: AUR vs. CPU utilization (5T, 4C, NL, IU)

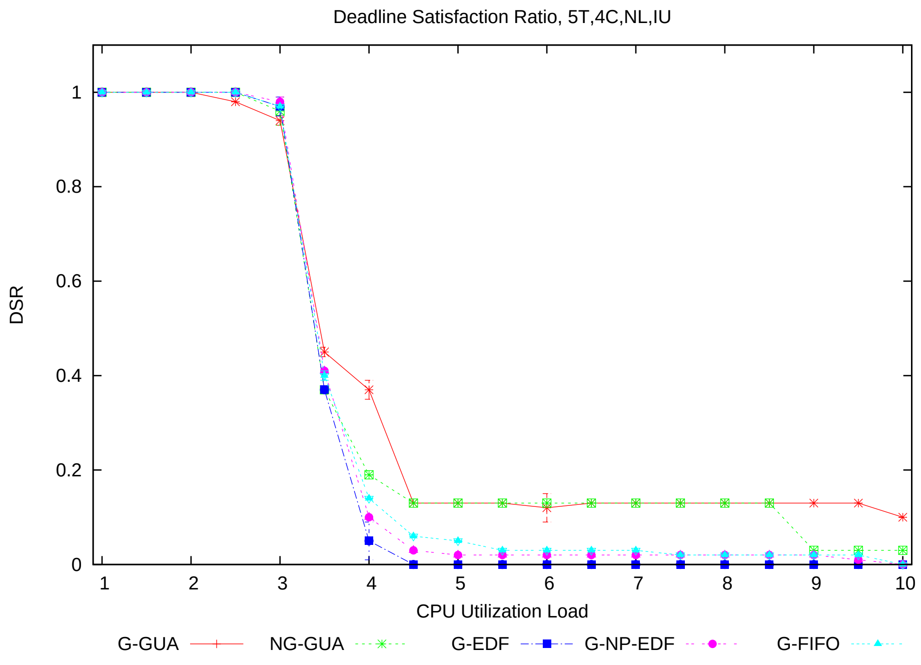


Figure 6.23: DSR vs. CPU utilization (5T, 4C, NL, IU)

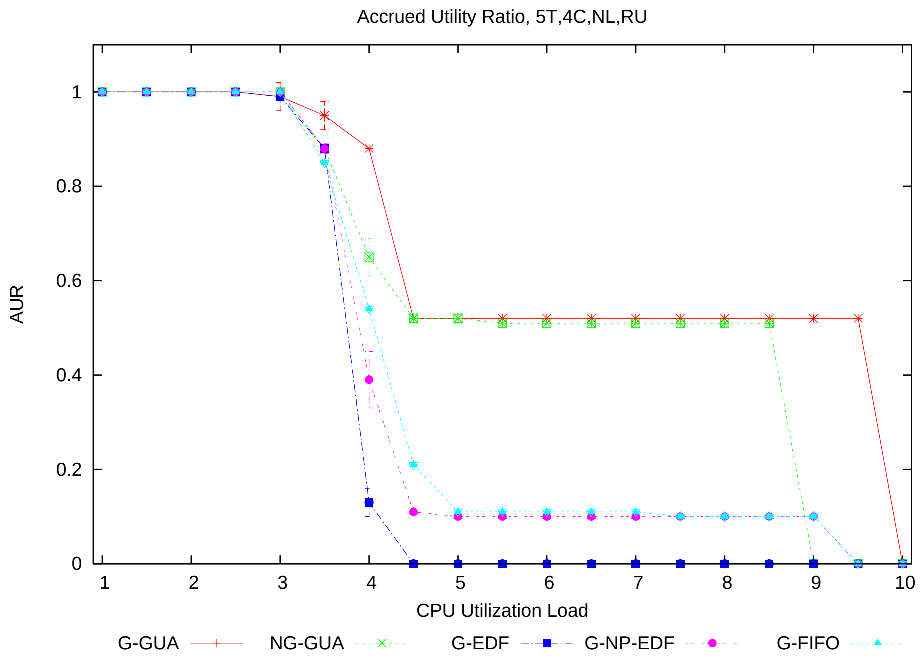


Figure 6.24: AUR vs. CPU utilization (5T, 4C, NL, RU)

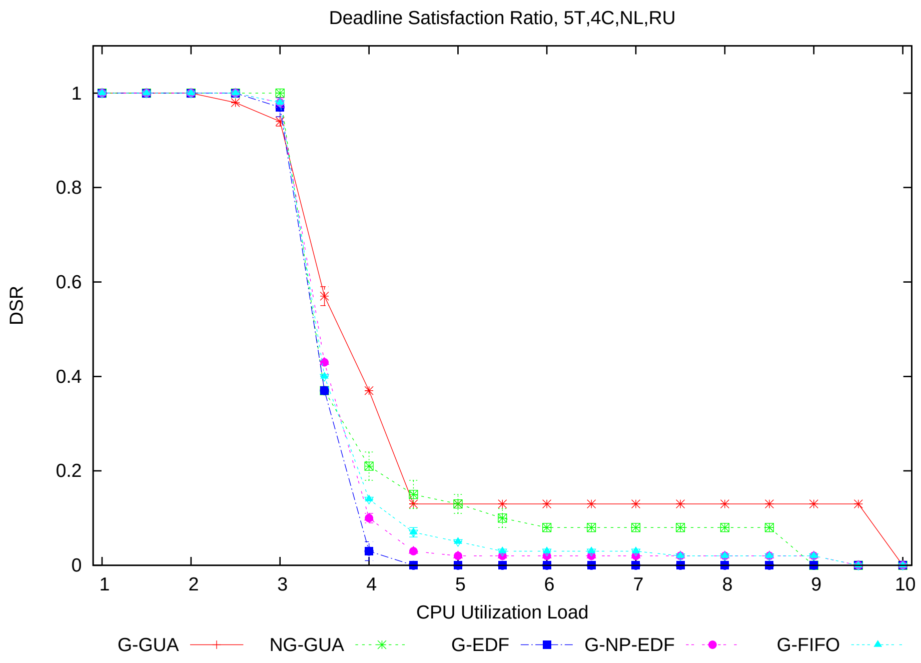


Figure 6.25: DSR vs. CPU utilization (5T, 4C, NL, RU)

In Figure 6.26, G-GUA is able to accrue 90% total utility up until 750% utilization load. NG-GUA is able to accrue around 80% total utility up until 750% utilization load. G-EDF and G-NP-EDF meet all deadlines and accrue 100% utility during underloads, but during overloads the domino effect prevents them from both meeting deadlines and accruing utility.

We observe a consistent performance improvement using all models with the 12T task-set. Figures 6.28 and 6.29 show the AUR and DSR results, respectively, for IU model using the 12T task-set. Figures 6.30 and 6.31 show the AUR and DSR results, respectively, for RU model using the 12T task-set. In the RU model, we observe that NG-GUA meets all deadlines similar to G-EDF at 350%, while G-GUA is only able to meet 95% of deadlines. However, G-GUA performs consistently better than NG-GUA from 400% utilization load to 950% utilization load with an average improvement of around 15% in the total accrued utility.

The performance with the 27T task-set is almost identical to the 12T results. Figures 6.32 and 6.33 show the AUR and DSR results, respectively, for the DU model using the 27T task-set, while Figures 6.34 and 6.35 show the AUR anad DSR results, respectively, for the IU model using the 27T task-set. G-GUA performs better than NG-GUA in the DU model with an average improvement of 15% in AUR values. Figures 6.36 and 6.37 show the AUR and DSR results, respectively, for the RU model for the 27T task-set. As the number of tasks increase, we observe an average standard deviation of around 6-8%. This amount of

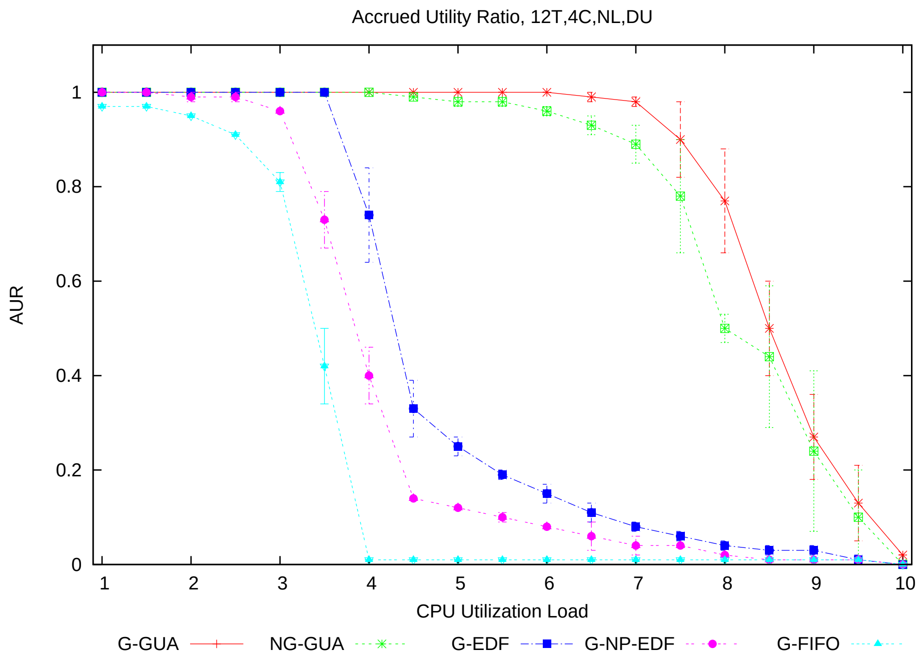


Figure 6.26: AUR vs. CPU utilization (12T, 4C, NL, DU)

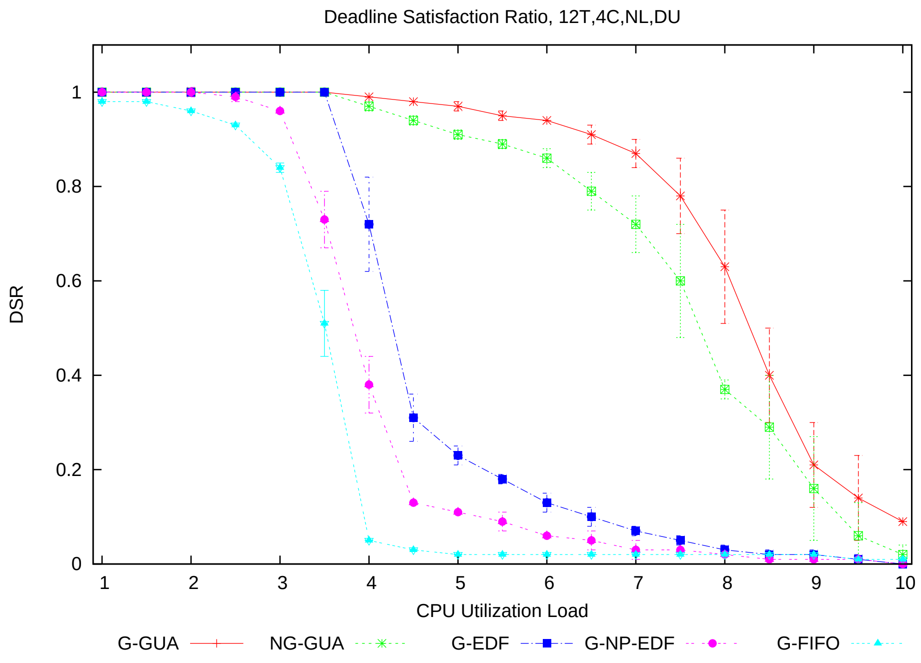


Figure 6.27: DSR vs. CPU utilization (12T, 4C, NL, DU)

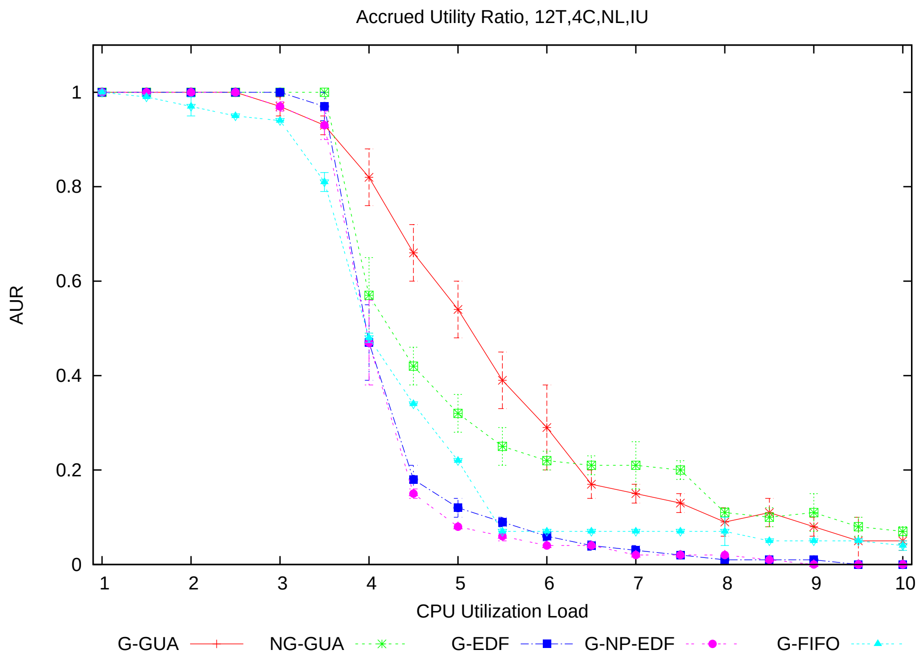


Figure 6.28: AUR vs. CPU utilization (12T, 4C, NL, IU)

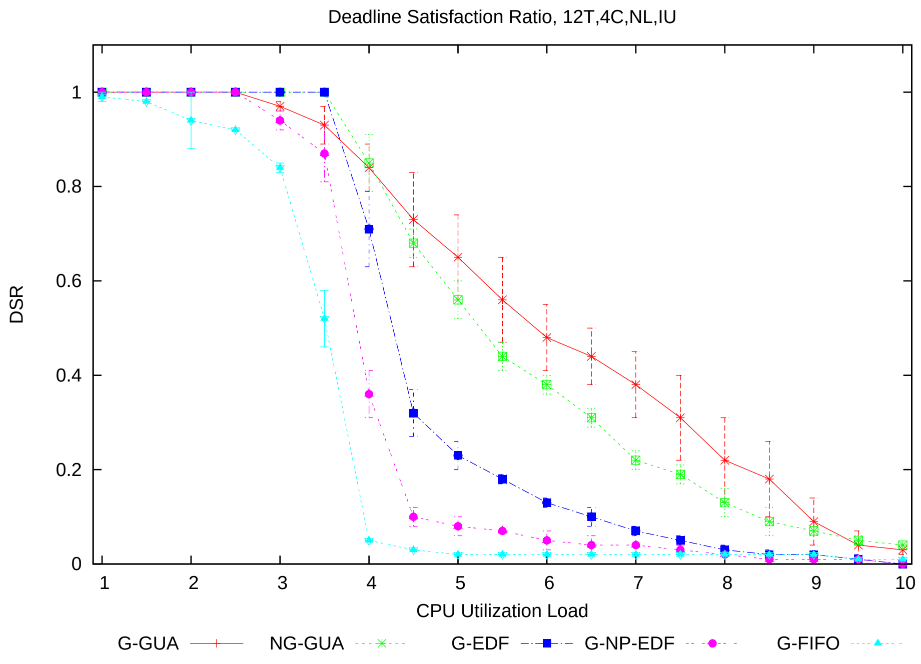


Figure 6.29: DSR vs. CPU utilization (12T, 4C, NL, IU)

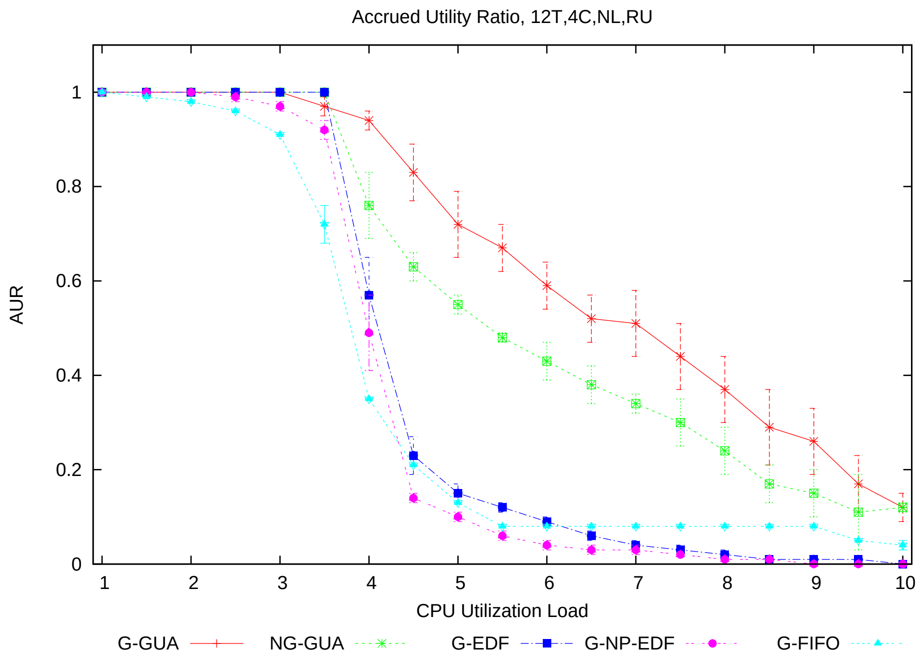


Figure 6.30: AUR vs. CPU utilization (12T, 4C, NL, RU)

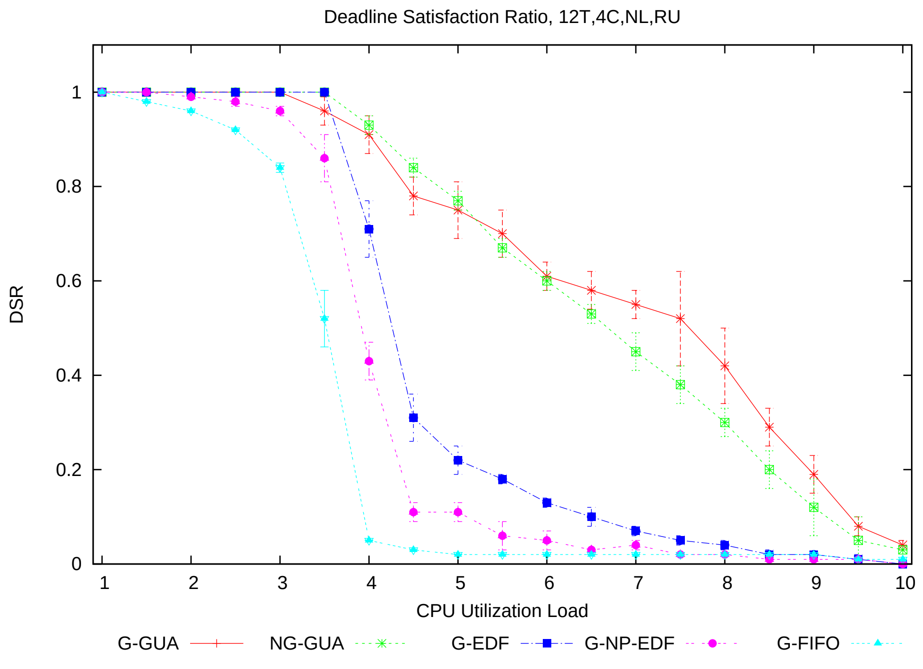


Figure 6.31: DSR vs. CPU utilization (12T, 4C, NL, RU)

