FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time Scheduling on Multi-core: Theory and Practice

Paulo Manuel Baltarejo de Sousa

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Doctoral Program in Informatics Engineering

Supervisor: Prof. Eduardo Manuel Medicis Tovar

Second Supervisor: Prof. Luís Miguel Pinho Almeida

October 29, 2013

Real-Time Scheduling on Multi-core: Theory and Practice

Paulo Manuel Baltarejo de Sousa

Doctoral Program in Informatics Engineering

October 29, 2013

Abstract

Nowadays, multi-core platforms are commonplace for efficiently achieving high computational power, even in embedded systems, with the number of cores steadily increasing and expected to reach hundreds of cores per chip in near future. Real-time computing is becoming increasingly important and pervasive, as more and more industries, infrastructures, and people depend on it. For real-time systems too, multi-cores offer an opportunity for a considerable boost in processing capacity, at relatively low price and power. This could, in principle, help with meeting the timing requirements of computationally intensive applications which could not be met on single-cores. However, real-time system designers must adopt suitable approaches for their system to allow them to fully exploit the capabilities of the multi-core platforms. In this line, scheduling algorithms will, certainly, play an important role.

Real-time scheduling algorithms for multiprocessors are typically categorized as global, partitioned, and semi-partitioned. Global scheduling algorithms store tasks in one global queue, shared by all processors. Tasks can migrate from one processor to another; that is, a task can be preempted during its execution and resume its execution on another processor. At any moment, the m highest-priority tasks are selected for execution on the m processors. Some algorithms of this kind achieve a utilization bound of 100% but generate too many preemptions and migrations. Partitioned scheduling algorithms partition the task set and assign all tasks in a partition to the same processor. Hence, tasks cannot migrate between processors. Such algorithms involve few preemptions but their utilization bound is at most 50%. In semi-partitioned (or task-splitting) scheduling algorithms most tasks are fixed to specific processors (like partitioned), while a few tasks migrate across processors (like global). This approach produces a better balance of the workload among processors than partitioning and consequently presents higher utilization bounds, additionally, it also reduces the contention on shared queues and the number of migrations (by reducing the number of migratory tasks). However, it must ensure that migratory tasks never execute on two or more processors simultaneously.

For all of the categories of scheduling algorithms mentioned, it generally holds that the real-time scheduling theory is based on a set of concepts and assumptions that have little correspondence with practice. The gap between theory and practice is wide, and this compromises both the applicability but also the reliability. Therefore, we believe that efficient scheduling algorithm implementations in real-operating systems can play an important role in reducing the gap between theory and practice. We defined that (i) real-time theory cannot be detached from practical issues and (ii) the schedulability analysis must incorporate practical issues, such as context switch (or task switch) and task release overheads, just to mention a few. Therefore, the real-time theory must be overhead-aware in order to be reliable and must also consider the practical (in most cases, operating system-related) issues to increase its implementability in real operating systems.

In this dissertation, we deal with semi-partitioned scheduling algorithms that are categorized as slot-based task-splitting to achieve the above mentioned claims. To this end, we define a set of design principles to efficiently implement those scheduling algorithms in a real operating system (a PREEMPT-RT-patched Linux kernel version). Grounded on implementations of the slot-based task-splitting scheduling algorithms in the Linux kernel, we identify and model all run-time overheads incurred by those scheduling algorithms. We, then, incorporate them into a new schedulability analysis. This new theory is based on exact schedulability tests, thus also overcoming many sources of pessimism in existing analysis. Additionally, since schedulability testing guides the task assignment under the schemes in consideration, we also formulate an improved task assignment procedure. The outcome is a new demand-based overhead-aware schedulability analysis that permits increased efficiency and reliability. We also devise a new scheduling algorithm for multiprocessor systems, called Carousel-EDF. Although, it presents some similarities with slot-based task-splitting scheduling algorithms, we classify the Carousel-EDF scheduling algorithm as a reserve-based scheduling algorithm. The underlying theory is also overhead-aware and we also implement it in the Linux kernel.

Resumo

Hoje em dia, a utilização de plataformas multi-processador de modo a obter maior poder de computação é bastante comum e é expectável que o número de processadores atinja as centenas por plataforma num futuro próximo. Por outro lado, tem-se verificado um aumento da importância da computação de tempo real à medida que é mais usada na indústria, em diversas infraestruturas e mesmo pelas pessoas. Dadas as especificidades dos sistemas de tempo real, estas plataformas representam uma oportunidade para aumentar a capacidade de processamento a baixo custo e com baixo consumo de energia. Todo este poder computacional pode, em princípio, ajudar a satisfazer os requisitos temporais de aplicações com elevada exigência computacional que não consegue ser satisfeita por plataformas equipadas com um só processador. No entanto, na engenharia destes de sistemas de tempo real devem ser adoptadas abordagens adequadas por forma a tirar o máximo proveito destas plataformas multi-processador. Nesse sentido, certamente que os algoritmos de escalonamento de tarefas desempenham um papel importante.

Os algoritmos de escalonamento para sistemas de tempo real em plataformas multi-processador são, geralmente, categorizados como globais, particionados e semi-particionados. Os algoritmos de escalonamentos globais guardam todas as tarefas numa fila global, partilhada por todos os processadores. As tarefas podem migrar entre os processadores; isto é, a execução de uma tarefa pode ser interrompida (preemptada) num processador e recomeçar a execução noutro processador. A qualquer instante, as m tarefas com mais prioridade estão em execução nos m processadores que compõem a plataforma. Alguns algoritmos globais conseguem taxas de utilização de 100%, no entanto à custa de muitas preempções e migrações. Os algoritmos de escalonamentos particionados, dividem (criam partições) as tarefas que compõem o conjunto de tarefas pelos processadores. As tarefas atribuídas a um processador só podem ser executadas por esse processador. Desta forma, as migrações de tarefas não são permitidas e desta forma diminuem o número de preempções. Porém, a taxa de utilização é de 50%. Os algoritmos semi-particionados criam partições com a maior parte das tarefas (como os algoritmos particionados) e as restantes (poucas) podem migrar entre os processadores (como os algoritmos globais). Se por um lado, esta abordagem permite um melhor balanceamento das tarefas entre os processadores e consequentemente uma taxa de utilização maior. Por outro lado, reduz a disputa pelo acesso às filas partilhadas que armazenam as tarefas migratórias e o número de migrações (porque diminuem as tarefas que podem migrar). Contudo, estes algoritmos têm que assegurar que uma tarefa migratória não executa em dois ou mais processadores simultaneamente.

Geralmente, a teoria subjacente a todas as categorias de algoritmos de escalonamento para sistemas de tempo real descritas anteriormente é baseada num conjunto de conceitos e suposições que têm pouca correspondência com a prática; isto é, não têm em conta as especificidades das plataformas e dos sistemas operativos. A diferença entre a teoria e a prática é enorme e consequentemente compromete quer a sua aplicabilidade quer a sua confiabilidade. Portanto, nós acreditamos que implementações eficientes dos algoritmos de escalonamento num sistema operativo podem ajudar a diminuir a diferença entre a teoria e a prática. Assim sendo, nós definimos que (i) a teoria de sistemas de tempo real não pode estar dissociada dos detalhes de implementação e (ii) a análise de escalonabilidade deve incorporar os detalhes da prática, tais como, o custo associado à troca de tarefas no processador, ao aparecimento das tarefas no sistema, entre outras. Portanto, a teoria de sistemas de tempo real deve ter em conta os custos por forma a ser mais confiável e também mais implementável nos sistemas operativos.