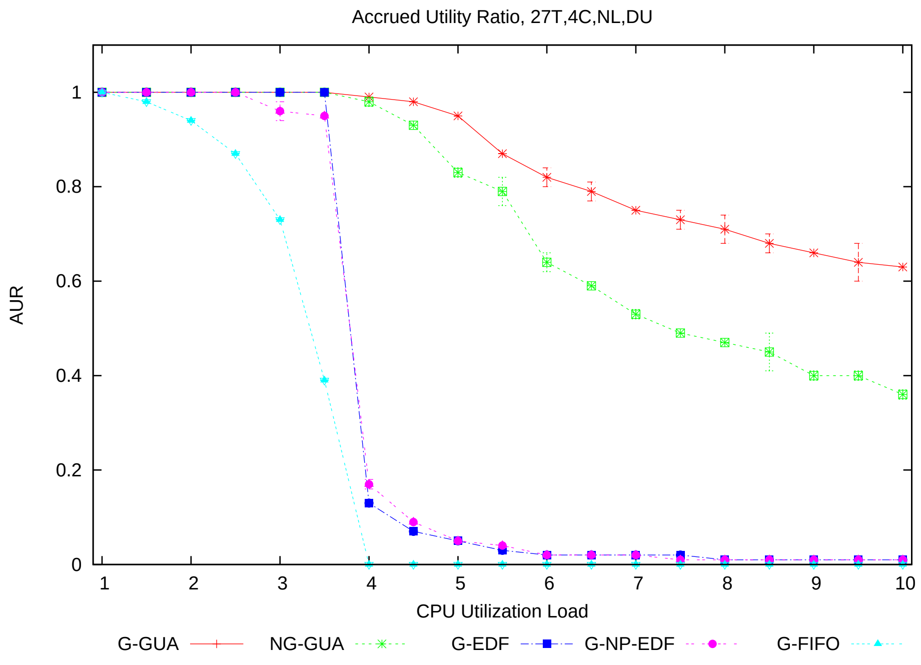


Figure 6.32: AUR vs. CPU utilization (27T, 4C, NL, DU)

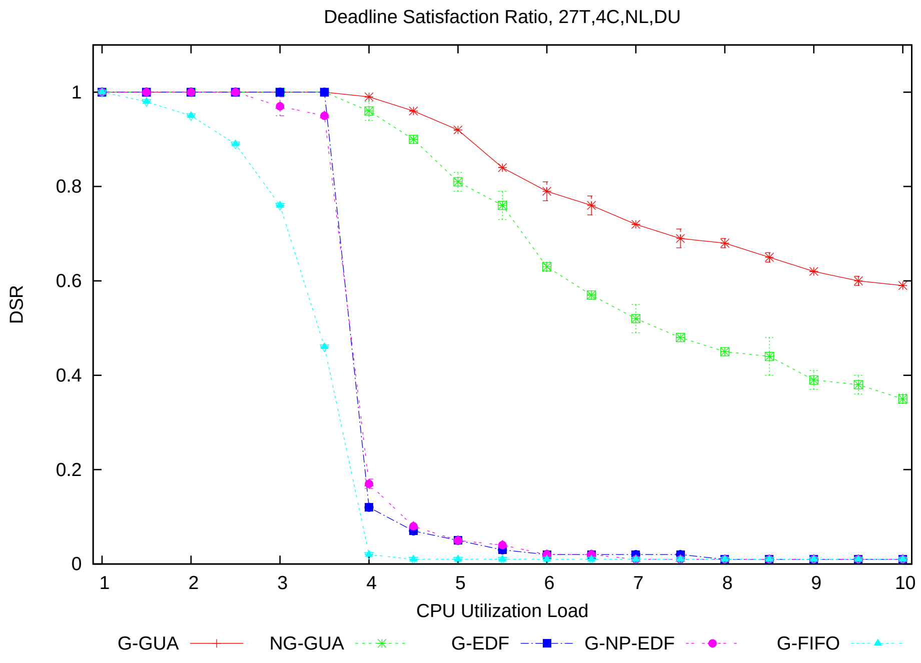


Figure 6.33: DSR vs. CPU utilization (27T, 4C, NL, DU)

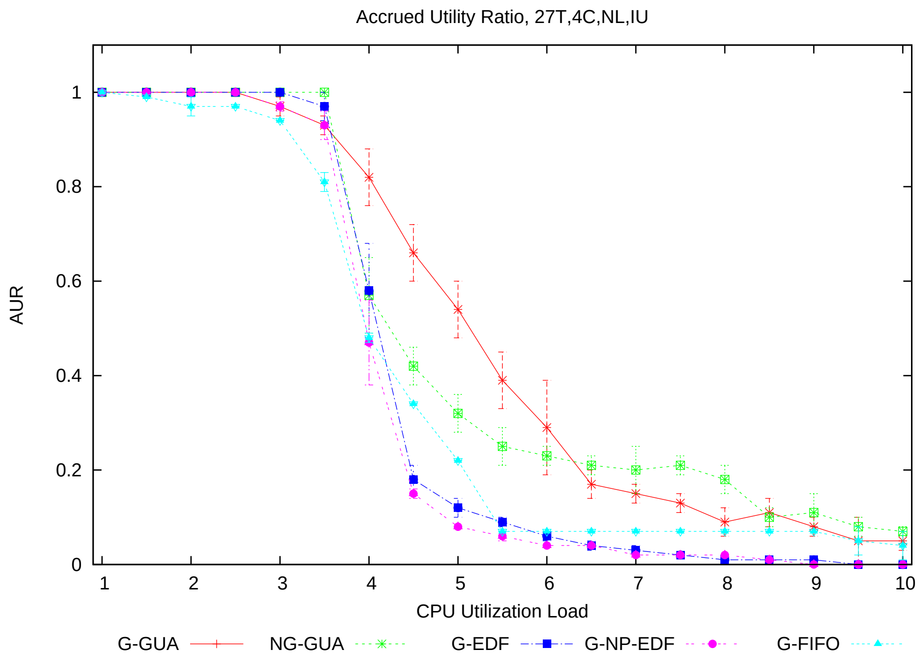


Figure 6.34: AUR vs. CPU utilization (27T, 4C, NL, IU)

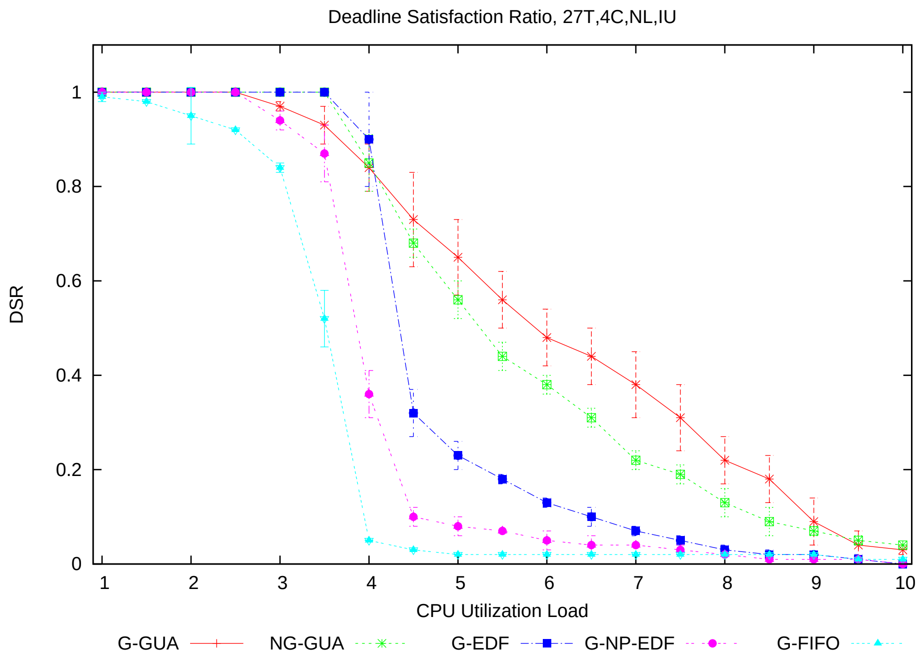


Figure 6.35: DSR vs. CPU utilization (27T, 4C, NL, IU)

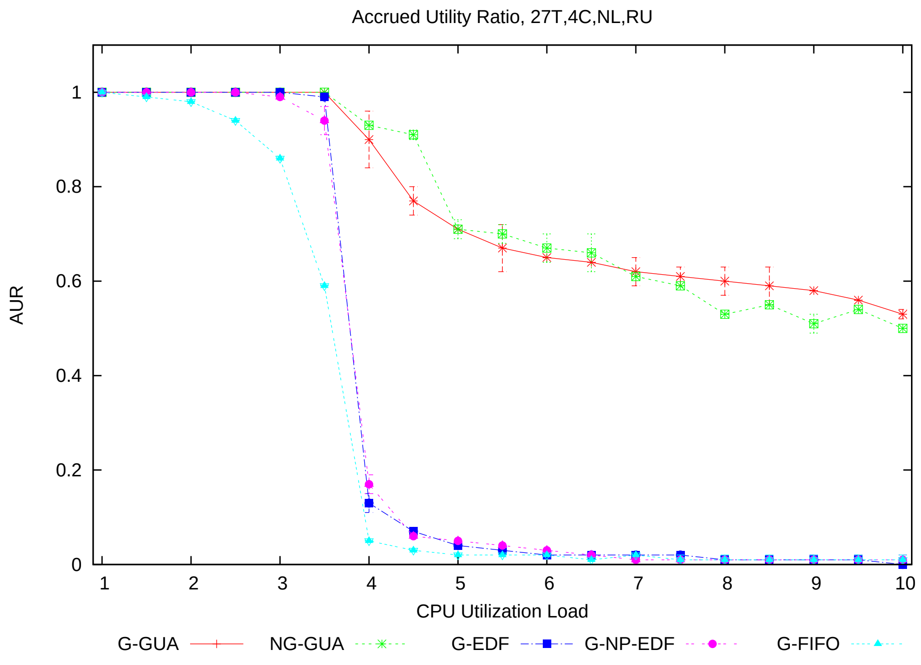


Figure 6.36: AUR vs. CPU utilization (27T, 4C, NL, RU)

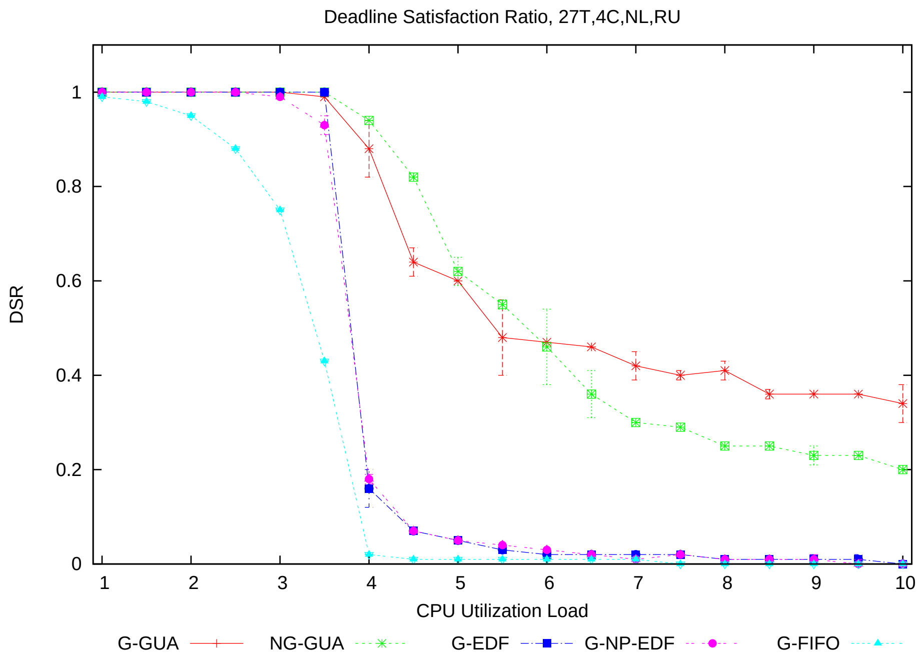


Figure 6.37: DSR vs. CPU utilization (27T, 4C, NL, RU)

variance is acceptable given the fact that it is impossible to have the same operating system environment for every test execution. Due to the `PREEMPT_RT` patch, ChronOS is able to withstand jitter from non real-time based applications. However, as mentioned earlier, there are critical operating system primitives that have higher priority in ChronOS which are required for the proper functioning of the operating system, such as the timer interrupts and the I/O interrupts.

#### 6.2.1.3 Eight-core Platform Results

In order to establish the performance of both NG-GUA and G-GUA, we move to a higher platform. The eight-core platform uses two Intel Xeon E5520 quad-core processors which have four physical cores, thus supporting four hardware threads. The processors provide Hyper-Threading (HT) using which we can simulate eight logical processors. However, ChronOS does not support HT. As a result, we need to disable HT and use two Intel Xeon E5520 quad-core processors to create an eight-core platform with eight hardware threads.

With the increase in the number of processors, we avoid using the 5T and 12T task-sets, as the number of tasks in these task-sets do not scale up to the number of processors used. Instead, we use the 27T and 50T task-sets. Using a 50T task-set allows us to ascertain if our scheduling algorithms incur any scalability issues and also provides a better perspective to the experimental results on the eight-core platform.

Figures 6.38 and 6.39 show the AUR and DSR results, respectively, with the DU model using the 27T task-set. As we are using an eight-core platform, we observe that G-EDF is able to meet all deadlines up until 780% utilization load while G-NP-EDF is able to meet all deadlines up until 750% utilization load. These results are as expected and meet the schedulability tests for both these algorithms [10]. G-FIFO, on the other hand, starts missing deadlines at 400% utilization load and at full system load (800%), it has already degraded to around 10% deadline satisfaction ratio.

During overloads, both G-NP-EDF and G-EDF start missing deadlines rapidly and reach to a 20% deadline satisfaction ratio during the light overload of 900% utilization load. G-FIFO, on the other hand, does not meet any deadlines during overloads. When compared to the deadline-based algorithms, G-GUA and NG-GUA not only provide a better deadline satisfaction ratio, but also accrue greater utility in the system. In Figure 6.38, we observe that G-GUA at the CPU utilization load of 900% provides around 99% accrued utility. This measures to an overall improvement of around 400% in total accrued utility. However, at 1200% CPU utilization load, G-GUA provides a 95% accrued utility, which measures to an overall improvement of around 1800% in total accrued utility. NG-GUA performs better than the deadline-based algorithms. However, G-GUA outperforms NG-GUA, as expected, by an average of 5-10% across 800-1200% CPU utilization load.

In Figures 6.40 and 6.41 we show the AUR and DSR results, respectively, for IU model using

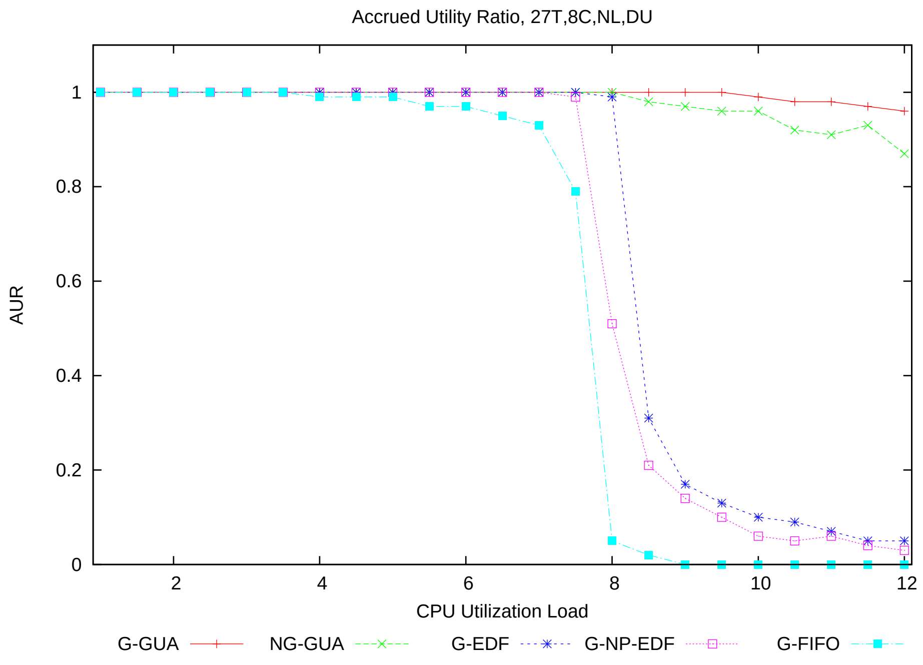


Figure 6.38: AUR vs. CPU utilization (27T, 8C, NL, DU)

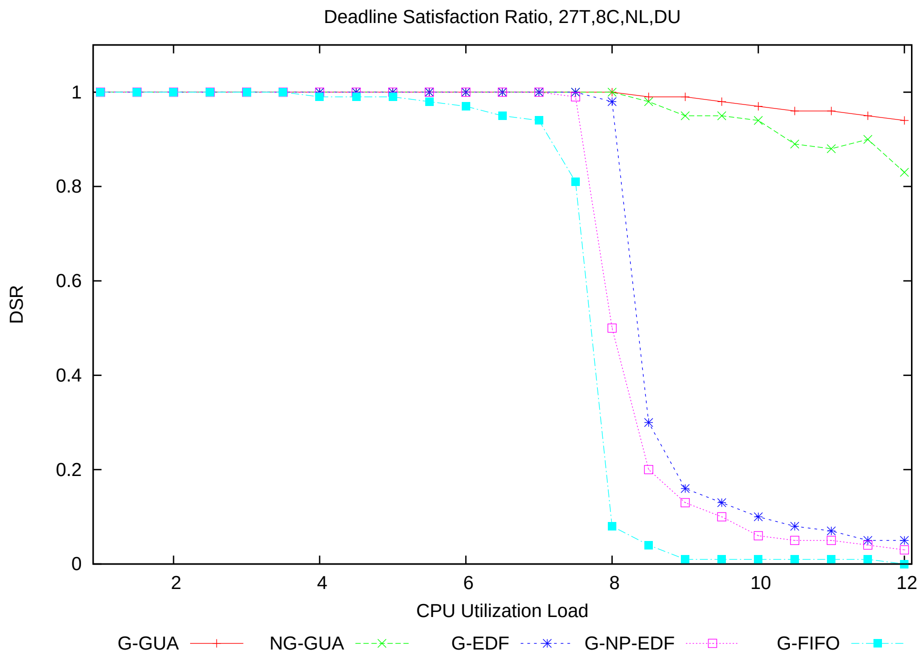


Figure 6.39: DSR vs. CPU utilization (27T, 8C, NL, DU)

the 27T task-set. The results are similar to the DU model. However, we observe that NG-GUA is able to meet all deadlines during underloads and defaults to the G-EDF behavior. In Figures 6.42 and 6.43 we show the AUR and DSR results, respectively, for the RU model using 27T task-set. The standard deviation of the results on eight-core are in the range of $0.01 - 0.03$, which show that the results are consistent across multiple runs.

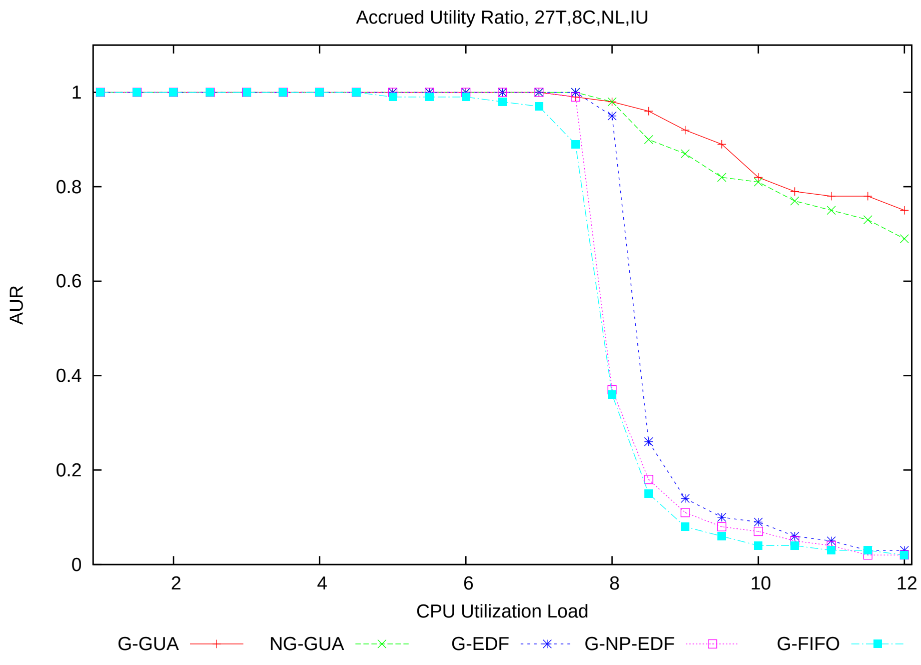


Figure 6.40: AUR vs. CPU utilization (27T, 8C, NL, IU)

We now consider the effect of increasing the number of tasks. We use the 50T task-set with all the three models. Figures 6.44, 6.46 and 6.48 show the AUR results for DU, IU and RU models, respectively, using 50T task-set while, Figures 6.45, 6.47 and 6.49 show the DSR results for DU, IU and RU models, respectively, using the 50T task-set.

We observe that the performance of NG-GUA and G-GUA is consistent with the DU model results of 27T task-set shown in Figure 6.38. However, with the increase in the number of tasks, there is a slight drop in the total accrued utility at certain CPU utilization loads. In Figure 6.48 at 900% CPU utilization load, we observe an accrued utility of 94%. At the same load on the 27T task-set, we observe an accrued utility of around 99%, which is a drop of around 5%. However at 1200% CPU utilization load, we observe an accrued utility of 85% with the 27T task-set and around 83% with the 50T task-set, which is a drop of around 3%. We cover this is detail in Section 6.4, where we present the scheduling overheads for NG-GUA and G-GUA as a function of increased number of tasks and CPU utilization load.

Overall, we observe that the performance of both NG-GUA and G-GUA outperforms that

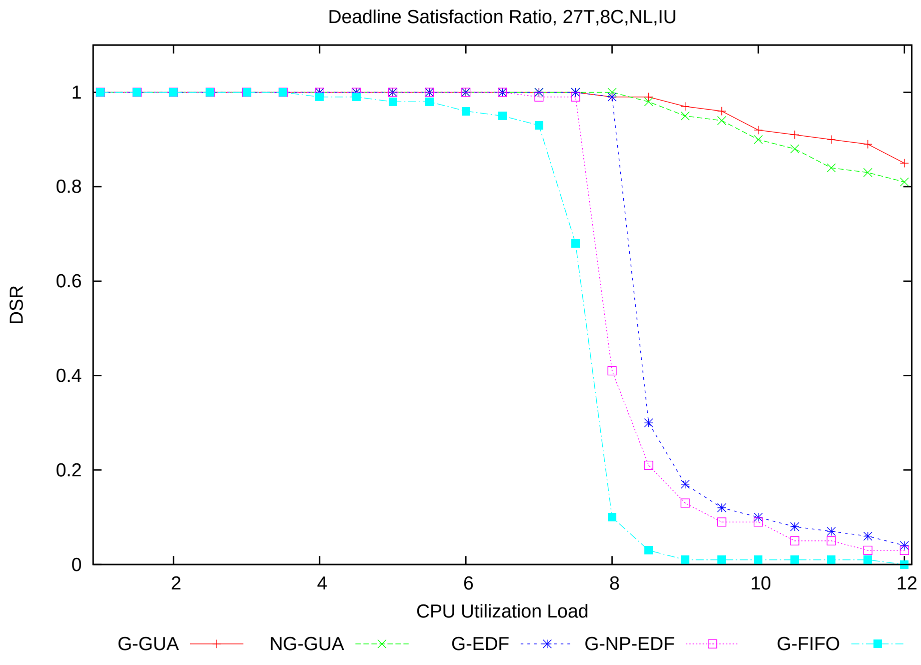


Figure 6.41: DSR vs. CPU utilization (27T, 8C, NL, IU)

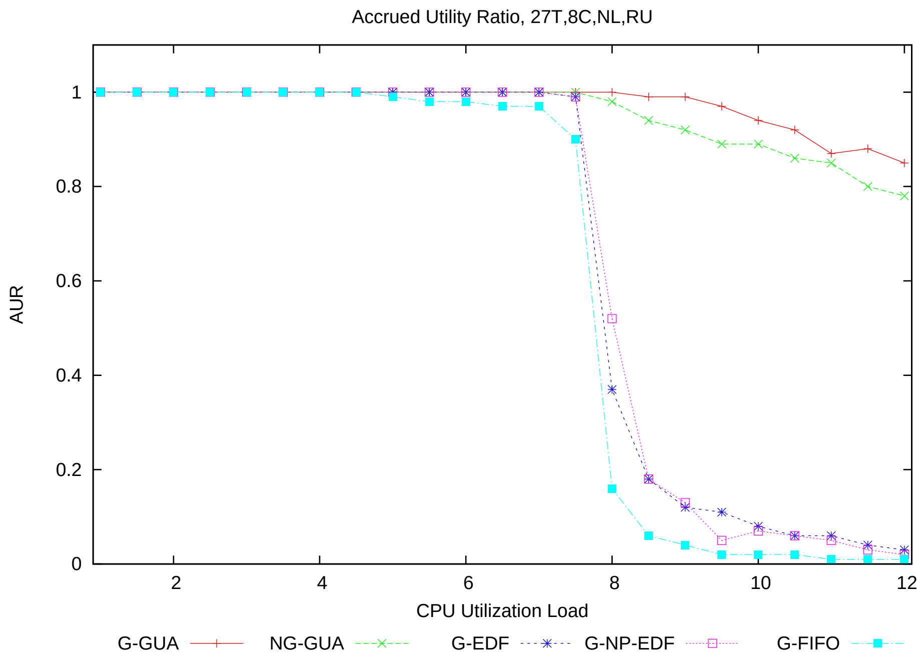


Figure 6.42: AUR vs. CPU utilization (27T, 8C, NL RU)

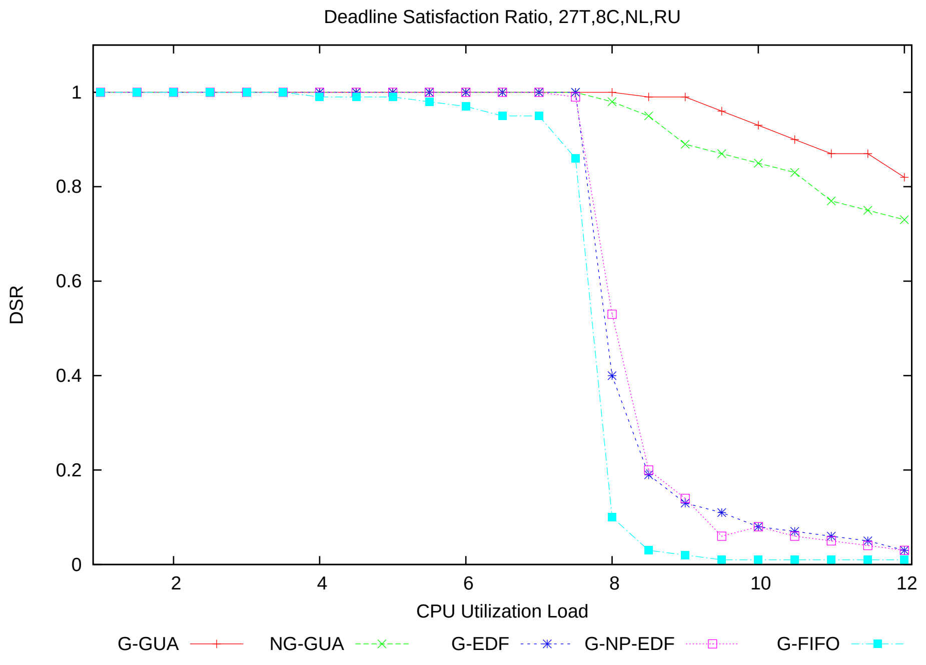


Figure 6.43: DSR vs. CPU utilization (27T, 8C, NL, RU)

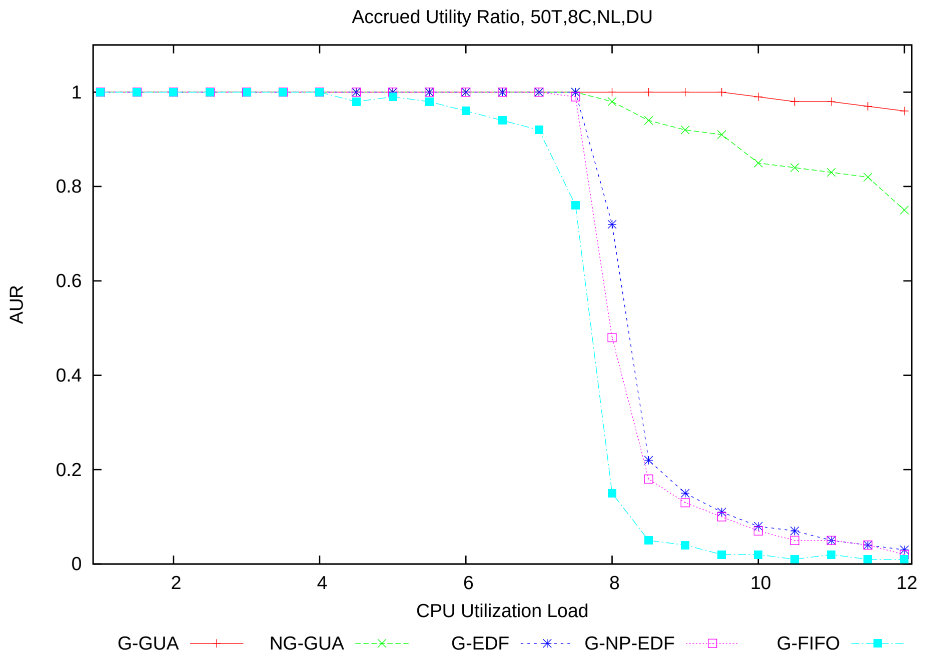


Figure 6.44: AUR vs. CPU utilization (50T, 8C, NL, DU)

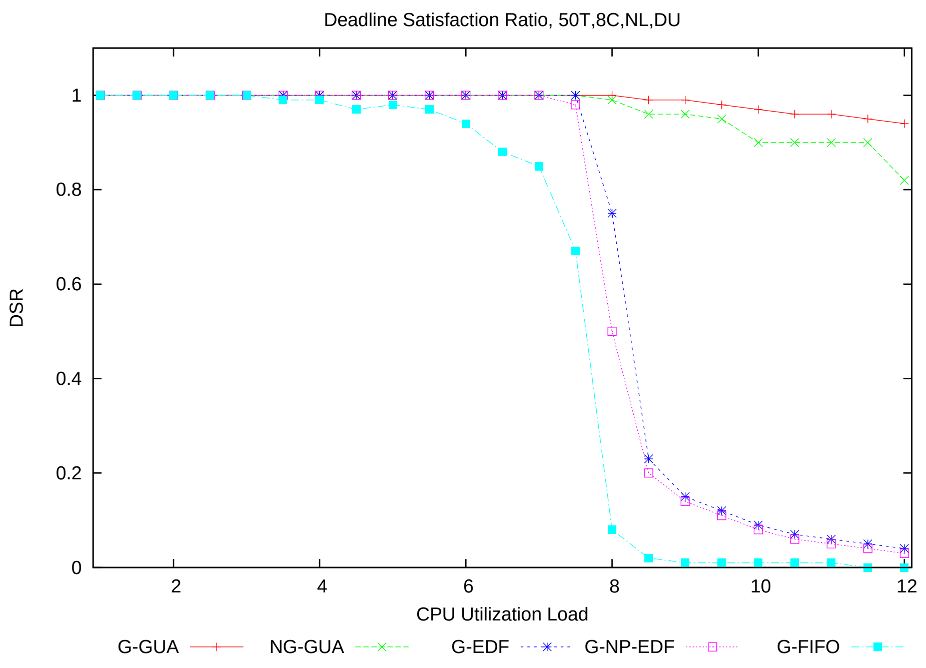


Figure 6.45: DSR vs. CPU utilization (50T, 8C, NL, DU)

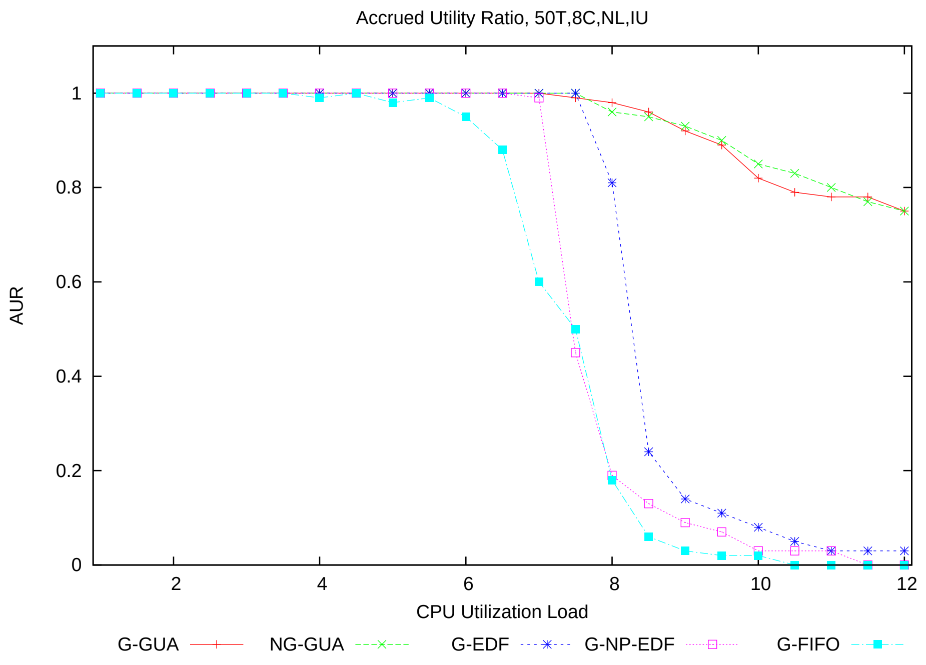


Figure 6.46: AUR vs. CPU utilization (50T, 8C, NL, IU)

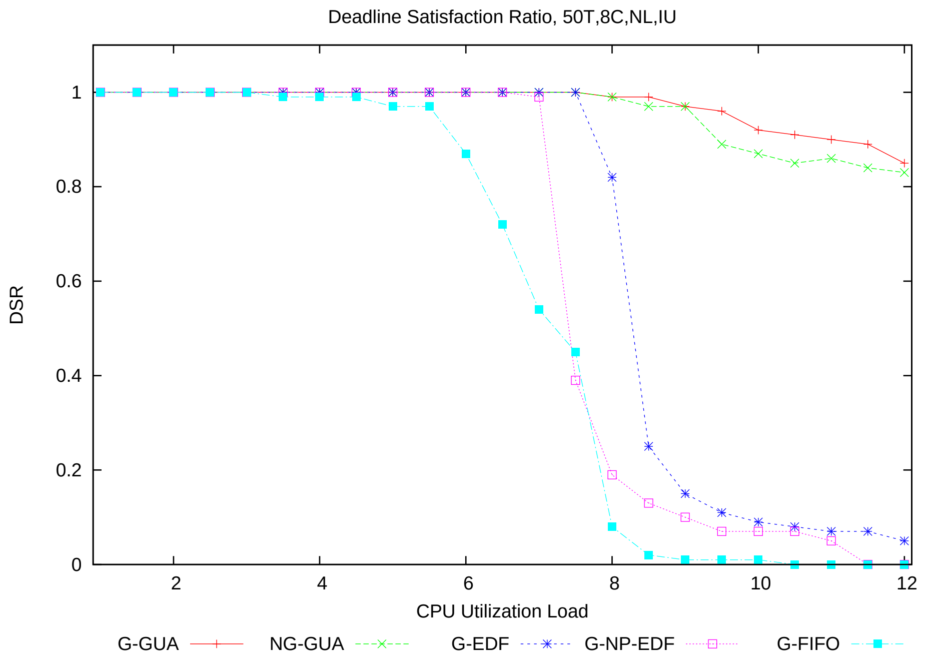


Figure 6.47: DSR vs. CPU utilization (50T, 8C, NL, IU)

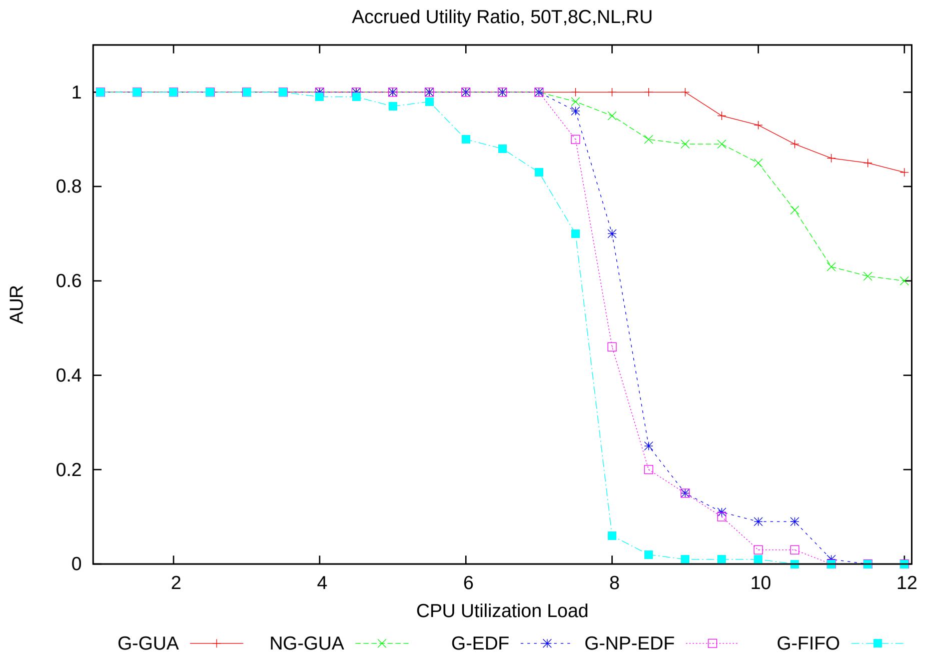


Figure 6.48: AUR vs. CPU utilization (50T, 8C, NL, RU)

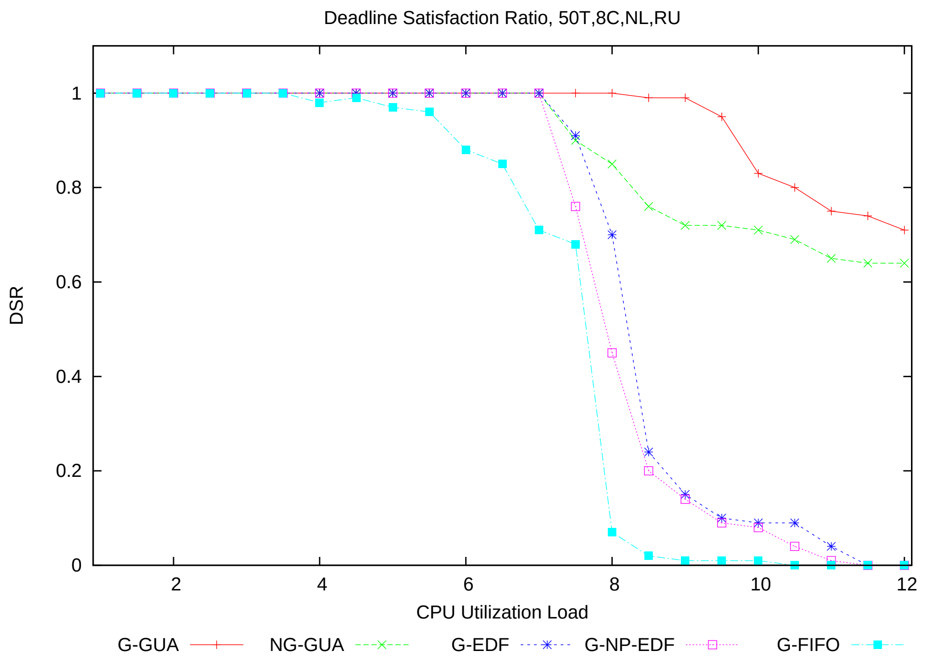


Figure 6.49: DSR vs. CPU utilization (50T, 8C, NL, RU)

of the deadline-based scheduling algorithms, such as G-EDF and G-NP-EDF. Between the two UA scheduling algorithms, NG-GUA provides a better deadline satisfaction ratio than G-GUA during underloads. This is due to the fact that NG-GUA defaults to a G-EDF-like behavior. However during overloads, NG-GUA accrues better utility than G-EDF. As gMUA defaults to NG-GUA for the non-dependent case, its performance is similar to NG-GUA. G-GUA, on the other hand, outperforms NG-GUA (and gMUA) in accrued utility during overloads and provides an average improvement of about 15% over NG-GUA/gMUA across CPU utilizations.