Nesta dissertação, nós focamo-nos nos algoritmos semi-particionados, mais especificamente, num conjunto de algoritmos de escalonamento para sistemas de tempo real (para plataformas multi-processador) cuja principal característica é a divisão do tempo em reservas temporais. Com o propósito de alcançar os requisitos enumerados anteriormente, nós definimos um conjunto de princípios para uma implementação eficiente num sistema operativo (neste caso, numa versão do sistema operativo Linux melhorada com funcionalidades apropriadas para sistemas de tempo real). Fundamentado em implementações dos algoritmos em causa no sistema operativo Linux, nós identificamos e modelamos todos os custos operacionais associados à execução desses algoritmos no referido sistema operativo. Então, nós incorporamos todos esses custos operacionais numa nova análise de escalonamento. Esta nova teoria é baseada em testes de escalonabilidade exactos eliminando, desta forma, muitas fontes de pessimismo da análise existente. Esta nova análise permite uma melhoria no processo de criação de partições, isto é, mais tarefas por partição, logo menos partições. O resultado desta nova análise é um aumento da eficiência e da confiabilidade dos algoritmos. Também criamos um novo algoritmo de escalonamento para sistemas multi-processador, designado de “Carousel-EDF”. Este algoritmo apresenta algumas semelhanças com os algoritmos em consideração nesta dissertação. A teoria desenvolvida para este novo algoritmo também é baseada em testes de escalonabilidade exactos que também incorpora os inerentes custos operacionais.

Acknowledgements

The research work leading to this dissertation was developed at the Real-Time Computing Systems Research Centre (CISTER), from the School of Engineering of the Polytechnic Institute of Porto (ISEP/IPP).

I would like to express my thanks to Prof. Eduardo Tovar. He provided me crucial support to successfully develop my research work. Prof. Eduardo Tovar provided me constructive criticism and invaluable guidance that have significantly improved the quality of the work presented in this dissertation.

I am also thankful to Prof. Luís Almeida, for accepting to be my supervisor and also by the great effort in reviewing this dissertation.

Officially, I had two supervisors. Nevertheless, I would like to thank my third, unofficial, supervisor: Konstantinos Bletsas. I have no words to express my gratitude to his outstanding collaboration.

I would like to express my thanks to Prof. Pedro Souto for his insightful comments, discussions, and help, which far exceeded what anyone can ever ask for.

Thanks also to Nuno Pereira for his collaboration in part of the work and especially for his encouragement and friendship. I would like to express my gratitude and appreciation to all my co-authors.

I spent the last 10 years as CISTER collaborator. Therefore, I would like to thank all my colleagues in CISTER for all the great support at various levels and for their encouragement. My special thanks to Sandra Almeida and also to Inês Almeida for the administrative support.

I am teaching assistant at the Department of Computer Engineering of ISEP. I am deeply thankful to all the coordinators of the courses I teach (Luís Lino Ferreira, Luís Nogueira, Alexandre Bragança and Luís Miguel Pinho) for their support, comprehension, and, especially, for freeing me from some teaching-related labour.

I cannot forget my LaTeX helper and friend Paulo Matos, thank you.

Finally, I have to express my gratitude and, especially, my happiness for the unconditional love, continuous support, and patience that I have received from my lovely daughters, Rita and Matilde, and from my wife, Elisa. Elisa, this work that I am finishing now is one more step in our life's journey that we started 17 years ago. Without you, I would not be able to finish this work. I love you so much.

This work was supported by FCT (Fundação para a Ciência e Tecnologia) under the individual study grant SFRH/BD/46199/2008.

Paulo Baltarejo Sousa

"All things are difficult before they are easy"

Dr. Thomas Fuller

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Challenges 2
1.3 Approach 4
1.4 Thesis statement 6
1.5 Contributions 6
1.6 Outline 7

2 Background on multiprocessor platforms **9**
2.1 Introduction 9
2.2 Hardware in multiprocessor systems 9
2.2.1 Single-core processors 10
2.2.2 Multiprocessor systems 11
2.2.3 Sources of unpredictability 13
2.3 Software in multiprocessor systems 16
2.3.1 Linux kernel overview 17
2.3.1.1 Clock and timers 17
2.3.1.2 Tasks and run-queues in the Linux kernel 19
2.3.1.3 Linux modular scheduling framework 21
2.3.1.4 The real-time scheduling class of Linux 24
2.3.2 The Linux PREEMPT-RT patch 25
2.3.2.1 Priority inversion and Priority inheritance 25
2.3.2.2 Fully preemptible kernel 26
2.3.2.3 Locking primitives 27
2.3.2.4 Threaded ISRs 28
2.3.2.5 Scheduling policies 28
2.3.3 Real-time scheduling add-ons 29
2.3.4 Micro kernel-based approaches 30
2.4 Summary 33

3 Background on real-time systems **35**
3.1 Introduction 35
3.2 Basic concepts 35
3.2.1 System model 36
3.2.2 Fundamental concepts of scheduling 37
3.3 Review of relevant work on multiprocessor scheduling 40
3.4 Comparison of scheduling algorithms 43
3.4.1 Global scheduling 44

3.4.2 Partitioned scheduling 47
3.4.3 Semi-partitioned scheduling 48
3.4.4 Discussion 51
3.5 Historical perspective of slot-based scheduling algorithms 53
3.6 Summary 55

4 Slot-based task-splitting scheduling algorithms 57
4.1 Introduction 57
4.2 Assumptions about the hardware architecture 58
4.3 Generic slot-based task-splitting 58
4.4 Overview of specific scheduling schemes 61
4.4.1 Discussion of S-EKG and its original theory 61
4.4.2 Discussion of NPS-F and its original theory 62
4.4.3 Reasoning about the parameter δ 63
4.5 Instantaneous migration problem 64
4.6 Implementation of the slot-based task-splitting dispatcher 68
4.6.1 Scheduler implementation 69
4.6.2 Adding support for slot-based task-splitting scheduling algorithms to the RT scheduling class 72
4.7 Scheduling overheads 74
4.8 Summary 75

5 Slot-based task-splitting overhead-aware schedulability analysis 77
5.1 Introduction 77
5.2 Basis for the overhead-aware schedulability analysis 77
5.3 New demand-based and overhead-aware schedulability analysis 80
5.3.1 New demand-based schedulability test for mapping tasks to servers . . . 80
5.3.2 New demand-based schedulability test for assigning servers to processors 84
5.3.2.1 Non-split servers 85
5.3.2.2 Split servers 89
5.4 New server-to-processor assignment procedure 94
5.4.1 Assignment rules 94
5.4.2 Assignment procedure 95
5.4.2.1 Task to processor assignment procedure in S-EKG 96
5.4.2.2 New server-to-processor assignment for NPS-F 97
5.4.2.3 Effect of assignment rules on the schedulability analysis 98
5.5 Schedulability tests 100
5.6 Summary 101

6 Reserve-based scheduling algorithm 103
6.1 Introduction 103
6.2 Basics of the reserve-based scheduling algorithm 103
6.2.1 Off-line procedure 104
6.2.2 Run-time task-dispatching algorithm 105
6.3 Implementation of the Carousel-EDF task-dispatching algorithm 105
6.3.1 Scheduler implementation 106
6.3.2 Releasing jobs 107
6.4 Overhead-aware schedulability analysis 108
6.4.1 Server inflation 109