### 6.2.2 Comparison with Partitioned Scheduling Algorithms

In this section we compare NG-GUA and G-GUA with partitioned scheduling algorithms. We consider the two state-of-the-art competitors P-EDF and P-DASA. The partitioned scheduling algorithms work differently than the global scheduling algorithms. While global scheduling algorithms provide an on-line scheduler where a single ready queue is maintained for all the real-time tasks across processors; partitioned scheduling algorithms, on the other hand, use an off-line bin-packing heuristic to divide the tasks into processor bins. The tasks are then assigned to the processors, where each processor runs the single processor variant of the algorithm and schedules only the tasks that have been assigned to it. As a result, partitioned schedulers may or may-not incur less overheads than their global variants.

We compare NG-GUA and G-GUA with P-EDF and P-DASA to find if, even with global scheduling overheads, our algorithms compare in performance with the partitioned algorithms. P-EDF runs EDF on each of the processors. EDF uses the real-time tasks that have been assigned to it using the off-line partitioning heuristic. As EDF, on a single processor, is optimal for $U \leq 1$, P-EDF tries to mimic EDF behavior to meet as many deadlines as possible. However, during overloads, P-EDF starts missing deadlines rapidly. In a similar fashion, P-DASA runs the UA scheduling algorithm DASA on each of the processors, wherein DASA tries to accrue greater overall utility on each of the processors.

In order to partition the tasks on to the processors, we implement Baruah's first-fit partitioning algorithm [9] to divide the task-sets off-line into processor bins. The algorithm uses the EDF schedulability criteria and assigns tasks to processors such that the resultant task-set is feasible on that processor. Once the partitioned task-sets are created, they are set a processor affinity.

We perform the comparisons on a four-core platform and consider two types of task-sets with an increasing number of tasks.

1. The first task-set uses 12 tasks with deadlines/periods in the range of $[300ms - 20000ms]$ and the utilization load for each task in the range of $[0.01 - 0.4]$.
2. The second task-set uses 27 tasks with deadlines/periods in the range of $[50ms - 7500ms]$ and the utilization load for each task in the range of $[0.01 - 0.3]$.

The task-sets do not use any locks. We assign random utilities to the tasks in the range of $[100 - 5000]$. The task-set is used "as-is" with the global scheduling algorithms (NG-GUA and G-GUA), while it is partitioned using Baruah's first-fit algorithm for the partitioned scheduling algorithms. Table 6.3 shows the legend used in the experimental results.

Table 6.3: Legend used in experimental results for comparison with other partitioned scheduling algorithms

| Symbol | Description |
|---|---|
| `RU` | Random Utility |
| `NL` | No locks used |
| `BF` | Baruah's first-fit partition [9] |
| `xC` | Experiment using $x$ number of processors |
| `xT` | Experiment using $x$ number of tasks |

Figures 6.50 and 6.51 show the AUR and DSR results, respectively, for the RU model using a partitioned 12T task-set for partitioned algorithms and the "as-is" 12T task-set for global

scheduling algorithms. From Figure 6.51, we observe that P-EDF is able to meet all deadlines till 375% utilization load. As we are using a four-core platform, the result is in accordance with the schedulability criteria of P-EDF [2]. However, during overloads, P-EDF starts missing deadlines and by 550% utilization load, we observe that P-EDF meets only around 5% deadlines. This is primarily due to the domino effect of EDF based algorithms during overloads.

P-DASA, on the other hand, does not meet all its deadlines during underloads, as expected [31]. DASA is a utility accrual scheduling algorithm that does not default to EDF during underloads. As a result, we observe that P-DASA starts missing deadlines at around 300% utilization load. On the other hand, P-DASA performs better than P-EDF during overloads and it not only meets more deadlines that P-EDF, but also provides better accrued utility.

However, when compared with NG-GUA and G-GUA, the total accrued utility of P-DASA is lower. At 400% utilization load, P-DASA provide a total accrued utility of around 70%, while both NG-GUA and G-GUA provide a 100% accrued utility, which is a performance improvement of around 40%. For higher utilization loads, both NG-GUA and G-GUA accrue higher utilities with G-GUA outperforming every other scheduler. At 700% utilization load, G-GUA has around 150% improvement in total accrued utility. The vertical errors bars show the variance in the results, the reason for which has been discussed in the earlier sections and remains the same.

Figures 6.52 and 6.53 show the AUR and DSR results, respectively, for RU model using the 27T partitioned task-set for the partitioned algorithms and “as-is” 27T task-set for the global scheduling algorithms. P-EDF behaves as expected and starts missing deadlines during overloads. P-DASA accrues utility during overloads and has a better AUR compared to P-EDF. NG-GUA and G-GUA perform better than P-DASA during overloads. At 450% utilization load, NG-GUA has an improvement of around 50% in accrued utility over P-DASA. Between 550% to 800% utilization load, both NG-GUA and G-GUA have an average improvement of 60% in accrued utility over P-DASA. The results show some variability in the total accrued utility of NG-GUA and G-GUA with a standard deviation of around 6-8%. As mentioned earlier, this amount of variability is expected due to the various operating system primitives. Between 400% and 600% utilization loads, we observe that NG-GUA performs better than G-GUA by around 4%. This can be attributed to issues with cache-warm up or other I/O interruptions inside ChronOS.

Overall, we observe that the deadline-based partitioned scheduling (such as P-EDF) do not perform well during overload conditions. On the other hand, the partitioned UA based scheduling algorithms (such as P-DASA) perform better than the deadline variant. However, G-GUA and NG-GUA outperform the partitioned algorithms in total accrued utility during overloads.

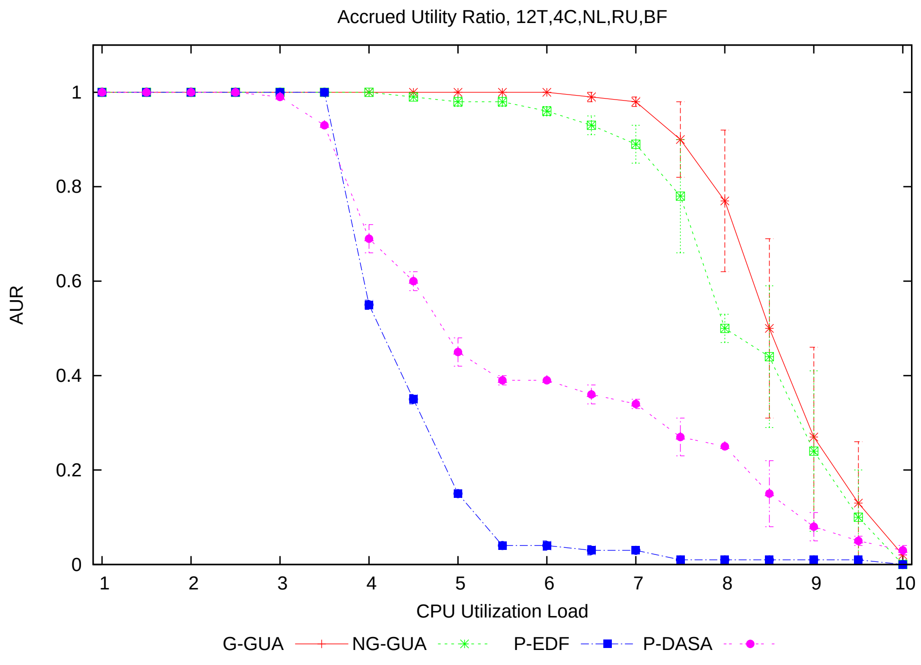


Figure 6.50: AUR vs. CPU utilization (12T, 4C, NL, RU, BF)

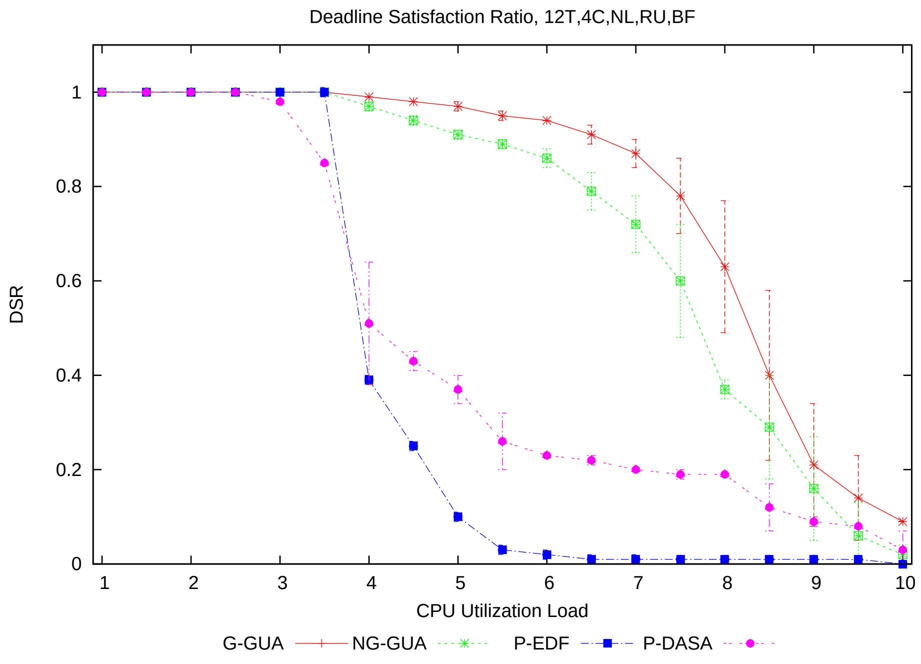


Figure 6.51: DSR vs. CPU utilization (12T, 4C, NL, RU, BF)

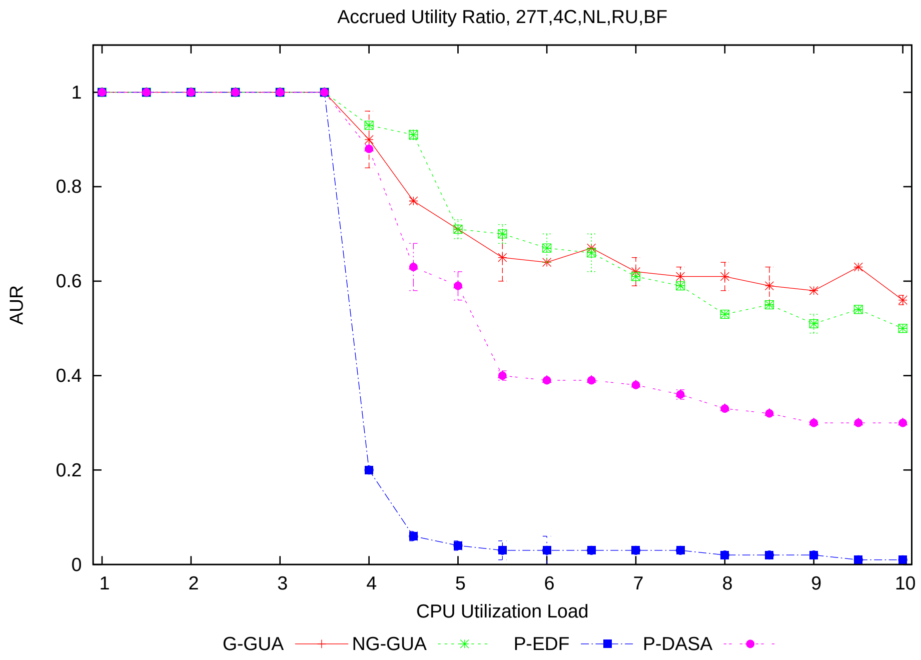


Figure 6.52: AUR vs. CPU utilization (27T, 4C, NL, RU, BF)

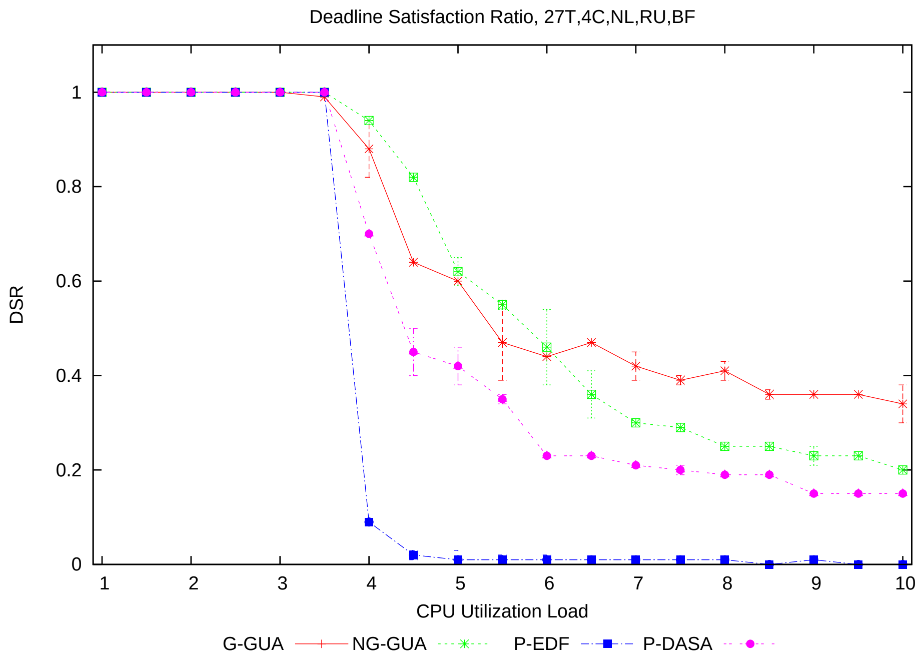


Figure 6.53: DSR vs. CPU utilization (27T, 4C, NL, RU, BF)

## 6.3 Results With Dependencies

In this section we present the experimental results of our evaluation of NG-GUA and G-GUA in the presence of dependencies. We compare the results with PIP based algorithms, such as G-FIFO-PIP, GNP-EDF-PIP.

We use locks as a means of creating contention between tasks and use the following models:

- Fixing the number of locks per task and the critical section length for each lock but varying the total utilization load.
- Fixing the total utilization load and the number of locks per task but varying the critical section length for each lock.
- Fixing the total utilization load and the critical section length for each lock but varying the number of locks per task.

In ChronOS, we implement locks as `futexes`, which allows us to share a context from the kernel-space to the user-space. The test application uses the ChronOS system call to request for a lock. This is done using the `do_vt_rt_mutex(struct mutex_data __user *mutexreq, int operation)` system call. The `operation` can be a `MUTEX_REQUEST` or `MUTEX_RELEASE`. The application provides the `futex` information using the `mutexreq` data-structure.

We compare NG-GUA and G-GUA with the other PIP enabled scheduling algorithms on the four-core platform. We consider a 12T base task-set that creates 12 tasks with deadlines/periods in the range of $[150ms - 3000ms]$; with the utilization load for each task in the range of $[0.01 - 0.4]$. We consider the RU model and assign random utilities to the tasks in the range of $[100 - 5000]$.

The base task-set creates a total of 12 locks in the system which are distributed amongst individual tasks in the task-set in order to create four different variants.

**1 lock per task** - We distribute the locks in the 12T task-set such that each task gets to request for a single lock.

**2 locks per task** - We distribute the locks in the 12T task-set such that each task gets to request two locks.

**3 locks per task** - We distribute the locks in the 12T task-set such that each task gets to request three locks.

**4 locks per task** - We distribute the locks in the 12T task-set such that each task gets to request four locks.

Note that more than one task can request the same lock. The assignment of locks in the task-set is done randomly using an automatic task-set generator application. Table 6.4 shows the legend used in experimental result.

Table 6.4: Legend used in experimental results for comparison with other global scheduling algorithms in the presence of dependencies

| Symbol | Description |
|---|---|
| `RU` | Random Utility |
| `xL` | Experiment using $x$ number of locks per task |
| `xCS` | Experiment using critical section length equal to $x$ percent of total task execution cost |
| `xUt` | Experiment using $x$ utilization load |
| `xC` | Experiment using $x$ number of processors |
| `xT` | Experiment using $x$ number of tasks |

### 6.3.1 Varying CPU utilization Load

In this section we discuss the experimental results of running NG-GUA and G-GUA in the presence of dependencies. The results are compared with other deadline-based scheduling algorithms that handle resource using the priority inheritance protocol. Using PIP, if the task $J_a$ with higher priority is blocked on a resource $R_i$, which is being owned by a task $J_b$ that has a lower priority, the scheduler bumps the priority of task $J_b$ to that of task $J_a$. This is done to allow $J_b$ to finish execution and release the resource. For the deadline-based scheduler, the priorities can be mapped to the deadlines. A task with an earlier deadline is the most eligible task in the system. If that task is blocked on a resource owned by another task that has a later deadline, the scheduler allows the latter task to be executed such that the resource is released and made available.

In this experiment, we desire to find the effect of increasing CPU utilization load on the accrued utility in the presence of locks. We consider the following combinations:

1. We use the 12T task-set with 1 lock per task and fix the critical section length of the lock to 5% of the task's worst case execution cost, while varying the CPU utilization load from 100-1000%.
2. We use the 12T task-set with 1 lock per task and fix the critical section length of the lock to 25% of the task's worst case execution cost, while varying the CPU utilization load from 100-1000%.

3. We use the 12T task-set with 4 locks per tasks and fix the total critical section length of the locks to 5% of the task's worst case execution, while varying the CPU utilization load from 100-1000%.
4. We use the 12T task-set with 4 locks per tasks and fix the total critical section length of the locks to 25% of the task's worst case execution, while varying the CPU utilization load from 100-1000%.

Figures 6.54 and 6.55 show the AUR and DSR results, respectively, for the RU model with 12T task-set with one lock and 5% critical section length. Even with a 5% critical section length and a single lock, we observe that both G-NP-EDF and G-FIFO are not able to meet most of their deadlines during underloads. During overloads, they continue to miss deadlines and as a result the overall accrued utility of the system comes down. Both NG-GUA and G-GUA perform much better than the deadline-based scheduler, with an average improvement of around 120% in the total accrued utility of the system. As we increase the critical section length to 25%, we observe in Figures 6.56 and 6.57 that total accrued utility of NG-GUA and G-GUA falls downs. However, the AUR is still higher than the deadline-based scheduling algorithms. GNP-EDF starts losing deadlines as early as 200% CPU utilization load. On the other hand, between 200% to 400% utilization load, G-GUA shows an improvement in AUR by around 40%.

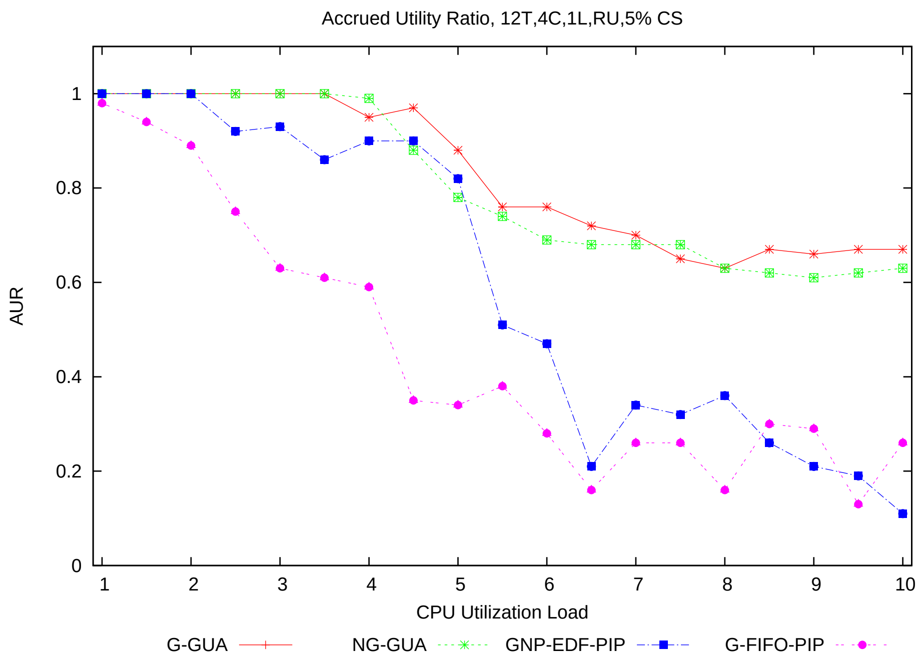


Figure 6.54: AUR vs. CPU utilization (12T, 4C, 1L, RU, 5% CS)

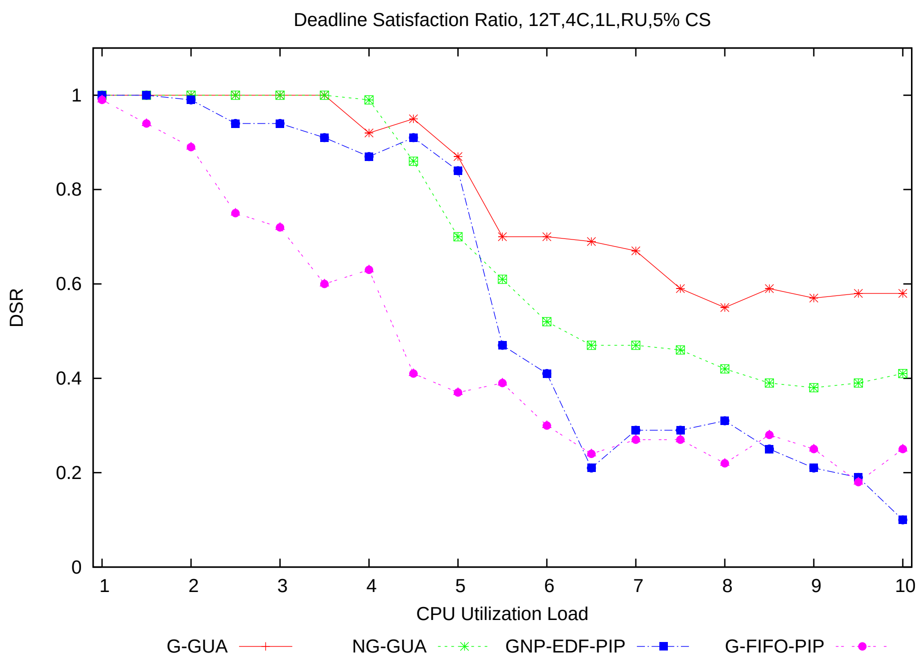


Figure 6.55: DSR vs. CPU utilization (12T, 4C, 1L, RU, 5% CS)

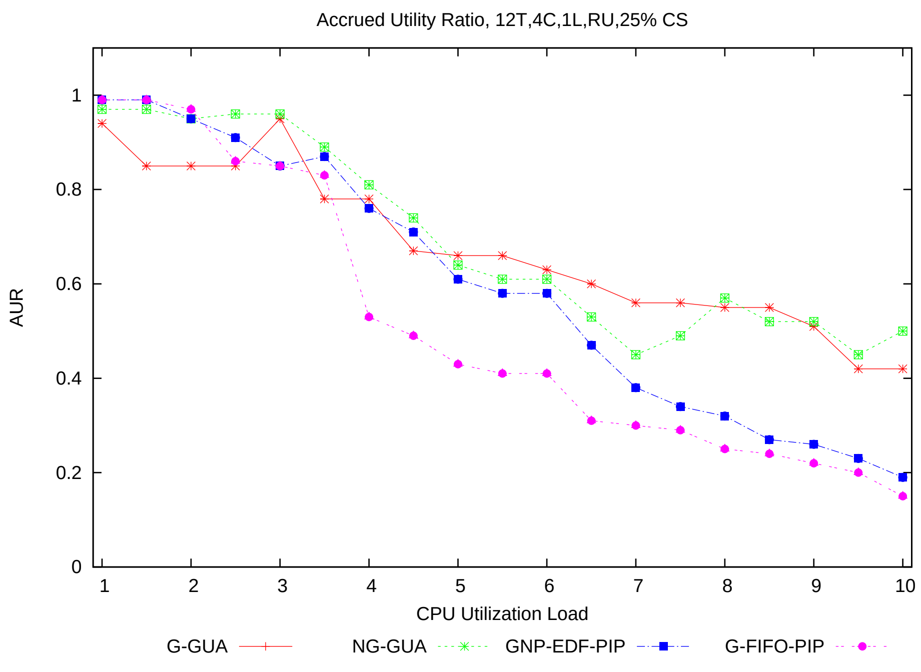


Figure 6.56: AUR vs. CPU utilization (12T, 4C, 1L, RU, 25% CS)

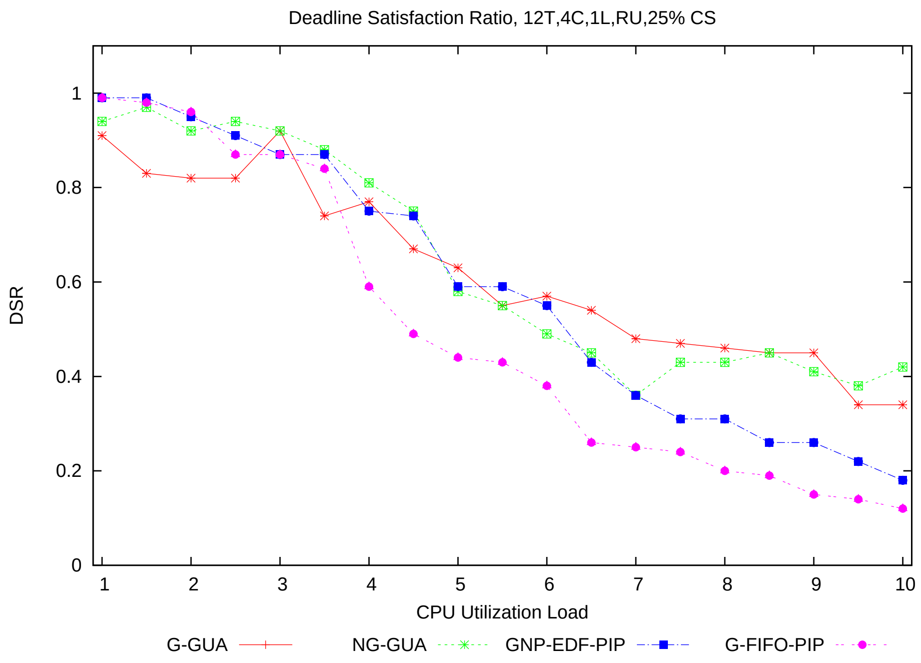


Figure 6.57: DSR vs. CPU utilization (12T, 4C, 1L, RU, 25% CS)

In Figures 6.58 and 6.59, which shows the results with 4 locks per task, with the critical section length as 5% of the WCET. We observe that by increasing the number of locks for the same critical section length, the performance of NG-GUA and G-GUA remains consistent. In Figure 6.58, NG-GUA is able to meet all deadlines till 350% utilization load. After 350% utilization load, G-NP-EDF's DSR falls down drastically and as a result its overall AUR is reduced. NG-GUA, on the other hand, provides an average of 75% AUR from 300% utilization load, all the way up to 1000% utilization load.

At 25% critical section length, the AUR results, as shown in Figures 6.60, indicates that, although the overall accrued utility of the system is low, G-GUA and NG-GUA still provide the highest utility accrual, especially during light overloads, ranging from 350% utilization load to 500% utilization load.

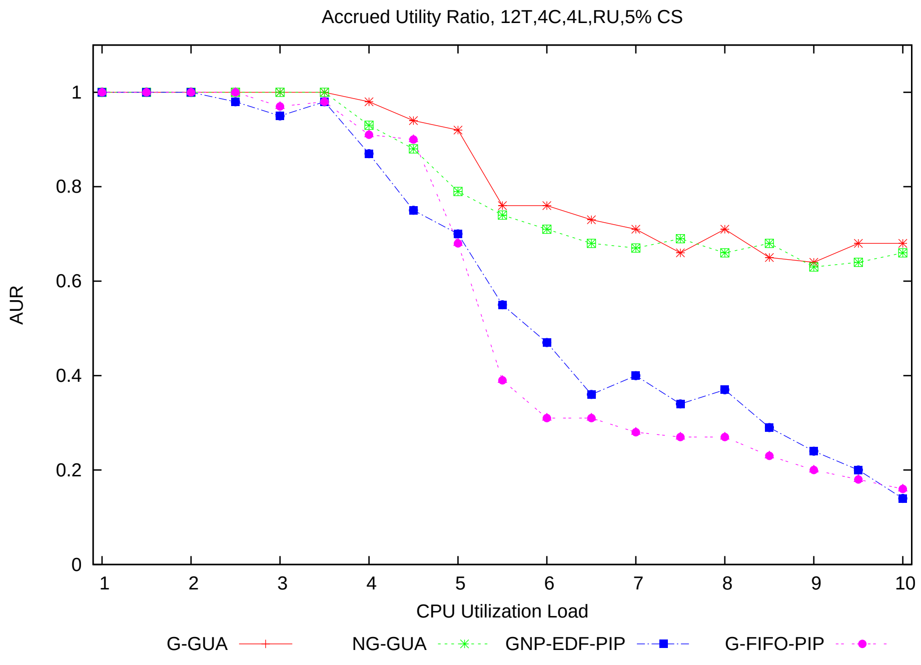


Figure 6.58: AUR vs. CPU utilization (12T, 4C, 4L, RU, 5% CS)

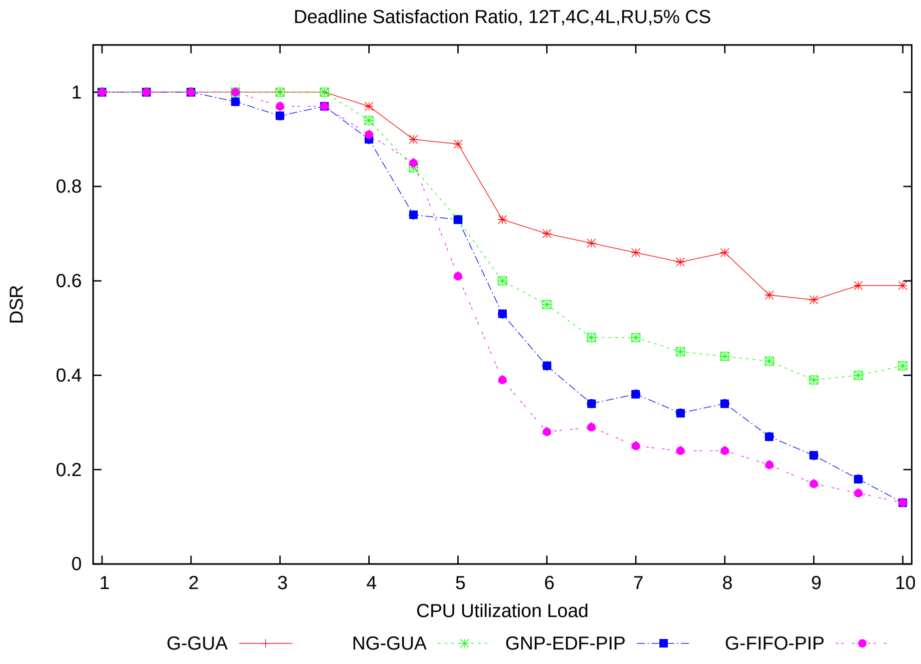


Figure 6.59: DSR vs. CPU utilization (12T, 4C, 4L, RU, 5% CS)

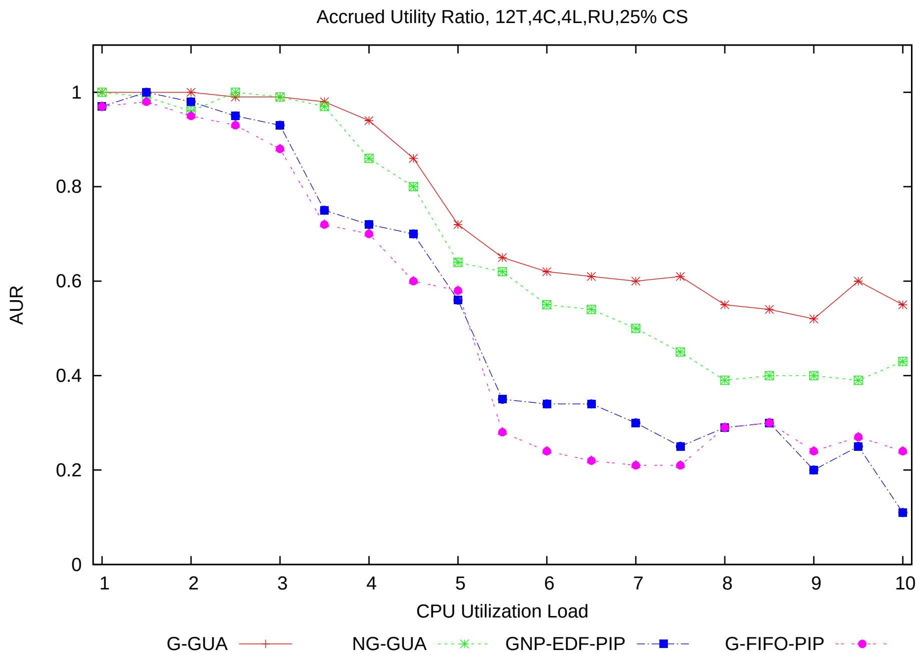


Figure 6.60: AUR vs. CPU utilization (12T, 4C, 4L, RU, 25% CS)

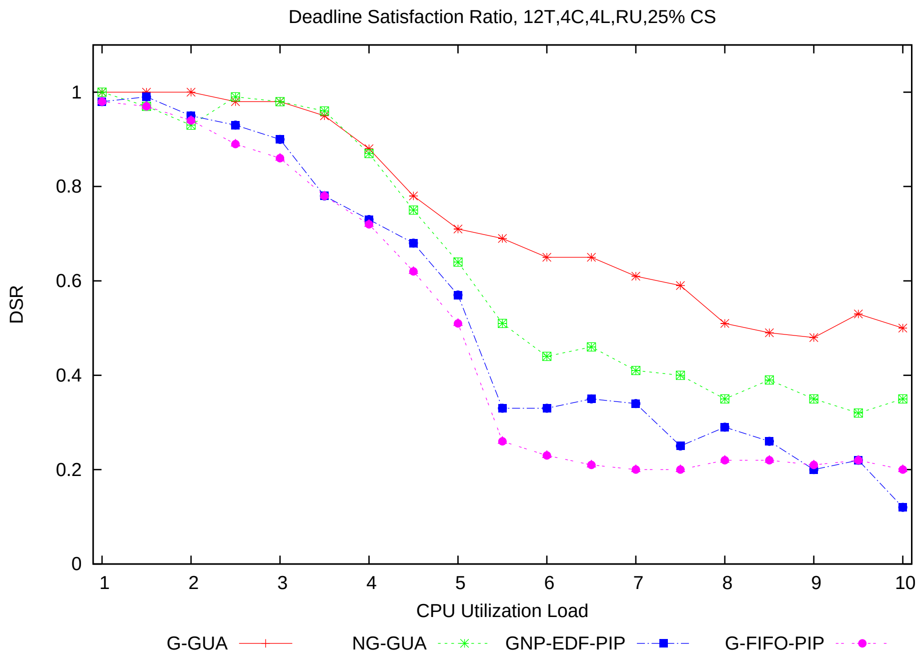


Figure 6.61: DSR vs. CPU utilization (12T, 4C, 4L, RU, 25% CS)

### 6.3.2 Varying Lock Critical Section Length

In this section, we consider the effects of varying the critical section length of locks while keeping the number of locks and the utilization load fixed. We consider the utilization loads of 400% and 800% that show a light-overload and a heavy-overload scenarios.

We consider the following combinations:

1. We use the 12T task-set with 1 lock per task and fix the utilization load at 400%; while varying the critical section length from 5% to 25%
2. We use the 12T task-set with 1 lock per task and fix the utilization load at 800%; while varying the critical section length from 5% to 25%
3. We use the 12T task-set with 4 locks per task and fix the utilization load at 400%; while varying the critical section length from 5% to 25%
4. We use the 12T task-set with 4 locks per task and fix the utilization load at 800%; while varying the critical section length from 5% to 25%

Figure 6.62 shows the AUR values across various scheduling algorithms for critical section lengths equal to 5%, 10% and 25% of the total worst case execution time. We observe that G-GUA provides a consistent average accrued utility of around 80% across all critical section lengths when a single lock is used. This is at 400% utilization load which, on a four-core platform, is light-overload as the system reaches its full utilization potential. Figure 6.63 provides the same comparison at 800% utilization load, which is a heavy-overload, considering a four-core system. We observe that G-GUA still provides an average of 60% accrued utility, even when the system has a heavy-overload. NG-GUA performs equally as G-GUA. On the other hand, G-GUA performs better than NG-GUA by around 5%. This behavior is consistent with the increase in critical section lengths. At 25% critical section length, G-GUA has an AUR of 50%, which is an improvement of 150% over G-NP-EDF with PIP for the same point.