6.4.2 Generation of the carousel . . . 111
6.4.3 Supporting other job release mechanisms . . . 112
6.5 Utilization bound . . . 113
6.6 Upper bounds on preemptions . . . 116
6.7 Summary . . . 116

7 Evaluation 119
7.1 Introduction . . . 119
7.2 The off-line schedulability analysis . . . 119
7.2.1 Quantification of overheads . . . 119
7.2.2 Experiment design . . . 121
7.2.3 Preliminary evaluation of the merits of slot-based task-splitting over partitioning, when overheads are considered . . . 123
7.2.4 Evaluation of the new analysis in the absence of overheads . . . 125
7.2.4.1 Experiments for the S-EKG scheduling algorithm . . . 125
7.2.4.2 Experiments for the NPS-F scheduling algorithm . . . 128
7.2.5 Evaluation of the new analysis in the presence of overheads . . . 131
7.2.6 Reliability of the schedulability analysis . . . 132
7.2.7 Slot-based *vs.* reserve-based . . . 136
7.3 The run-time task-dispatching algorithm . . . 137
7.3.1 Experimental setup . . . 138
7.3.2 Why overheads have to be accounted for . . . 138
7.3.3 Enhancing real-time capabilities of the Linux kernel . . . 140
7.4 Summary . . . 143

8 Conclusions and future work 145
8.1 Introduction . . . 145
8.2 Summary of results . . . 145
8.3 Future work . . . 148

References 151

A Interrupts 161

B Other results 165
B.1 Off-line task-to-processor algorithm . . . 165
B.2 Run-time task-dispatching algorithm . . . 166

C Papers and materials 173
C.1 List of papers by the author . . . 173
C.2 Materials . . . 174

List of Figures

1.1 An example of the operation of a slot-based task-splitting multiprocessor scheduling. Task τ_2 is a split-task while tasks τ_1, τ_3, and τ_4 are non-split tasks that execute only on one processor. 5
1.2 An example of the operation of the hierarchical approach for slot-based task-splitting multiprocessor scheduling. 5

2.1 Illustration of a computer according to the von Neumann architecture. 11
2.2 Illustration of a multiprocessor architecture according to the shared memory model. 12
2.3 Illustration of a multiprocessor architecture according to the distributed memory model. 13
2.4 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). 14
2.5 Illustration of the use of an Advanced Programmable Interrupt Controller (APIC) in multiprocessor systems. 15
2.6 Linux-based computer system architecture. 17
2.7 Scheduling framework architecture. 22
2.8 RT scheduling class ready-queue. 25
2.9 Illustration of the priority inversion and priority inheritance. 26
2.10 Computer system architecture with the micro kernel layer. 30
2.11 Xenomai computer system architecture. 32

3.1 Illustration of the job timing parameters. 37
3.2 Illustration of Dhall's effect. 41
3.3 Illustration of the partitioning problem. 41
3.4 Global-EDF execution time-line. 44
3.5 Global EDZL execution time-line. 45
3.6 PF execution time-line. 46
3.7 DP-Wrap execution time-line. 47
3.8 EDF-WM execution time-line. 49
3.9 S-EKG execution time-line. 50
3.10 NPS-F execution time-line. 51

4.1 Task-to-server mapping. 59
4.2 Server-to-processor assignment. 60
4.3 Run-time dispatching time-line produced by the slot-based task-splitting algorithm for the task set example. 61
4.4 The intuition beyond δ parameter. 64
4.5 Illustration of the adjacent time slots that cause the instantaneous migration problem. 65

4.6 Illustration of the time slot shifting to solve the instantaneous migration problem. In spite of it is not shown in the figure there is no penalty for non-split servers; that is, the system supplies the same execution time to the non-split servers. . . . 65
4.7 Experimental evaluation of the impact of the cache and bus memory contention on the reading time of an array. These experiments highlight the benefit of the staggered against the aligned versions. 68
4.8 The required data used to implement the task-dispatching algorithm and its organization. 70

5.1 Illustration of the execution demand of a task against the processing supplied by the system. 79
5.2 Illustration of the execution demand of a task against the processing supplied by the system considering time reserves. 80
5.3 Illustration of release jitter and release overhead. 82
5.4 Illustration of the release interference for non-split servers. In this example server $\tilde{P}_q$ may suffer release interference from the arrivals of tasks served by $\tilde{P}_{q-1}$ and $\tilde{P}_{q+1}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_p). 86
5.5 Illustration of the reserve overhead. In this example, the execution of job $\tau_{i,j}$, inside reserve A, is delayed by $ResL_{i,j}$, the overhead of that reserve. 88
5.6 Illustration of the IPI latency at the release of a split job. It does not include the time for context switching. 90
5.7 Illustration of the potential release interference exerted by neighbouring servers. In this example server $\tilde{P}_q$ may suffer release interference from the arrivals of tasks served by $\tilde{P}_{q-1}$ and $\tilde{P}_{q-2}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_p) but also from the arrivals of tasks served by $\tilde{P}_{q+1}$ and $\tilde{P}_{q+2}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_{p+1}). 91
5.8 Illustration of the Ω and O^{fake} parameters. 92
5.9 Illustration of the assignment rule A1. 95
5.10 Illustration of the assignment rule A2. 96
5.11 Illustration of the S-EKG task-to-processor mapping. 96

6.1 Run-time dispatching time-line produced by the Carousel-EDF for the task set example presented in Section 4.3. 105
6.2 Data required to implement the Carousel-EDF scheduling algorithm in Linux kernel. 107
6.3 Illustration of the reserve latency. The $ResL$ incorporates the release of jobs that arrival time has elapsed. 110

7.1 Comparison of between NPS-F and P-EDF scheduling algorithms in the presence of overheads. 124
7.2 Comparison between the original S-EKG and the new schedulability analysis for task sets composed by mixed, $u_i \in [0.05, 0.95)$, tasks. 126
7.3 Comparison between the original S-EKG and the new schedulability analysis considering task sets composed by light, $u_i \in [0.05, 0.35)$, medium, $u_i \in [0.35, 0.65)$, and heavy tasks, $u_i \in [0.65, 0.95)$. 127
7.4 Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by mixed, $u_i \in [0.05, 0.95)$, tasks. 128
7.5 Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by light, $u_i \in [0.05, 0.35)$, medium, $u_i \in [0.35, 0.65)$, and heavy tasks, $u_i \in [0.65, 0.95)$. 129

7.6 Comparison between S-EKG and NPS-F scheduling algorithms according to the new demand-based schedulability analysis using δ equal to one. 130
7.7 Task-to-processor assignment patterns of the S-EKG and NPS-F. 131
7.8 Evaluation of the effect of the overheads on the new schedulability analysis. . . . 132
7.9 Multiple time slot lengths in a system. 133
7.10 Fraction of task sets schedulable according to the utilization-based analysis, but not schedulable according to the new demand-based overhead-aware schedulability analysis, as a function of the type of tasks of the task set and of the design parameter δ. 134
7.11 Fraction of mixed task sets that are considered schedulable by the original schedulability analysis but are not by the schedulability with overheads ($CpmdO$ equal to zero) 136
7.12 Comparison between the NPS-F and Carousel-EDF based on the normalized inflation. 137
7.13 A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met. 141
7.14 A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met. 142