Figure 6.64 shows the AUR results when 4 locks are used at 400% utilization load. We observe that G-GUA outperforms every other algorithm and provides a consistent average AUR of 80% across critical section lengths. NG-GUA is seen to provide a similar behavior, with G-GUA performing better by an average of 10%. G-GUA shows around 90% improvement in AUR for a critical section length of 5%. Figure 6.65 shows the same results at 800% utilization load. We observe that with 4 locks, the performance of deadline-based algorithm has degraded. G-GUA and N-GUA consistently perform better. Both G-GUA and NG-GUA provide an average AUR of around 60% across all critical section lengths.

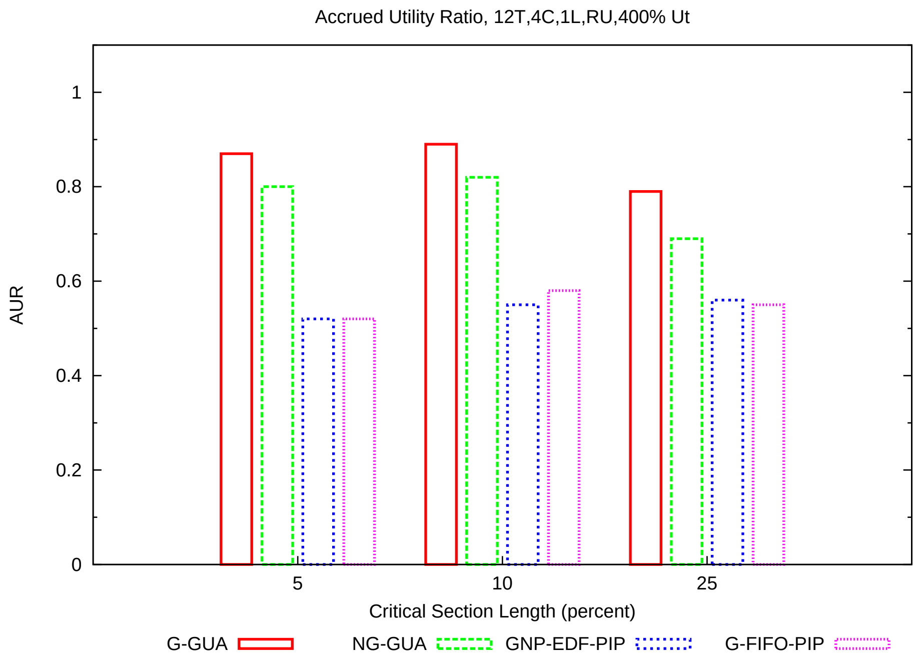


Figure 6.62: AUR vs. Critical section length (12T, 4C, 1L, RU, 400% CPU utilization)

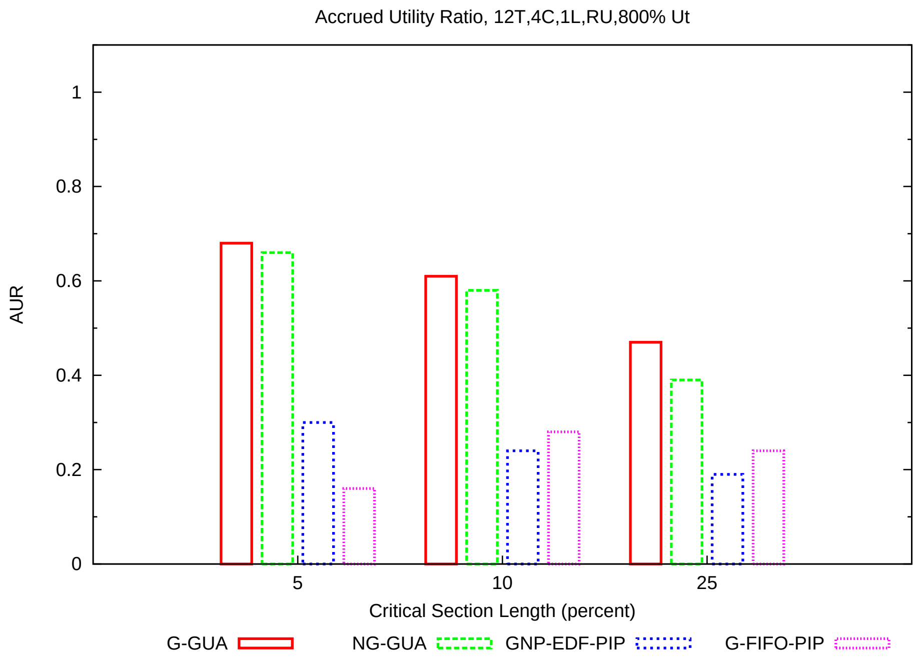


Figure 6.63: AUR vs. Critical section length (12T, 4C, 1L, RU, 800% CPU utilization)

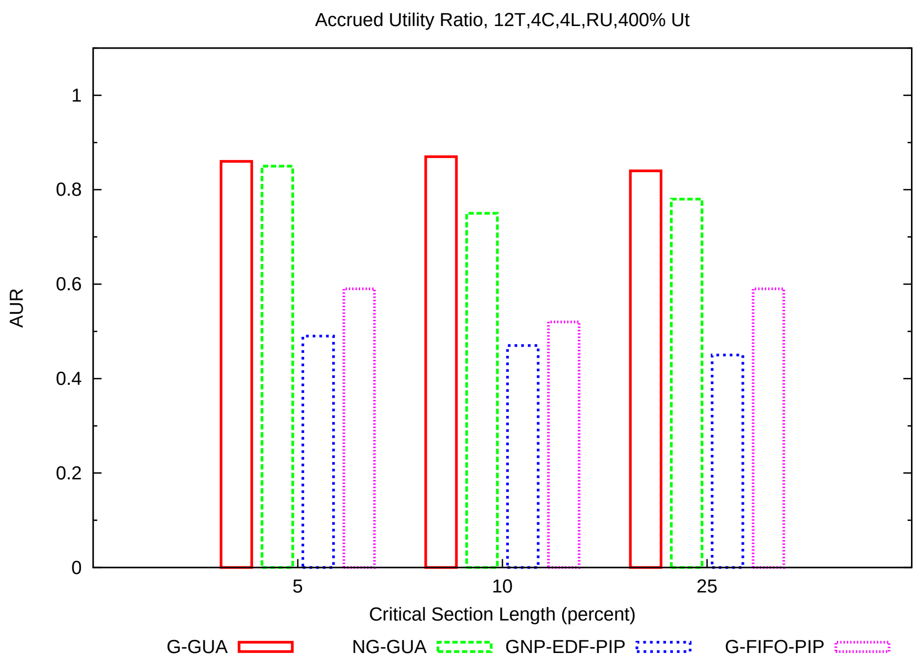


Figure 6.64: AUR vs. Critical section length (12T, 4C, 4L, RU, 400% CPU utilization)

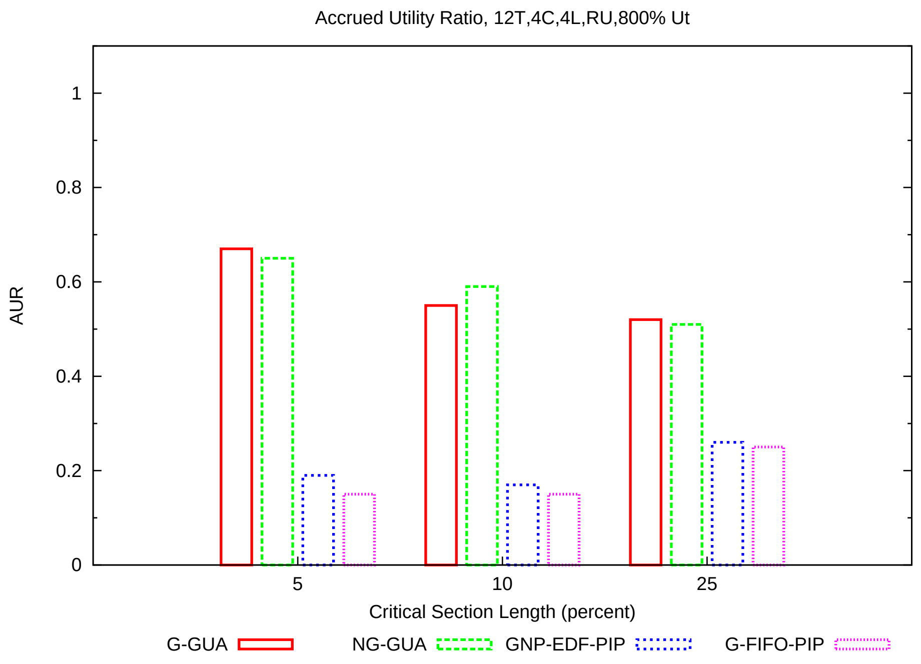


Figure 6.65: AUR vs. Critical section length (12T, 4C, 4L, RU, 800% CPU utilization)

### 6.3.3 Varying Number of Locks per Task

In this section, we consider the effects of varying the number of locks per task, while keeping the critical section length and the utilization load fixed. We consider the utilization loads of 400% and 800% that show light-overload and heavy-overload scenarios.

We consider the following combinations:

1. We fix the critical section length at 5% and the utilization load at 400%; while varying the total number of locks per task
2. We fix the critical section length at 5% and the utilization load at 800%; while varying the total number of locks per task
3. We fix the critical section length at 25% and the utilization load at 400%; while varying the total number of locks per task
4. We fix the critical section length at 25% and the utilization load at 800%; while varying the total number of locks per task

Figure 6.66 shows the total accrued utility as a function of the number of locks, with a fixed critical section length of 5% and a utilization load of 400%. This is a light-overload scenario. We observe that increasing the number of locks does not have an impact on the overall accrued utility for all the algorithms. G-GUA consistently outperforms other scheduling algorithms by providing an average AUR of around 85% across all locks. NG-GUA performs equally better with G-GUA providing an average 6% better performance. The deadline-based schedulers provide around 50% AUR across all locks. Figure 6.67 considers the same scenario with 25% critical section length and we observe a similar behavior.

However, during heavy-overloads, the performance of the UA scheduling algorithms is better than the deadline-based algorithms. This is shown in Figure 6.68 for an 800% utilization load with 5% critical section length and in Figure 6.69 for the same load with 25% critical section length. We consider the worst-case scenario with 25% critical section length and a heavy-overload at 800% utilization load. G-GUA provides an average 50% utility accrual across varying number of locks, while NG-GUA provides an average 45% utility accrual. G-GUA provides an average improvement of 120% in AUR over GNP-EDF.

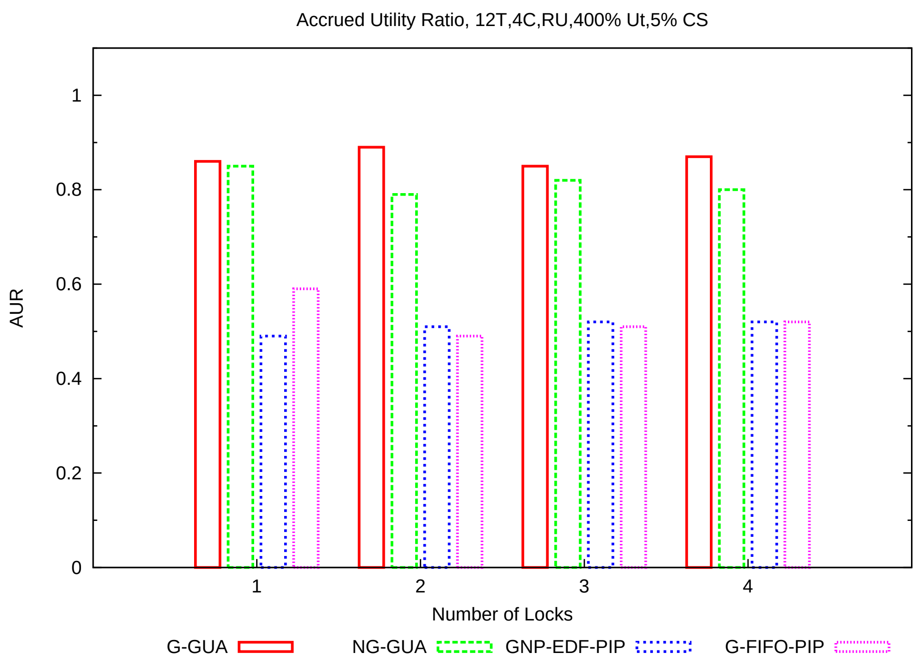


Figure 6.66: AUR vs. Number of locks (12T, 4C, RU, 5% CS, 400% CPU utilization)

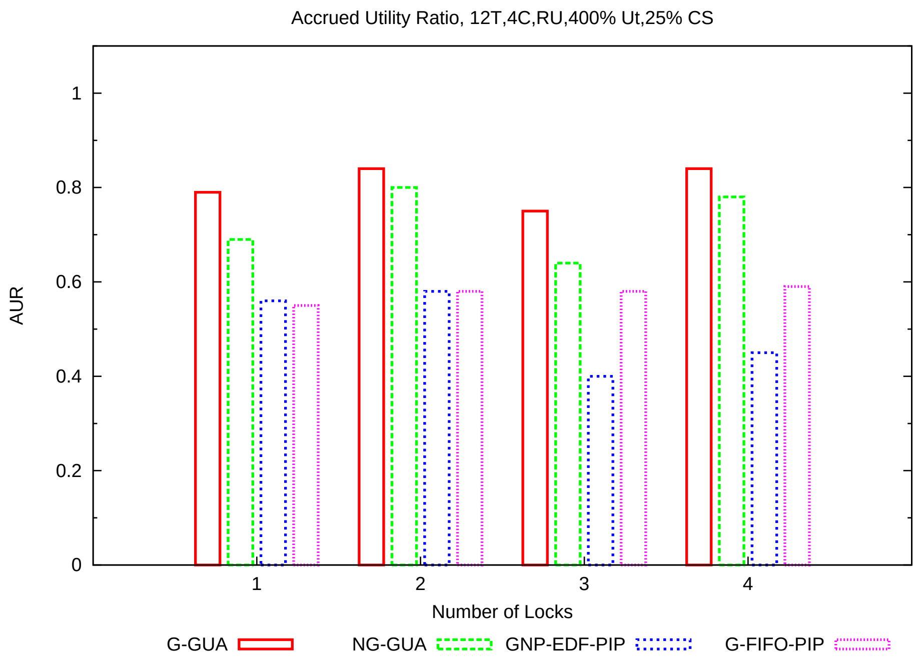


Figure 6.67: AUR vs. Number of locks (12T, 4C, RU, 25% CS, 400% CPU utilization)

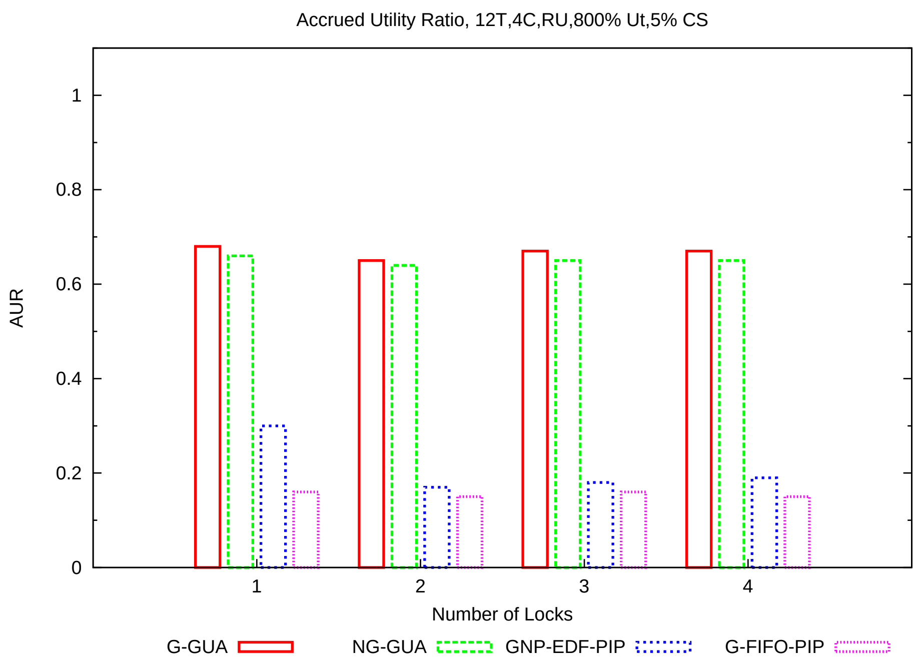


Figure 6.68: AUR vs. Number of locks (12T, 4C, RU, 5% CS, 800% CPU utilization)

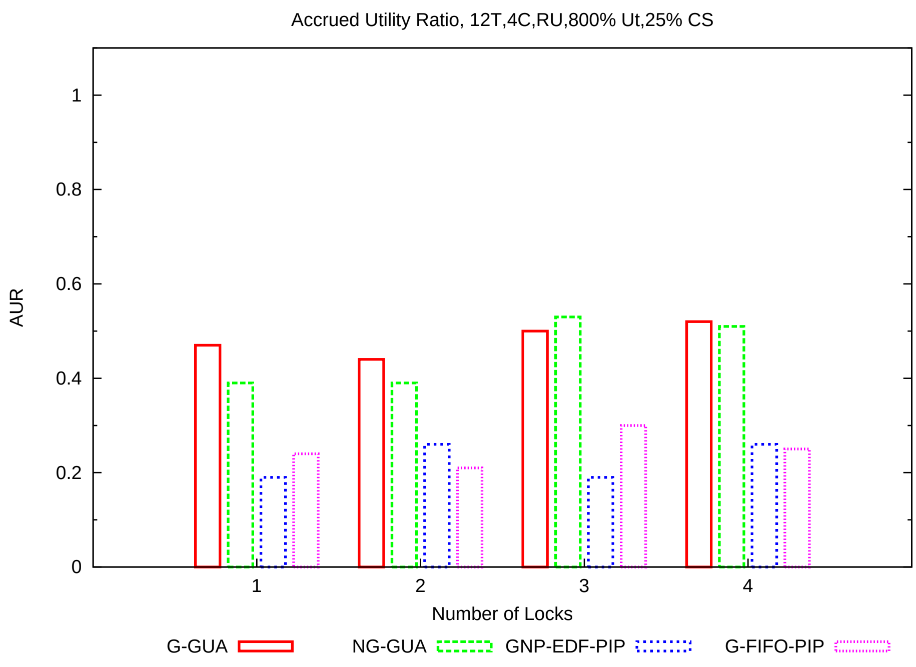


Figure 6.69: AUR vs. Number of locks (12T, 4C, RU, 25% CS, 800% CPU utilization)

## 6.4 Overhead Measurements

In this section, we measure and compare the scheduling overheads for G-GUA and NG-GUA on ChronOS. We consider a four-core platform and use three task-sets. The details of the task-sets follow:

1. The first task-set uses 5 tasks, with deadline/periods in the range of $[50000\mu s - 5000000\mu s]$; and a utilization load of each task in the range of $[0.1 - 0.4]$.
2. The second task-set uses 12 tasks, with deadlines/periods in the range of $[300000\mu s - 20000000\mu s]$; and a utilization load of each task in the range of $[0.01 - 0.4]$.
3. The third task-set uses 27 tasks, with deadlines/periods in the range of $[50000\mu s - 7500000\mu s]$; and a utilization load of each task in the range of $[0.01 - 0,3]$.