B.1 Comparison of the new NPS-F demand-based overhead-aware schedulability analysis for short-, mixed-, and long-period task sets. 166
B.2 The mean and the standard deviation of the time slot length for short-, mixed-, and long-period task sets according to the type of tasks. 167
B.3 A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met for mixed-period, in the range [5, 250] milliseconds. 168
B.4 A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met for long-period, in the range [50, 250] milliseconds. 169
B.5 A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met for mixed-period (in the range [5, 250] milliseconds) task sets. 170
B.6 A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met for long-period (in the range [50, 250] milliseconds) task sets. 171

List of Tables

3.1 Task set example. 43

7.1 Experimentally-derived values for the various scheduling overheads (in microseconds). 120

List of Algorithms

5.1 Pseudo-code of the new task-to-server mapping algorithm. 84
5.2 Pseudo-code algorithm of the `inflate_sb_non_split` function. 89
5.3 Pseudo-code of the new server-to-processor assignment algorithm. 99
5.4 Pseudo-code algorithm of the schedulability test functions. 101
6.1 Pseudo-code algorithm of reserve sequence generator. 112
6.2 Pseudo-code algorithm of first reserve generator 113
7.1 Pseudo-code algorithm of the task set generator. 122

List of Code Listings

2.1 The required steps to implement a high resolution timer in the Linux kernel. . . . 20
2.2 The `struct task_struct` data structure. 20
2.3 The `struct rq` data structure. 21
2.4 The `rt_sched_class` definition. 23
2.5 The `schedule` function. 24
4.1 The C language code of the task used to investigate the impact of the contention for shared cache and also for memory bus. 67
4.2 Changes to the `enqueue_task_rt` function. 72
4.3 Changes to the `check_preempt_curr_rt` function. 73
4.4 Changes to the `pick_next_task_rt` function. 73
6.1 Timer callback function that release all jobs whose $a_{i,j}$ is earlier that the beginning of the current reserve. 108
7.1 The C language code of the task for experiments. 139

Acronyms and Symbols

ADEOS	Adaptive Domain Environment for Operating Systems
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
BF	Best-Fit
BKL	Big Kernel Lock
CFS	Completely Fair Scheduling
CPMD	Cache-related Preemption and Migration Delay
CPU	Central Processing Unit
dbf	demand bound function
DMA	Direct Memory Access
DP-Fair	Deadline Partitioning Fairness
EDF	Earliest Deadline First
EDF-SS	Earliest Deadline First scheduling of non-split tasks with Split tasks scheduled in Slots
EDF-US	Earliest Deadline First Utilization Separator
EDF-WM	Earliest Deadline First Window-constraint Migration
EDZL	Earliest Deadline first until Zero Laxity
EKG	Earliest deadline first with task splitting and K processors in a Group
FF	First-Fit
FIFO	First-In-First-Out
FPZL	Fixed Priority until Zero Laxity
GPOS	General Purpose Operating System
HPET	High Precision Event Timer
HRT	High-Resolution Timer
I/O	Input/Output
I/O APIC	Input/Output APIC
ILP	Instruction-Level Parallelism
IPI	Inter-Processor Interrupt
ISR	Interrupt Service Routine
LAPIC	Local Advanced Programmable Interrupt Controller
LITMUS^{RT}	LInux Testbed for MUltiprocessor Scheduling in Real-Time systems
LLF	Least-Laxity-First
LXRT	LinuX-RT
NF	Next-Fit
NP-hard	Non-deterministic Polynomial-time hard
NPS	Notional Processor Scheduling
NPS-F	Notional Processor Scheduling - Fractional
NUMA	Non-Uniform Memory Access

P-EDF	Partitioned Earliest Deadline First
P-Fair	Proportionate Fairness
PF	P-Fairness
PI	Priority Inheritance
PIT	Programmable Interval Timer
POSIX	Portable Operating System Interface
PREEMPT-RT	Linux kernel PREEMPT-RT patch
QPA	Quick Processor-demand Analysis
RESCH	REal-time SCHeduler
ReTAS	Real-time TAsk-Splitting scheduling algorithms framework
RM	Rate Monotonic
RM-US	RM Utilization Separator
RR	Round-Robin
RT	Real-Time
RTAI	Real-Time Application Interface
RTLinux	Real-Time Linux
RTOS	Real-Time Operating System
S-EKG	Sporadic Earliest deadline first with task splitting and K processors in a Group
sbf	supply bound function
SMT	Simultaneous MultiThreading
TSC	Time Stamp Counter
UMA	Uniform Memory Access
WF	Worst-Fit

Symbol	**Interpretation**	**Definition**
τ	A task set	
τ_i	The i^{th} task	
C_i	The worst-case execution requirement of task τ_i	See Section 3.2
T_i	The minimum inter-arrival time of task τ_i	See Section 3.2
D_i	The relative deadline of task τ_i	See Section 3.2
u_i	The utilization of task τ_i	See Section 3.2
n	The number of tasks of τ	
$\tau_{i,j}$	The j^{th} job of task τ_i	See Figure 3.1
$a_{i,j}$	The arrival time of job $\tau_{i,j}$	See Figure 3.1
$d_{i,j}$	The absolute deadline of job $\tau_{i,j}$	See Figure 3.1
$f_{i,j}$	The execution finishing time of job $\tau_{i,j}$	See Figure 3.1
P_p	The p^{th} processor	
$x[P_p]$	The processor P_p's x reserve length	See Equation 5.34
$N[P_p]$	The processor P_p's N reserve length	See Equation 5.34
$y[P_p]$	The processor P_p's y reserve length	See Equation 5.34
$U[P_p]$	The utilization of processor P_p	
m	The number of processors	
$\tilde{P}_q$	The q^{th} server	
$\tau[\tilde{P}_q]$	The set of tasks assigned to the server $\tilde{P}_q$	
$U[\tilde{P}_q]$	The utilization of server $\tilde{P}_q$	See Equation 4.1
$U^{infl}[\tilde{P}_q]$	The inflated utilization of server $\tilde{P}_q$	See Algorithm 5.2 and Section 6.4.1
$U^{infl}_x[\tilde{P}_q]$	Part of the inflated utilization of server $\tilde{P}_q$ assigned to the x reserve of the respective processor	See Equation 5.34
$U^{infl}_y[\tilde{P}_q]$	Part of the inflated utilization of server $\tilde{P}_q$ assigned to the y reserve of the respective processor	See Equation 5.34
k	The number of servers	
S	The time slot length	See Equations 4.2 and 5.41
δ	A designer-set integer parameter controlling the migration frequency of split tasks	See Section 4.4.3
α	The inflation factor of S-EKG	See Equation 4.2
Ω	The time interval between the two split server reserves	See Equation 4.8
U_s	Utilization of the system	See Equation 3.2
$\mathrm{UB}_{\mathrm{S-EKG}}$	Utilization bound of S-EKG	See Equation 4.3
$\mathrm{UB}_{\mathrm{NPS-F}}$	Utilization bound of NPS-F	See Equation 4.7
$U^{infl:orig}_{s:S-EKG}$	The cumulative processing capacity requirements for all (inflated) servers under the original analysis for S-EKG	See Equation 4.5
$U^{infl:orig}_{s:NPS-F}$	The cumulative processing capacity requirements for all (inflated) servers under the original analysis for NPS-F	See Equation 4.6

Symbol	Interpretation	Definition
Rule A1	The server should not be split	See Figure 5.9
Rule A2	The minimum separation between two reserves of the same server	See Figure 5.10
$RelJ$	A upper bound for the release jitter	See Section 4.7 and Figure 5.3
$RelO$	A upper bound for the release overhead	See Section 4.7 and Figure 5.3
$ResJ$	A upper bound for the reserve jitter	See Section 4.7 and Figures 5.5 and 6.3
$ResO$	A upper bound for the reserve overhead	See Section 4.7 and Figures 5.5 and 6.3
$CtswO$	A upper bound for the context (or task) switch overhead	See Section 4.7 and Figures 5.5 and 6.3
$ResL$	A upper bound for the reserve latency	See Section 4.7 and Figures 5.5 and 6.3
$IpiL$	A upper bound for IPI latency	See Section 4.7 and Figure 5.6
$CpmdO$	A upper bound of the CPMD overhead	See Section 4.7

Chapter 1

Introduction

1.1 Motivation

The computer era has been characterised by a continuous (and probably unfinished) *race* between processor manufacturers against processor consumers. Every new performance advance (permitting computational tasks to be executed faster by the processor) leads to another level of greater performance demands from consumers. Traditionally, performance advances were mainly achieved by increasing the *clock speed*. However, such increases have become limited by power consumption and heat dissipation [Low06].

In this context, *multi-core* architectures (consisting of multiple processing units or cores integrated in a single chip) offer an ideal platform to accommodate the recurrent increase of performance demands from consumers and solve the performance limitation of single-core architectures. The increase in processing capacity can be achieved by incorporating multiple cores in one single chip, with each core being able to run at a lower frequency, reducing heat and power consumption. Nowadays, multi-core platforms are commonplace for efficiently achieving high computational power [SMD$^+$10], even in embedded systems, with the number of cores steadily increasing, expected to reach hundreds of cores per-chip in the future [TIL12].

Nevertheless, what is the actual impact of multi-core platforms in current computing technology? In [Bak10], the author makes the following question: "*What will we make of these new processors?*". He then continues with: "*This question should be on the minds of the developers of every kind of software.*" This lead us to another question: do multi-core platforms represent an evolution or a revolution? We believe that multi-core platforms are more than an evolution, since system designers must adopt suitable approaches for their system that allow them to exploit the full capabilities of multi-core platforms. Therefore, this can imply giving up or, at least, reassessing the approaches followed in the context of single-cores during the last decades.

Real-time computing is becoming increasingly important and pervasive, as more and more industries, infrastructures, and people depend on it. For real-time systems too, multi-cores offer an opportunity for a considerable boost in processing capacity, at relatively low price and power.

This could, in principle, help with meeting the timing requirements of computationally intensive applications that could not be met on single-cores.

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [Sta88]; that is, the computation results must be delivered within a time bound, called *deadline*. Therefore, one of the most characteristic features of real-time systems is the notion of *time*. The time granularity is not necessarily the same across all real-time systems. In other words, microseconds could be suitable for some real-time systems, while for other, the timing requirement could be in the scale of milliseconds, seconds, minutes, or even hours. Therefore, the time granularity depends on the environment where the system operates.

Typically, real-time applications are composed by a set of tasks. Among other characteristics, each real-time task has an associated relative deadline. The results produced by each task must be delivered to the application during the *execution time window* defined by the *arrival time* (the time instant when a task becomes ready to be executed) of each task plus the respective relative deadline (which is typically different for each task). Naturally, each task requires some amount of time for computing its results (which is also typically different for each task). This imposes the use of some mechanism to schedule those tasks in the available processing units. In other words, real-time applications require a scheduling mechanism, usually denoted as *scheduling algorithm*, to create a task execution order such that their temporal constraints are satisfied. One frequent metric used to evaluate scheduling algorithms is the *utilization bound*, which is defined as a threshold for the task set workload such that all tasks meet their deadlines when the task set workload does not exceed such threshold.

Real-time applications are employed in a wide variety of fields. Given the particularities of some fields it must be assured, prior run-time, that the applications will not fail, especially, in application domains where a failure can have catastrophic consequences. This is assured by the *schedulability test*. The schedulability test establishes, at design time, whether the timing constraints are going to be met at run-time or not.

The real-time systems research community has developed a comprehensive toolkit comprised of scheduling algorithms, schedulability tests, and implementation techniques, which have been very successful: they are currently taught at major universities world-wide; they are incorporated in design tools and they are widely used in industry. Unfortunately, the big bulk of those results were limited to systems equipped with a single-core processor (or uniprocessor) only. Today, the increasing number of real-time systems running on multi-core (or multiprocessor) platforms impose the need for developing an analogous toolkit for those platforms.

1.2 Challenges

Multiprocessor scheduling algorithms have been categorized as *global*, *partitioned*, or *semi-partitioned*. Global scheduling algorithms store all tasks in one global queue, shared by all processors (or cores). At any time instant, the m highest-priority tasks among those are selected for execution on the *m* processors. Tasks can migrate from one processor to another during the execution; that is, an execution of a task can be stopped (or preempted) in one processor and resumed on another processor. Some scheduling algorithms [BGP95, BCPV96, AS00, AS04, CRJ06, RLM$^+$11] of this class present a utilization bound of 100%. However, the global shared queue imposes the use of some locking mechanisms to serialize the access to that queue. This compromises the scalability of global scheduling algorithms, as stated in [Bra11]: "*it would be unrealistic to expect global algorithms to scale to tens or hundreds of processors*". Another disadvantage of global scheduling is related to the fact that the higher number of preemptions and migrations could cause *cache misses*; that is, a failed attempt to read or write data from the processor cache, implying a main memory access that has a much higher latency.

In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same core (or processor) and tasks are not allowed to migrate from one processor to another. This class of scheduling algorithms presents a utilization bound of at most 50%. However, it conceptually transforms a multiprocessor system composed by *m* processors into *m* single-core (or uniprocessor) systems, thus simplifying the scheduling problem. Therefore, partitioned scheduling schemes require two algorithms: the *off-line* task-to-processor assignment algorithm and the *run-time* task-dispatching algorithm. The first one assigns tasks to processors and the second one schedules tasks at run-time to execute on the processor(s). However, assigning tasks to processors is a bin-packing problem, which is known to be NP-hard [CGJ97]. The task-dispatching algorithm schedules the statically assigned tasks using a uniprocessor scheduling algorithm, such as the Earliest Deadline First (EDF) [Liu69], which assigns the highest-priority to the most urgent ready task; that is, it assigns the highest-priority to the task with less remaining time until the end of its execution time window. One advantage of partitioned scheduling algorithms when compared with global scheduling algorithms, is the absence of migrations and respective overheads, namely those related to increased cache misses. However, the processor manufacturers have largely addressed this issue using a hierarchical cache approach. Therefore, this allows migrative scheduling approaches to be considered for real-time multiprocessor systems.

Hence, we share the same opinion as stated in [LFS$^+$10]: "*Newer multiprocessor architectures, such as multi-core processors, have significantly reduced the migration overhead. The preference for partitioned scheduling may no longer be necessary on these types of multiprocessors*". Semi-partitioned (or *task-splitting*) scheduling algorithms for multiprocessor systems try to solve or reduce the drawbacks and limitations presented by global and partitioned scheduling algorithms. Typically, under task-splitting scheduling algorithms, most tasks (called *non-split tasks*) execute on only one processor (as in partitioning) while a few tasks (called *split tasks*) use multiple processors (as in global scheduling). Note that it is *not* the code of such tasks that is split, but the execution requirement of such tasks. This approach produces a better balance of workload than partitioning. (Re-)using part of the processor capacity, which would otherwise be lost to bin-packing-related fragmentation, permits the design of scheduling algorithms with a higher utilization bound. Moreover, having just few migrating tasks limits the migration-related overheads and reduces (or removes altogether) the need for locking mechanisms, by not requiring system-wide global shared queues. As with partitioned schemes, semi-partitioned scheduling schemes are defined in terms of two algorithms: one for the task-to-processor assignment at design time and another for task-dispatching at run-time.