These task-set are the same that were used in the earlier experimental evaluation. Instead of representing the deadline range in millisecond ($ms$), we present the values in microseconds ($\mu s$). This has been done so that the overhead numbers can be matched with the total task execution costs for a fairer comparison.

In order to measure the overheads in ChronOS, we instrument the Linux kernel and capture various triggers such as, task migration and global scheduling; and use the `read_tsc()` call to get the time with a nanosecond granularity. We measure the overheads for each task-set over an increasing utilization load. We find that that average task migration overhead in ChronOS is $8\mu s$, with a standard deviation of $3\mu s$.

Figure 6.70 shows the scheduling overheads for G-GUA under a variable utilization load for task-sets with 5, 12 and 27 tasks, respectively. The time is presented in microseconds within a $[0 - 40]\mu sec$ range. We observe that with increasing number of tasks, the total scheduling overhead of G-GUA increases. At a 400% utilization load, we observe that when the number of tasks are increased from 5 to 27, the average scheduling overhead increases from $10\mu s$ to $17\mu s$. Similarly at 800%, we observe that the average scheduling overhead increases from $11\mu s$ to $30\mu s$.

On the other hand, we observe that the non-greedy-GUA has a lesser overhead than the greedy variant. Figure 6.71 shows the results for NG-GUA. The average scheduling overhead increases as the number of tasks in the system are increased but the overhead is lower than that of G-GUA. At 100% utilization load, NG-GUA has around $5\mu s$ scheduler overhead as compared to $10\mu s$ for G-GUA. However, at 800% utilization load, we observe that the average scheduler overhead increases to around $15\mu s$ for the 27 task-set as compared to $30\mu s$ for G-GUA. The high overhead of G-GUA is expected as it is more aggressive while trying to maximize the accrued utility when compared to NG-GUA.

Figure 6.72 compares both NG-GUA and G-GUA overheads at 100% and 800% utilization loads, with an increasing number of tasks. As the number of tasks in the system increase,

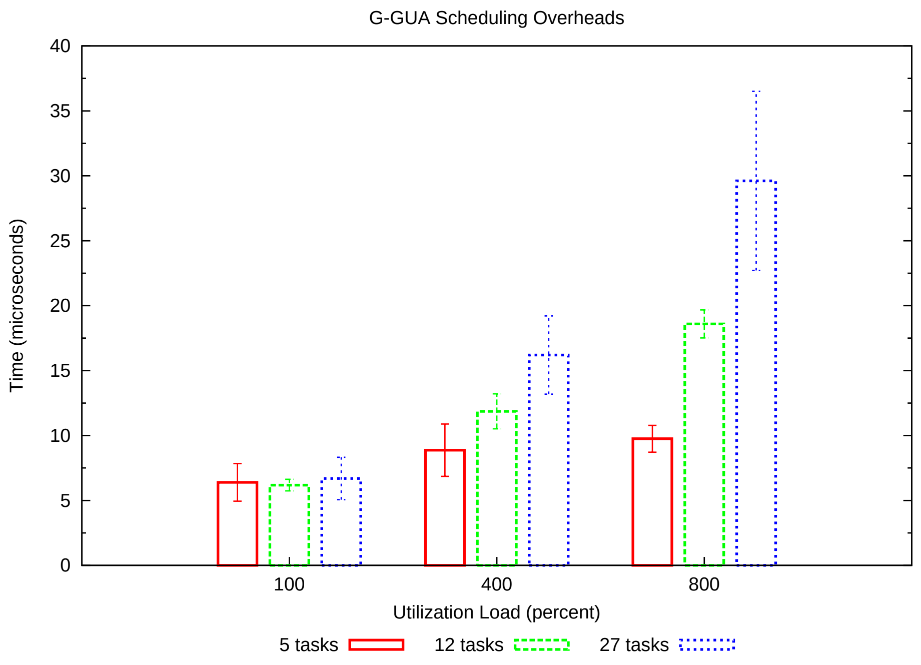


Figure 6.70: Scheduling overheads for G-GUA under variable utilization load

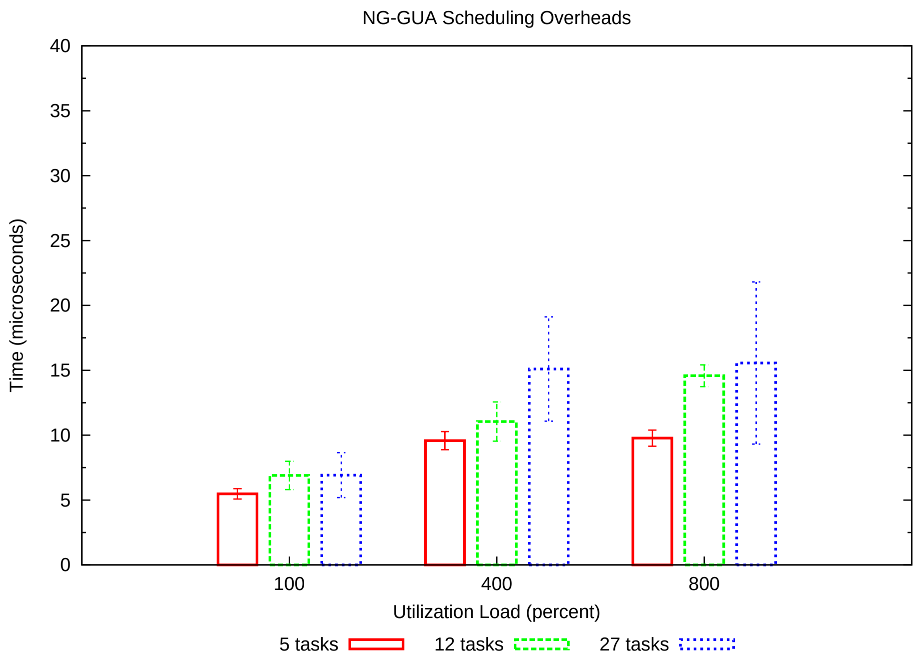


Figure 6.71: Scheduling overheads for NG-GUA under variable utilization load

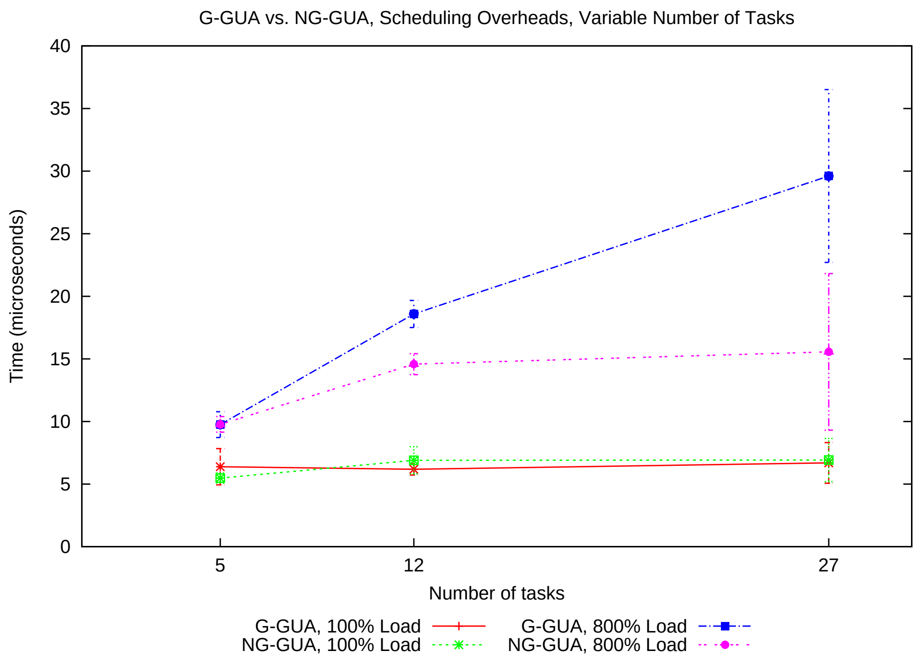


Figure 6.72: G-GUA vs. NG-GUA, scheduling overheads with variable number of tasks

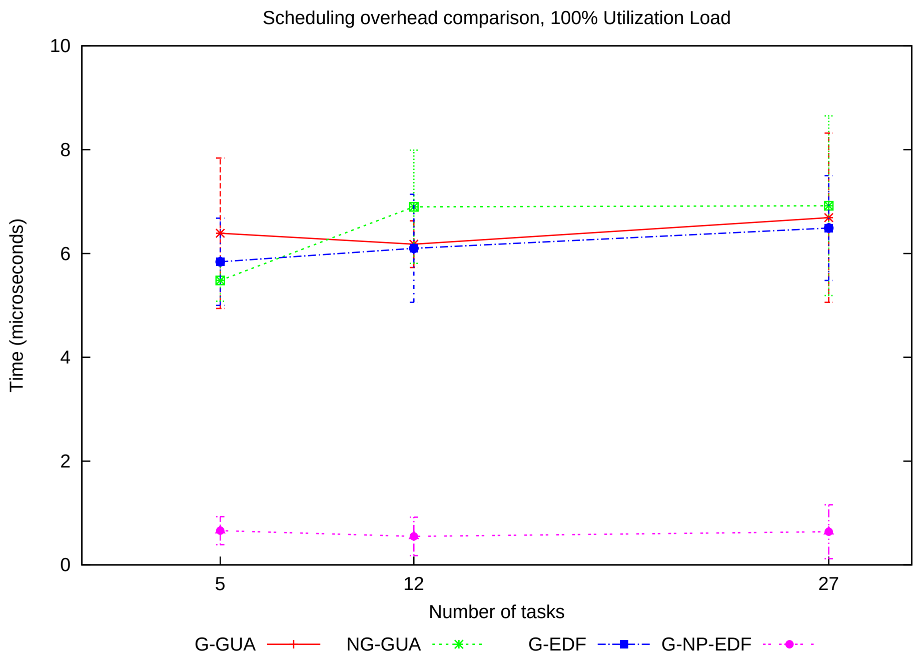


Figure 6.73: Comparison of scheduling overheads at 100% utilization load

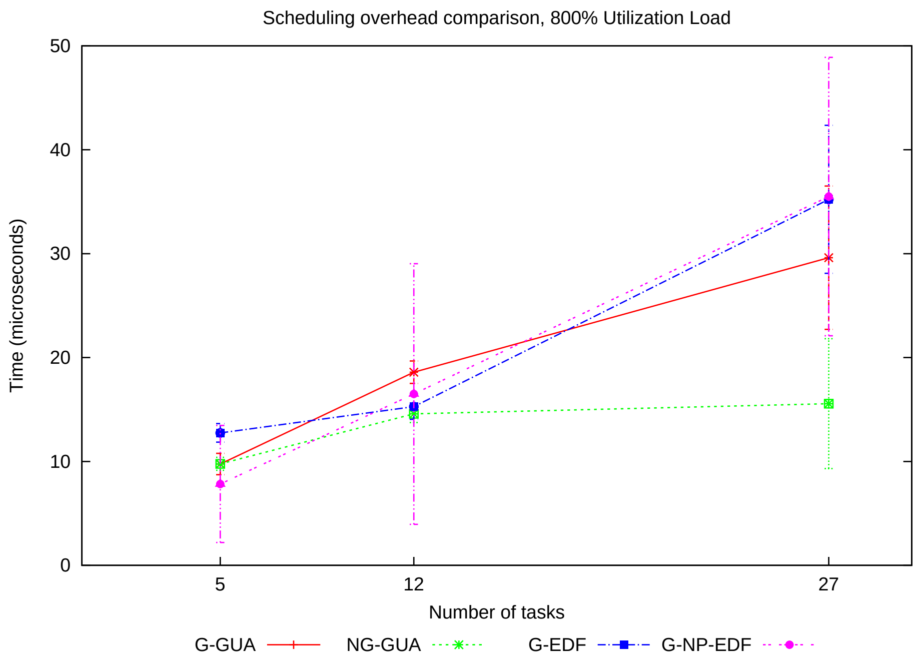


Figure 6.74: Comparison of scheduling overheads at 800% utilization load

the average scheduling overhead for NG-GUA and G-GUA increases. As mentioned earlier, NG-GUA observes lesser overhead as compared to G-GUA. In Figures 6.73 and 6.74, we compare the scheduling overheads of NG-GUA and G-GUA with G-EDF and G-NP-EDF. It can be seen that during 100% utilization loads, G-EDF has similar overheads like our algorithms. G-NP-EDF, on the other hand, has a lower scheduler overhead when compared with the other algorithms. However, we observe that during 800% utilization load, G-NP-EDF's overhead shoots up from around $1\mu s$ during 100% utilization load to around $35\mu s$ under 800% utilization load.

NG-GUA and G-GUA perform sorting of the *zero in-degree* tasks based on the PIP deadlines and GVD, respectively. In our implementation of these algorithms in ChronOS, we have used a quicksort-based sorting algorithms, which has a worst-case performance cost of $O(n^2)$. As we measured the overheads using task-sets that do not use any locks, all the tasks in the ready queue are *zero in-degree* and hence eligible for schedule. We observe that the total cost of sorting can be reduced if we use more optimized data structures (e.g., binomial heap has a worst case cost of $O(\log n)$ and $O(1)$ for lookup of the minimum key).

## 6.5 Conclusions

In this chapter, we performed a number of experiments to compare the performance of the GUA class of algorithms with other global and partitioned scheduling algorithms in the absence and presence of dependencies.

The results indicate that G-GUA and NG-GUA perform better than other algorithms during overloads in accruing total utility. We also observe that G-GUA and NG-GUA meet more deadlines during overloads than the various deadline-based scheduling algorithms, such as G-EDF, G-NP-EDF and P-EDF.

We also measure the average scheduling overheads of the algorithms and find that for 27 tasks with periods in the range of $[50000\mu s - 7500000\mu s]$, NG-GUA has an average scheduling overhead of $15\mu s$, while that of G-GUA is around $30\mu s$.

# Chapter 7

# Conclusions and Future Work

In this thesis, we addressed the problem of real-time scheduling on multiprocessors, focusing on applications that are subject to run-time uncertainties causing overloads, and task dependencies — a previously open problem. The thesis presents the GUA class of algorithms for this problem. Since the problem is NP-hard, the algorithms are polynomial-time heuristics. The algorithms construct a directed acyclic graph representation of the task dependency relationship, and build a global multiprocessor schedule of the zero in-degree tasks to heuristically maximize the total accrued utility and ensure mutual exclusion. Deadlocks are detected through a cycle-detection algorithm, and resolved by aborting a task in the deadlock cycle. The GUA class of algorithms include the NG-GUA and G-GUA algorithms. The two algorithms differ in the way schedules are constructed towards meeting all task deadlines, when possible to do so. We establish several properties of the algorithms including conditions under which all task deadlines are met, satisfaction of mutual exclusion constraints, and deadlock-freedom.

We also create a Linux-based real-time kernel for multiprocessors called ChronOS, which is extended from the `PREEMPT_RT` real-time Linux patch. ChronOS provides optimized interrupt service latencies and real-time locking primitives (by virtue of the `PREEMPT_RT` patch), and provides a scheduling framework for the implementation of a broad range of real-time scheduling algorithms, including utility accrual, non-utility accrual, global, and partitioned scheduling algorithms.

We implement the GUA class of algorithms and their competitors in ChronOS and conduct experimental studies. Our study reveals that —

(1) In the absence of dependencies, the GUA class of algorithms accrue higher utility and satisfy greater number of deadlines than the deadline-based algorithms (G-EDF, G-NP-EDF) by as much as 750% and 600%, respectively.

(2) In the absence of dependencies, the GUA class of algorithms accrue higher utility and satisfy greater number of deadlines compared to the partitioned algorithms by as much as

90% and 75%, respectively for P-DASA; and 450% and 600%, respectively for P-EDF.

(3) As gMUA defaults to NG-GUA without dependencies, the performance of NG-GUA and gMUA is similar.

(4) G-GUA outperforms NG-GUA and gMUA by accruing 25% more utility, while both NG-GUA and gMUA satisfy 5% more deadlines during underloads than G-GUA.

(5) In the presence of dependencies, both NG-GUA and G-GUA accrue higher utility and satisfy greater number of deadlines than G-NP-EDF-PIP by as much as 250% and 150%, respectively.

Our research demonstrates that it is possible to design scheduling algorithms for the dynamic, multiprocessor real-time scheduling problem space (i.e., those characterized by execution overruns, unpredictable task arrivals, etc.), such that they yield an optimal timeliness behavior (e.g., meeting all deadlines; obtaining maximum total utility), when total utilization demand does not exceed the algorithms' utilization bound, and a best-effort timeliness behavior at all other times.

This approach was pioneered in the Alpha OS kernel [44], which included two generations of TUF/UA scheduling algorithms for scheduling single-processor systems [55, 31]. At its core, this thesis demonstrates that a similar approach can also be successfully extended for multiprocessors. The key challenge in doing so is the construction of $m$-processor schedules with best-effort timeliness behavior during overloads ($U > m$) with or without dependencies, that seamlessly yield optimal timeliness behavior during underloads ($U \leq m$) without dependencies. This is a difficult problem because, deadline scheduling is non-optimal for multiprocessors, unlike that for single-processors (for which deadline scheduling is optimal). In particular, global EDF has a utilization bound of $\approx m/2$ (without dependencies). Thus, using deadline schedulers as the basis for multiprocessor TUF/UA scheduling, as done in Alpha's TUF/UA schedulers [55, 31] for the single-processor case, will result in loss of utilization—e.g., with global EDF as the basis, as is the case with NG-GUA, "overloads" start at $\approx m/2$. Optimal multiprocessor real-time schedulers are either quantum-based (e.g., the Pfair class of algorithms) [11] or are strongly dependent upon presumptions made about task arrival and task execution-time behaviors (e.g., periodic/sporadic arrivals, WCETs) [25, 21, 37]. Thus, it is very difficult to seamlessly extend them to have best-effort timeliness behaviors for the dynamic problem space, where tasks may have execution overruns and unpredictable arrivals. The thesis therefore designs NG-GUA with global EDF as its basis (suffering from this utilization loss), and G-GUA as an alternative solution without global EDF as its basis and one that greedily constructs schedules at all times. Designing multiprocessor TUF/UA scheduling algorithms with utilization bounds that are greater than $\approx m/2$ is a direction for future work.

In addition, the presence of task dependencies can result in many task dependency chains, similar to the single-processor case. But unlike the single-processor case, since there are $m > 1$ processors, for the multiprocessor case, up to $m$ of these chains (or tasks at the head of those chains) can potentially be executed. In [31], the dependency chains are computed at the per task level. This works well for the single-processor case as only one task needs to be selected at the end of schedule. However, we can not take this method and apply it to the multiprocessor case because to ensure mutual exclusion we can not execute tasks on processors that depend on each other. Hence, we need to find the dependency relationship of all the tasks. The challenge is to find out the most effective way in which this can be done. In the design of the GUA algorithms, we solve this problem by creating a directed acyclic graph to represent the dependency relationship between tasks. Thus, at the end of the graph creation, we can consider the zero in-degree nodes, which are not dependent on other tasks, as eligible for the final schedule.