However, for all the categories of scheduling algorithms described above, it generally holds that the real-time scheduling theory is based on a set of concepts and assumptions that have little correspondence with practice, as stated in [Bak10]: "*Despite progress in theoretical understanding of real-time scheduling on multiprocessors, there is cause for concern about the validity of the theory for actual systems. The problem is that real multi-core processors do not fit the assumptions of the models on which the theory is based*". Therefore, we believe that efficient scheduling algorithm implementations in real operating systems can play an important role in reducing the gap between theory and practice. In other words, (i) real-time theory cannot be detached from practical issues and (ii) the schedulability analysis must incorporate practical issues, such as context switch (or task switch) and task release overheads, just to mention a few. Therefore, the real-time theory must be *overhead-aware* in order to be reliable. By incorporating those overheads in the schedulability tests we will further be able to validate the above proclaimed advantages of task-splitting algorithms in terms of efficiency.

1.3 Approach

In this dissertation, we deal with semi-partitioned scheduling algorithms that are categorized as slot-based task-splitting. These algorithms subdivide the time into *time slots*. Each processor time slot is composed by *time reserves* carefully positioned with a time offset from the beginning of a time slot. Time reserves are periodic and are used to execute tasks. Non-split tasks are executed within one periodic time reserve while split tasks are executed within two time reserves located on different processors. The positioning of the processor reserve in time is statically assigned (relative to the beginning of a time slot) so that no two reserves serving the same split task overlap in time.

In the course of this work, we developed the ReTAS (that stands for Real-time TAsk-Splitting scheduling algorithms) framework [ReT12]. The ReTAS framework implements, in a unified way, slot-based task-splitting scheduling algorithms in the Linux kernel. These implementations allow us, on one hand, to be aware of the specificities of operating systems in handling these types of scheduling algorithms and, on the other hand, to study such scheduling algorithms running in a real operating system. The conjugation of both allow us to make some improvements to the scheduling algorithms under consideration in order to overcome the difficulties imposed by the real operating system.

According to [SMD$^+$10], Moore's law (which projects that the density of circuits on chip will double every 18 months) still applies for multi-core architectures. Therefore, the number of processing units in multi-core architectures will increase exponentially with time. Hence, we believe it is important that an implementation of a multiprocessor scheduling algorithm has a task-dispatching overhead that is low as a function of the number of processors (ideally independent of the number of processors). To this end, the design principles for implementing slot-based task splitting algorithms must follow one important rule: *the interactions among processors must be reduced*. Let us provide an example scenario to help stressing this rule and also to provide an intuitive overview of the slot-based task-splitting scheduling algorithms. Figure 1.1 shows a typical time-line for a slot-based task-splitting scheduling algorithm. Time is divided into time slots of length S and each processor time slot is in turn divided into time reserves (denoted x, N, and y). Assume that task τ_2 is a split-task and executes on two processor reserves $y[P_1]$ and $x[P_2]$. Task τ_1 is a non-split task that executes on only one processor reserve, $N[P_1]$, and tasks τ_3 and τ_4 are also non-split tasks that share the same processor reserve, $N[P_2]$. Then, when both are ready to be executed, the scheduler selects the highest-priority task according to the EDF scheduling algorithm.

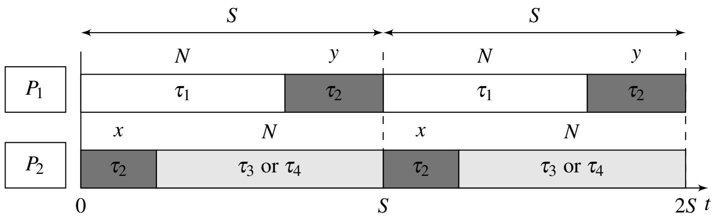


Figure 1.1: An example of the operation of a slot-based task-splitting multiprocessor scheduling. Task τ_2 is a split-task while tasks τ_1, τ_3, and τ_4 are non-split tasks that execute only on one processor.

In terms of implementation, a hierarchical approach can be used. Instead of assigning tasks directly to processors, tasks are first assigned to servers, and then, servers are instantiated by the means of processor reserves. Therefore, each server has a set of tasks (one or more) to schedule, and one or two processor reserves to execute such tasks. Each server, $\tilde{P}_q$, schedules its tasks according to the EDF scheduling algorithm. Figure 1.2 shows a typical time-line for slot-based task-splitting scheduling algorithm using the hierarchical approach based on servers.

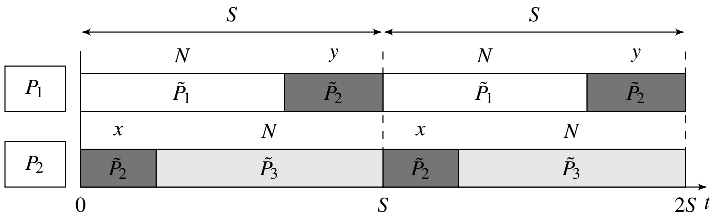


Figure 1.2: An example of the operation of the hierarchical approach for slot-based task-splitting multiprocessor scheduling.

We assume that each processor is able to setup a timer to generate an interrupt signal at a certain time in the future. Then, the time slot is independently managed by each processor (usually, all processor architectures provide such a timer mechanism). This timer is used to trigger scheduling decisions at the beginning of each reserve. Furthermore, we assume that processor clocks are synchronized (note that this is a hardware issue that is out of the scope of this work).

From Figures 1.1 and 1.2, we can identify that the most problematic issue to satisfy the aforementioned rule is the migration of the split-tasks. Whenever a split task consumes its reserve on processor P_p it has to *immediately* resume its execution on another reserve on processor P_{p+1}, we denote this issue as the *instantaneous migration problem*. From a practical point of view, that precision is not possible, due to many sources of unpredictability in a real operating system. Note that, even with precise time clock synchronization the problem could persist; for instance, an interrupt timer could be prevented to be fired on a specific time instant, and thus delayed, if the interrupt system or the respective interrupt is disabled. Consequently, the dispatcher of processor P_{p+1} can be prevented from selecting the task in consideration for execution because processor P_p has not yet relinquished that task. One solution could be for processor P_{p+1} to send an Inter-Processor Interrupt (IPI) to P_p to relinquish that split task. Another could be for P_{p+1} to setup a timer x time units in the future to force the invocation of its *dispatcher* (or *scheduler*).

We chose the latter option for two reasons: (i) we know that if a processor has not yet relinquished the split task it was because something is preventing it from doing so (e.g., the execution of an Interrupt Service Routine (ISR)); (ii) using IPIs in the way mentioned above introduces a dependency between processors that could compromise the scalability of the dispatcher. This approach allows us to define a new schedulability theory for slot-based task-splitting scheduling algorithms, and this new theory employs exact, processor demand-based, schedulability tests. Consequently, it makes it inherently more efficient than the original analysis for the respective algorithms, which employs utilization-based tests. We identify and model into the new analysis all types of scheduling overheads manifested under the considered scheduling algorithms. This renders the new, unified, schedulability analysis *overhead-aware*. Since schedulability tests are used to guide the off-line task-to-processor assignment in this class of algorithms, we also formulate a sophisticated overhead-aware accompanying task-to-processor assignment algorithm. This brings increased efficiency and reliability to slot-based task-splitting scheduling algorithms.