However, which $m$ head tasks (or zero in-degree tasks) should have the highest execution eligibility? The potential utility density (or PUD) metric pioneered in [31] is shown to be highly effective in determining task execution eligibility for the single-processor case. But how can we determine task potential utility densities when similar tasks can appear in the dependency chains of several tasks? Unlike the single processor case, here, many tasks can be concurrently dispatched for execution. The PUD metric works in [31] for the single-processor case as a single task needs to be selected at the end of the schedule. However, on the multiprocessor case we can not use the PUD metric alone in order to pick one task over another. This is because there could be a zero in-degree task $T_i$ that owns a resource $R_j$ and blocks other tasks in the system, but has a low PUD value. As a result, (due to the low PUD value), $T_i$ would always be pushed to the back of the queue, thus preventing other tasks that are currently blocked on it from executing. The challenge here is to find a metric that provides a way to represent the overall benefit the system can accrue if a particular task, say $T_i$, is selected for execution. We answer this question in the design of the GUA class of algorithms by defining a metric called the Global Value Density (or GVD). The GVD for a zero in-degree task represents the aggregate value density for the entire dependency chain, which gives a fair representation of the dependency chain and thus provides the highest execution eligibility for a task that is currently blocking other tasks.

The GUA class of algorithms have higher scheduling costs than past multiprocessor real-time scheduling algorithms. The asymptotic cost of G-GUA is $O(mn \log n)$ and NG-GUA is $O(mn \log n)$ for $n$ tasks on $m$ processors. In contrast, the asymptotic cost of gMUA is $O(mn \log n)$, G-EDF is $O(n \log n)$ and P-EDF is $O(n \log n)$. On our four-core platform, the worst-case and average-case scheduling overheads of G-GUA for 27 tasks at 800% utilization load were $80\mu s$ and $30\mu s$, respectively, while those for NG-GUA were $45\mu s$ and $15\mu s$, respectively. We observe an average $8\mu s$ migration cost. Thus, the GUA class of algorithms are effective only if the application can tolerate their higher scheduling overheads. Designing multiprocessor TUF/UA scheduling algorithms with smaller scheduling costs is another important direction for future work.

## 7.1 Contributions

To summarize, the research contributions of the thesis include:

1. the GUA class of multiprocessor real-time scheduling algorithms that allow tasks to be subject to run-time uncertainties, overloads, and dependencies, and yield optimal total utility when possible and best-effort timeliness behavior otherwise — the first such multiprocessor real-time scheduling algorithms to do so.

2. the ChronOS multiprocessor real-time Linux kernel that provides optimized interrupt service latencies and real-time locking primitives (by virtue of `PREEMPT_RT` patch) and a scheduling framework that allows the implementation of a broad range of real-time scheduling algorithms, including utility accrual, non-utility accrual, global, and partitioned scheduling algorithms – the first such multiprocessor real-time Linux kernel.

## 7.2 Future Work

Although, our results in this thesis show that NG-GUA and G-GUA provide improvements in the overall accrued utility, there are some open research problems that can be used to optimize this performance further. Below, we enumerate some of these.

### 7.2.1 Parallelizing Schedule Creation

The current approach used by the GUA class of algorithms is to create a global schedule for all available processors in the system at every scheduling event. This has been implemented in ChronOS as the "Stop-the-World" model. In this architecture model, the processor creating the global schedule sends an `IPI` to all the other processors to stop what they are doing and enter the scheduler.

For global scheduling algorithms, such as NG-GUA and G-GUA, it is necessary to have all the processors wait until the creation of the new schedule to avoid schedule complications. This is primarily because the GUA class of algorithms use the heuristic of Global Value Density which is calculated based on a task's remaining execution cost. However, while all the processors are blocked before the final schedule is available, we can optimize the GUA class of algorithms such that the creation of the final schedule uses all the available processors. One approach of doing this is for the scheduling algorithm to create jobs that can be assigned to worker threads dedicated for scheduling on each of the processors.

At this point, it is not known if this approach would provide any additional optimization benefits to the overall performance of the GUA class of algorithms; or if the parallel approach would scale with the increase in the number of processors. We propose the future work to consider the design of a parallel versions of the GUA class of algorithms.

### 7.2.2 Reducing Scheduling Overheads

The current asymptotic cost of the GUA class of algorithms is $O(mn \log n)$. This is primarily because the algorithms require the list of *zero in-degree* tasks to be sorted either by deadline or by the GVD. In the current design and implementation of the algorithms, we have used quicksort-based sort which suffers from a worst-case performance penalty of $O(n^2)$, thus contributing to the main overhead cost of the algorithms. We propose the future work to design TUF/UA scheduling algorithms with a lesser scheduling overheads.

### 7.2.3 Cache-Aware Algorithms

One of the most important aspects of global scheduling algorithms is to be able to assign tasks to processors such that cache warm-up issues and task migrations from one processor run-queue to another are avoided. This happens when the final schedule created by the algorithms have tasks that belong to the same processor.

We implement a default mapping algorithm in ChronOS that assigns tasks from the final schedule to processors based on their origin. However, if the final schedule has tasks that belong to the same processor, the mapping algorithm suffers a worst-case task migration cost of $m - 1$, for a $m$ processor system.

We propose the future work to look at two ways in which this problem can be solved – (i) provide a better mapping algorithm in ChronOS; or (ii) ensure that the scheduling algorithm is cache-aware [38, 17]. Stenström provides a survey of various cache coherence schemes on multiprocessors in [64].

### 7.2.4 Approximate Algorithms

As mentioned earlier, the problem of scheduling real-time dependent tasks on multiprocessors is NP-Hard. In this thesis, we present polynomial-time real-time scheduling heuristics algorithms (NG-GUA and G-GUA) that yield optimal total utility when possible and best-effort timeliness behavior otherwise.

Another approach of solving this problem is to consider the design of approximate algorithms instead of a heuristic approach by defining it as an optimization problem. We believe that the results presented in this thesis can provide a useful insight to the overall performance and can be used as an input while designing an approximate algorithm. We propose this as a future work.

# Appendix A

# ChronOS System Calls

In code listing A.1, we show the ChronOS system calls related to the real-time scheduling that were added to 2.6.31.12 Linux kernel. The system calls related to Distributed Threads have been omitted due to the lack of context with the research contributions of this thesis.

```
/* Set the real-time scheduler */
long sys_set_scheduler(int rt_sched,
                       int prio,
                       unsigned int len,
                       unsigned long __user *user_mask_ptr);

/* Begin a real-time segment */
long sys_begin_rt_seg(struct rt_data __user * data,
                      struct timespec __user *deadline,
                      struct timespec __user *period);

/* End a real-time segment */
long sys_end_rt_seg(int tid, int newprio);

/* Request/Release locks */
long sys_do_vt_rt_mutex(struct mutex_data __user *mutexreq, int op)

/* Add an abort handler to a real-time task */
long sys_add_abort_handler(int tid,
                           int max_util,
                           struct timespec __user *deadline,
                           unsigned long exec_time)

/* Perform atomic operations */
long sys_atomic_int_op(int __user *resource, int op, int val)
```

Listing A.1: The ChronOS system calls added to the Linux 2.6.31.12 kernel

# Appendix B

# ChronOS Kernel Data Structures

In this section. we show the data structure that were added to the 2.6.31.12 Linux kernel for real-time scheduling. The data structures related to Distributed Threads have been omitted due to the lack of context with the research contributions of this thesis.

```
struct mutex_data {
    atomic_t val;
    int owner;
};
```

Listing B.1: Data structure used to define locks

```
struct abort_info {
    struct timespec deadline;
    unsigned long exec_time;
    int max_util;
    struct sigqueue *abort_sig;
};
```

Listing B.2: Data structure used for abort handlers

```
struct rt_graph {
    struct timespec agg_left;
    unsigned long   agg_util;
    long global_ivd;
    long in_degree;
    long out_degree;
    struct rt_info *neighbor_list;
    struct rt_info *next_neighbor;
    struct rt_info *parent;
    struct rt_info *depchain;
};
```

Listing B.3: Data structure for the DAG representation of the dependency chain

```c
/* Structure attached to struct task_struct */
struct rt_info {
    /* Real-Time information
     *
     * start_time is the start time of the rt segment, not to be confused
     * with struct task_struct.start_time, which is the creation time
     * of the thread.
     */
    struct timespec start_time;     /* monotonic time */
    struct timespec deadline;       /* monotonic time */
    struct timespec temp_deadline;      /* monotonic time */
    struct timespec period;         /* relative time */
    struct timespec left;           /* relative time */
    unsigned long exec_time;        /* WCET, us */
    int max_util;
    long inv_val_den;

    /* Lists */
    struct list_head g_task_list;
    struct list_head local_task_list;
    struct list_entry list[SCHED_LISTS];

    /* DAG used by x-GUA class of algorthims */
    struct rt_graph graph;

    /* Lock information */
    struct mutex_data *requested_resource;
    struct rt_info *dep;
    int lock_count;

    /* Abort information */
    struct abort_info abortinfo;

    /* Task state information */
    unsigned char flags;
    int cpu;
};
```

Listing B.4: The ChronOS main real-time data structure

```c
struct rt_sched_local {
    struct list_head list;
    /* Scheduler Name */
    char *name;
    /* Scheduling number and flags */
    int number;
    int flags;
    /* Scheduling function */
    struct rt_info* (*schedule) (struct list_head *head, int flags);
};
```

Listing B.5: Scheduler plugin for single-processor schedulers

```c
struct rt_sched_global {
    struct list_head list;
    /* Scheduler Name */
    char *name;
    /* Scheduling number and flags */
    int number;
    int flags;
    /* Scheduling functions */
    struct rt_info* (*schedule) (struct list_head *head, int flags, int cpus);
    struct rt_info* (*preschedule) (struct list_head *head, int flags);
    int (*arch_init) (void);
    void (*arch_release) (void);
    void (*block) (void);
    int (*map_tasks) (struct rt_info *head);
    /* The local scheduler to be used with this global */
    int local;
};
```

Listing B.6: Scheduler plugin for multiprocessor schedulers

# Appendix C

# Sample ChronOS Scheduling Kernel Modules

In code listing C.1, we show a sample kernel module used to implement the EDF scheduling algorithm in ChronOS. EDF is a uniprocessor scheduling algorithm. Hence, we use the scheduling plugin for local schedulers in ChronOS.

```
#include <linux/module.h>
#include <linux/list.h>

struct rt_info* sched_edf(struct list_head *head, int flags)
{
    /* Scheduling logic here */
    return task;
}

struct rt_sched_local rt_sched_edf = {
    .name = ``EDF'',
    .number = SCHED_RT_EDF,
    .flags = 0,
    .schedule = sched_edf
};

struct rt_sched_local *edf = &rt_sched_edf;

static int __init edf_init(void)
{
    return add_local_scheduler(edf);
}
module_init(edf_init);

static void __exit edf_exit(void)
{
    remove_local_scheduler(edf);
}
module_exit(edf_exit);

MODULE_DESCRIPTION(``description'');
MODULE_AUTHOR(``name'');
MODULE_LICENSE(``GPL'');
```

Listing C.1: Sample kernel module for EDF

In code listing C.2, we show a sample kernel module used to implement the NG-GUA scheduling algorithm in ChronOS. NG-GUA is a multiprocessor, global scheduling algorithm which uses the Stop-the-World architecture model. Hence, we use the scheduling plugin for global schedulers in ChronOS.

```
#include <linux/module.h>
#include <linux/list.h>

struct rt_info * presched_nggua(struct list_head *head, int flags)
{
    /* Pre-scheduling logic here */
    return NULL;
}

struct rt_info * sched_nggua(struct list_head *head, int flags, int cpus)
{
    /* Scheduling logic here */
    return NULL;
}

struct rt_sched_global rt_sched_nggua = {
    .name = ``NG_GUA'',
    .number = SCHED_RT_NGGUA,
    .flags = 0,
    .schedule = sched_nggua,
    .preschedule = presched_nggua,
    .arch = &rt_sched_arch_stw,
    .local = SCHED_RT_FIFO_RA
};

struct rt_sched_global *nggua = &rt_sched_nggua;

static int __init nggua_init(void)
{
    return add_global_scheduler(nggua);
}
module_init(nggua_init);

static void __exit nggua_exit(void)
```

```
{
    remove_global_scheduler(nggua);
}
module_exit(nggua_exit);

MODULE_DESCRIPTION(``description'');
MODULE_AUTHOR(``name'');
MODULE_LICENSE(``GPL'');
```

Listing C.2: Sample kernel module for NG-GUA

# Bibliography

[1] Chronos - linux based real-time multiprocessor scheduling framework. `http://www.chronoslinux.org`.

[2] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society.

[3] J. H. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68(1):157–204, 2004.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. Technical report, Chapel Hill, NC, USA, 2001.

[5] S. ANSI/ISO/IEC-8652:1995. Ada 95 reference manual. intermetrics, inc. January 1995.

[6] T. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760–768, aug. 2005.

[7] U. Balli, H. Wu, B. Ravindran, J. S. Anderson, and E. Douglas Jensen. Utility accrual real-time scheduling under variable cost functions. *IEEE Trans. Comput.*, 56(3):385–401, 2007.

[8] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 321–329, Washington, DC, USA, 2005. IEEE Computer Society.

[9] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. Comput.*, 55(7):918–923, 2006.

[10] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Syst.*, 32(1-2):9–20, 2006.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[12] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 280–288, Washington, DC, USA, 1995. IEEE Computer Society.

[13] S. K. Baruah and J. R. Haritsa. Scheduling for overload in real-time systems. *IEEE Trans. Comput.*, 46(9):1034–1039, 1997.

[14] A. Bechini and C. A. Prete. Performance-steered design of software architectures for embedded multicore systems. *Softw. Pract. Exper.*, 32(12):1155–1173, 2002.

[15] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *ECRTS '05*, pages 209–218, 2005.

[16] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *RTSS '08*, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society.

[17] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS '09: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, Washington, DC, USA, 2009. IEEE Computer Society.

[18] 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 on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

[19] K. Channakeshava. Utility accrual real-time channel establishment in multi-hop networks. Master's thesis, Virginia Tech, Aug 2008.

[20] K. Channakeshava, B. Ravindran, and E. D. Jensen. Utility accrual channel establishment in multihop networks. *IEEE Trans. Comput.*, 55(4):428–442, 2006.

[21] S.-Y. Chen and C.-W. Hsueh. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 147–156, Washington, DC, USA, 2008. IEEE Computer Society.

[22] H. Cho. *Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs.* PhD thesis, Virginia Tech, 2006.

[23] H. Cho, C. Na, B. Ravindran, and E. D. Jensen. On scheduling garbage collector in dynamic real-time systems with statistical timing assurances. *Real-Time Syst.*, 36(1-2):23–46, 2007.

[24] H. Cho, B. Ravindran, and E. D. Jensen. Lock-free synchronization for dynamic embedded real-time systems. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 438–443, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[25] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, Washington, DC, USA, 2006. IEEE Computer Society.

[26] H. Cho, B. Ravindran, and E. D. Jensen. Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373–384, 2007.

[27] H. Cho, B. Ravindran, and E. D. Jensen. T-l plane-based real-time scheduling for homogeneous multiprocessors. *J. Parallel Distrib. Comput.*, 70(3):225–236, 2010.

[28] H. Cho, B. Ravindran, and C. Na. Garbage collector scheduling in dynamic, multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(6):845–856, 2009.

[29] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *1993 Winter USENIX Conf.*, pages 127–146, 1993.

[30] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley. An adaptive, distributed airborne tracking system ("process the right tracks at the right time"). In *In IEEE WPDRTS, volume 1586 of LNCS*, pages 353–362. Springer-Verlag, 1999.

[31] R. K. Clark. *Scheduling Dependent Real-Time Activities.* PhD thesis, CMU, 1990. CMU-CS-90-155.

[32] U. C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 330–341, Washington, DC, USA, 2005. IEEE Computer Society.

[33] S. Fahmy. *Collaborative Scheduling and Synchronization of Distributable Real-Time Threads.* PhD thesis, Virginia Tech, 2010.

[34] S. Feizabadi, B. Ravindran, and E. D. Jensen. Msa: a memory-aware utility accrual scheduling algorithm. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 857–862, New York, NY, USA, 2005. ACM.

[35] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *USENIX Technical Conference*, pages 97–114, 1994.

[36] F. S. Foundation. Gnu general public license. `http://www.gnu.org/licenses/gpl.html`.

[37] K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 13–22, 2-4 2008.

[38] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 245–254, New York, NY, USA, 2009. ACM.

[39] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[40] P. Holman and J. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.

[41] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quaterly*, pages 21:177–185, 1974.

[42] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology – Portable Operating System Interface (POSIX) System Interfaces, Issue 6. 2001.* Open Group Technical Standard Base Specifications, Issue 6, 1992.

[43] E. Jensen, C. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems, 1985. IEEE RTSS, pages 112–122, 1985.

[44] E. Jensen and J. Northcutt. Alpha: a nonproprietary os for large, complex, distributed real-time systems. In *Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on*, pages 35–41, 11-12 1990.

[45] E. D. Jensen, A. Wellings, R. Clark, and D. Wells. The distributed real-time specification for java: A status report. In *Proceedings of The Embedded Systems Conference*, 2002.

[46] D. S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, 1974.

[47] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitioner's handbook for real-time analysis.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[48] G. Koren and D. Shasha. MOCA: a multiprocessor on-line competitive algorithm for real-time system scheduling. *Theor. Comput. Sci.*, 128(1-2):75–97, 1994.

[49] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 239–248, 1-3 2009.

[50] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms.* PhD thesis, Virginia Tech, July 2004.

[51] P. Li and B. Ravindran. Fast, best-effort real-time scheduling algorithms. *IEEE Trans. Comput.*, 53(9):1159–1175, 2004.

[52] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Trans. Comput.*, 55(4):454–469, 2006.

[53] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[54] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[55] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, CMU, 1986. CMU-CS-86-134.

[56] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28(1):39–68, 2004.

[57] R. Love. *Linux Kernel Development*. Novell Press, 2005.

[58] D. P. Maynard and S. E. Shipman. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project 88121, CMU CS Dept., December 1988.

[59] A. K.-L. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[60] I. Molnar. Config preempt realtime, fully preemptible kernel, vp-2.6.9-rc4-mm1-t4. `http://lwn.net/Articles/105948/`.

[61] OMG. Real-time corba 2.0: Dynamic scheduling specification. Technical report, Object Management Group, September 2001.

[62] L. Ramaswamy and B. Ravindran. A best-effort communication protocol for real-time broadcast networks. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*, page 519, Washington, DC, USA, 2002. IEEE Computer Society.

[63] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, page 114.1, April 2003.

[64] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[65] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[66] T. G. Tan and W. Hsu. Scheduling multimedia applications under overload and non-deterministic conditions. *IEEE RTSS*, 0:178, 1997.

[67] The Open Group. *MK7.3a Release Notes.* The Open Group Research Institute, Cambridge, Massachusetts, October 1998.

[68] N. M. Vallidis. *Whisper: a spread spectrum approach to occlusion in acoustic tracking.* PhD thesis, The University of North Carolina at Chapel Hill, 2002.

[69] J. Wang. Soft real-time switched ethernet: Best-effort packet scheduling algorithm, implementation, and feasibility analysis. Master's thesis, Virginia Tech, Sep 2002.

[70] J. Wang and B. Ravindran. Time-utility function-driven switched ethernet: Packet scheduling algorithm, implementation, and feasibility analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15(2):119–133, 2004.

[71] A. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley & Sons, 2004.

[72] H. Wu. *Energy-Efficient, Utility Accrual Real-Time Scheduling.* PhD thesis, Virginia Tech, July 2005.

[73] H. Wu, B. Ravindran, and E. D. Jensen. Utility accrual real-time scheduling under the unimodal arbitrary arrival model with energy bounds. *IEEE Trans. Comput.*, 56(10):1358–1371, 2007.

[74] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. *ACM Trans. Embed. Comput. Syst.*, 5(3):513–542, 2006.