1.4 Thesis statement

In this dissertation, we address slot-based task-splitting scheduling algorithms for multiprocessor systems. We bridge the gap between theory and practice by adapting the schedulability theory so that it accounts for the overheads that multiprocessor scheduling algorithms incur in a real system. The central preposition of this dissertation is:

> ***Slot-based task-splitting scheduling algorithms can be implemented in a real operating system and work in such systems. However, the real-world constraints cannot be ignored by the underlying theory.***

1.5 Contributions

In this section, we overview the main contributions presented in this dissertation:

C1: The fundamental basis for implementing slot-based scheduling algorithms in a real operating system and the implementation of slot-based scheduling algorithms in the Linux kernel;

C2: A new schedulability analysis that takes into account the real-world overheads, with the run-time overheads included into it.

C3: The Carousel-EDF scheduling algorithm for multiprocessor systems as well as the related overhead-aware schedulability analysis with its implementation in the Linux kernel.

Like any other variant of the task-splitting scheduling algorithms, slot-based scheduling algorithms require an off-line task-to-processor assigning algorithm and a run-time task-dispatching algorithm. Contribution C1 concerns the task-dispatching algorithms. In other words, it concerns the implementation of the scheduling algorithm into the Linux operating system. We define the challenges and the design principles for implementing a slot-based task-splitting algorithm in the Linux kernel [SAT10a]. Additionally, we provide the first implementation of a slot-based task-splitting algorithm [SAT10b, SAT11], an invaluable contribution in demonstrating the capabilities and limitations of new multiprocessor scheduling techniques on actual hardware. We further evolve those implementation to support, in a unified way, slot-based task-splitting scheduling algorithms [SBTA11] into mainline Linux and later in a PREEMPT-RT-patched [PR12] Linux kernel [SPT12].

Contribution C2 concerns the off-line procedure. We define a new scheduling theory that incorporates the scheduling overheads of a slot-based task-splitting algorithm [SBAT11]. We also define a unified overhead-aware schedulability analysis based on processor demand [SBT$^+$13].

Contribution C3 concerns the Carousel-EDF, a new hierarchical scheduling algorithm for a multiprocessor systems, and its overhead-aware schedulability analysis based on demand-bound functions [SSTB13].

As an overarching contribution, we show that considering real-world overheads is fundamental for the efficiency and especially for the reliability of the schedulability tests.

1.6 Outline

The rest of this dissertation is organized as follows. Chapter 2 and Chapter 3 are devoted to further providing the relevant context and motivation. The former deals with system platforms, namely hardware and operating system issues. The latter focuses on real-time scheduling concepts and methodologies. From Chapter 4 to Chapter 7, we present our innovative research work.

In Chapter 4, we describe in a unified way the slot-based task-splitting scheduling algorithms under consideration in this work, creating a system model that is used throughout this dissertation. We describe the implementation of those algorithms according to such model into the PREEMPT-RT-patched Linux kernel. We also identify and model the scheduling overheads incurred by those algorithms when running in a real system.

Chapter 5 presents the new overhead-aware and demand-based schedulability analysis. These two features of the new schedulability analysis increase the reliability and efficiency. Incorporating the scheduling overheads into the schedulability analysis allow us to reduce the unrealistic assumptions of the original analysis. This new theory is based on exact schedulability tests thus also overcoming many sources of pessimism in existing analysis.

Chapter 6 presents Carousel-EDF, a new hierarchical scheduling algorithm for a system of identical processors, and its overhead-aware schedulability analysis based on demand bound functions. Carousel-EDF presents some similarities with slot-based scheduling algorithms under consideration in this work. In a high-level view, it is a time-driven algorithm like slot-based scheduling algorithms, but contrarily to them, Carousel-EDF creates only one time reserve for each server. As a result, it reduces by up to 50% the number of preemptions incurred by the time managing mechanism. This is achieved through a time reserve rotation scheme. In a low-level view, it is event-driven, since it schedules tasks according to the EDF scheduling algorithm like slot-based scheduling algorithms. We denote Carousel-EDF as a reserve-based scheduling algorithm as opposed to the slot-based scheduling algorithms. We also describe the implementation of the Carousel-EDF into the PREEMPT-RT-patched Linux kernel.

Chapter 7 evaluates the effectiveness and the reliability of the new schedulability theory developed for slot-based task-splitting scheduling algorithms by comparing with the original theory. This evaluation is based on an extensive set of experiments focused on the task-to-processor assignment algorithm and also on the run-time task-dispatching algorithms. In this chapter, we also compare slot-based with reserve-based scheduling algorithms, and also highlight in detail the realities behind what amounts to lack of support for *any* real-time scheduling algorithm, by the present Linux kernel.

Finally, in Chapter 8, we provide a summary of the research results developed in the context of this work and outline future research lines.

Chapter 2

Background on multiprocessor platforms

2.1 Introduction

A processor is one of the computer components, arguably the fundamental one. A processor has the ability to execute instructions; an instruction is a single operation that given an input produces an output, and the output of an instruction can be the input of another. Then, it is possible to create a sequence (or list) of instructions logically ordered that performs some kind of computation; this is usually denoted by a computer *program*. In the computer's jargon, the set of all physical devices such as processor, memory, and graphic card, for example, is denoted by *hardware* and the collection of programs is denoted by *software*.

In this chapter, we will review platform-related aspects that are relevant to the research described in this dissertation. We start by devoting some attention to hardware architecture topics, focusing on the hardware characteristics that contribute to the unpredictability of the system. Then, we pursue with software. Important operating system details are reviewed, and Linux and Linux-based real-time operating systems will be described. An initial big picture of the Linux kernel will be provided, with special attention given to the features that are relevant to satisfying the requirements of real-time systems. We will also review different approaches employed by Linux-based real-time operating systems. Because the Linux PREEMPT-RT patch is used as base for our implementation work, a detailed description of the features that it introduces to the mainline Linux kernel is presented.

2.2 Hardware in multiprocessor systems

Some relevant computer architecture concepts will be exposed in this section. As a starting point, we overview the organization of single-core processor systems. This provides us with some background to the evolution into multiprocessor architectures, which we briefly overview next.

Several architectural advances introduced in modern single processor and multiprocessor systems such as memory caches, pipelines, and others were designed to increase the average performance. While, in average, these advances can provide significant performance improvements, they introduce significant unpredictability and make the task of providing timing guarantees much harder. We will discuss two important hardware features that introduce unpredictability and are of special relevance to the implementation of real-time scheduling algorithms: caches and interrupts.

2.2.1 Single-core processors

Briefly discussing single-core processor systems is useful to introduce some relevant hardware aspects. Therefore, we start by describing a generic overview of a single-core processor platform (see Figure 2.1). Many computational systems are based on von Neumann's architecture [EL98]. According to this architecture, a computational system consists of a CPU (that stands for Central Processing Unit), memory, and Input/Output (I/O) devices interconnected by a single bus. The CPU is the component responsible for executing applications. Applications are composed by data and instructions that are both stored in memory. Therefore, a CPU executing some application needs to fetch that application's information (data and instructions) from the main memory. As it is intuitive, the application execution speed depends not only on the CPU processing capacity, but also, on the memory access times.

From the birth of CPUs, back in the early 1970's, until nowadays, the CPU evolution has been tremendous; probably, there is no other comparable example in the engineering world. The increase of performance (measured by the amount of time to execute a given task) has been achieved via the CPU speed increases, Instruction-Level Parallelism (ILP) [RF93, Pai00] techniques, pipelines and memory caches.

The increase of CPU speed decreases the duration of clock cycle (in one clock cycle, a CPU can perform a basic operation such as, for instance, fetching an instruction). One of the ILP techniques used to increase the overall efficiency of a CPU is Simultaneous MultiThreading (SMT). SMT makes a single physical CPU system appear as multiple CPUs. For that purpose, in a physical CPU there are multiple logical CPUs. Each logical CPU is constituted by an almost complete set of CPU hardware; including registers, pipelines, and other elements, but sharing the execution unit. This provides a way for keeping, at the hardware level, multiple threads (or tasks) ready to be executed, thus reducing the context switch (or task switch) cost; that is, the cost of switching the execution unit from one task to another. From the operating system perspective, a SMT system is composed by many CPUs. However, in practice only one task can be executed at a given time. Hence, a SMT system is, actually, a single-CPU system. One common implementation of this is the *hyperthreading* [MBH$^+$02] by Intel.

The accesses to the main memory are too slow comparatively to the CPU speed, which causes the CPU to stall while waiting for the information to be returned. The main purpose of caches is to reduce the memory access latencies. A cache is a smaller and faster memory that stores copies of the application's data/instructions recently used by CPU. Caches are typically organized in a hierarchy of increasingly larger and slower caches, as follows. Whenever the CPU needs to read from or write to a location in main memory, it first checks the level 1 (L1) cache; if it hits (if the information is stored in L1 cache), the CPU proceeds. If the L1 cache (the fastest in terms of access) misses (if the information is not stored in L1 cache), the next larger cache (L2) (slower than L1) is checked, and so on, before main memory is checked. Whenever the required information is not stored in any of the previously mentioned memory units (caches and main memory) the CPU has to retrieve the information from an I/O device, for instance the hard disk that takes orders of magnitude more time to access than even main memory.

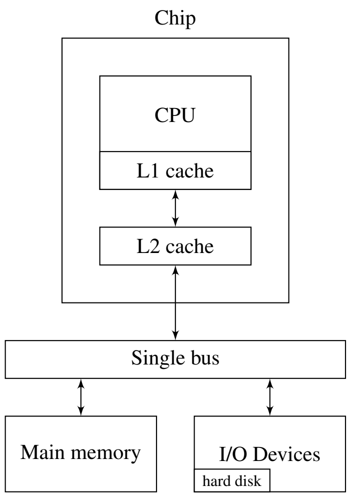


Figure 2.1: Illustration of a computer according to the von Neumann architecture.

2.2.2 Multiprocessor systems

During the last decades, each generation of CPUs grew smaller, faster, and computationally more powerful. However, each generation also dissipated more heat and consumed more power, which ultimately forced the CPU manufacturers to shift from single-core to multi-core processor architectures [GK06, Low06].

Before the multi-core processor architecture, which integrates several CPUs on a single chip (see Figure 2.2), there was no terminological distinction between CPU and processor. Generally, in the literature, the term *processor* was used to refer to a single-core system as opposed to the term *multiprocessor* that was used to refer to systems with two or more CPUs. In this dissertation, we use the term multiprocessor whenever we refer to more than one CPU (whether or not these CPUs are on the same chip) and processor whenever we refer to only one CPU. The term *uniprocessor* is used whenever we refer a system composed by only one CPU.

Note that, multiprocessor architectures are not a new concept and there is no single common architecture for multiprocessor systems. A multiprocessor can be classified according to the types of processors that compose it. A multiprocessor is said to be: *identical*, *heterogeneous*, or *uniform*. Identical, if all processors have the same features: clock speed, architecture, cache size and so on; Heterogeneous, if they have different capabilities and different speeds. Finally, in uniform multiprocessors, one processor may be x times faster than another one, with the meaning that it executes *all* possible tasks x times faster.

According to the memory model, multiprocessor systems are classified as: *shared* or *distributed* memory. In a shared memory model (see Figure 2.2) all processors share the main memory. One common implementation of the shared memory model is the multi-core architecture, which is now the new direction that manufacturers are focusing on [SMD$^+$10]. One important performance concern in a multi-core architecture is the communication among different components: CPUs, caches, memories, and I/O devices. Initially, such communication was done through a shared bus (see Figure 2.2). However, due to high latency and contention in the shared bus, processor manufacturers are instead developing advanced communication mechanisms. Examples of those mechanisms are: AMD HyperTransport, Intel Quickpath and Tilera iMesh, just to mention a few. Independently of the mechanism used to connect the main memory to processors, in the shared memory model, access times to that memory are uniform from any processor. This is known as Uniform Memory Access (UMA).

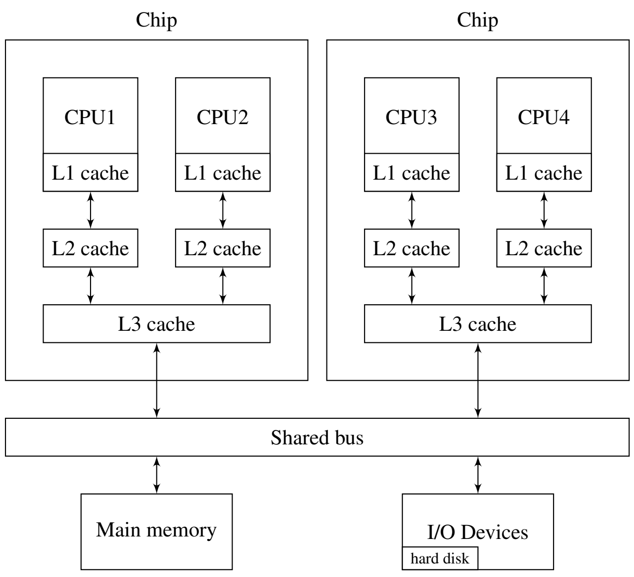


Figure 2.2: Illustration of a multiprocessor architecture according to the shared memory model.

In the distributed memory model (see Figure 2.3) each processor has its private memory and when remote data is required the processor has to communicate with respective remote processors, through an interconnection network. This architecture benefits from its modular organization with multiple independent processors; that is, only the interconnection network is shared among processors. Because this makes adding new processors very easy, this architecture is straightforward to scale. The major drawback of the distributed model is that inter-processor communication is more difficult. Whenever a processor requires data from another processor's memory, it must exchange messages with that processor. This introduces two sources of overhead: (i) it takes time to construct and send a message from one processor to another; and (ii) the receiving processor must be interrupted in order to deal with messages from other processors.

There is other variant of the distributed memory model called *distributed shared* memory. In this architecture, data sharing is implicit; that is, from the programmer's perspective, the distributed memory appears like a real shared memory model, because there is no need to invoke send and receive primitives. This architecture logically implements a shared memory model although the memory is physically distributed. The memory access times in these systems depend on the physical location of the processors and are not uniform. Due to this characteristic, these systems based on distributed memory model are termed Non-Uniform Memory Access (NUMA) systems.

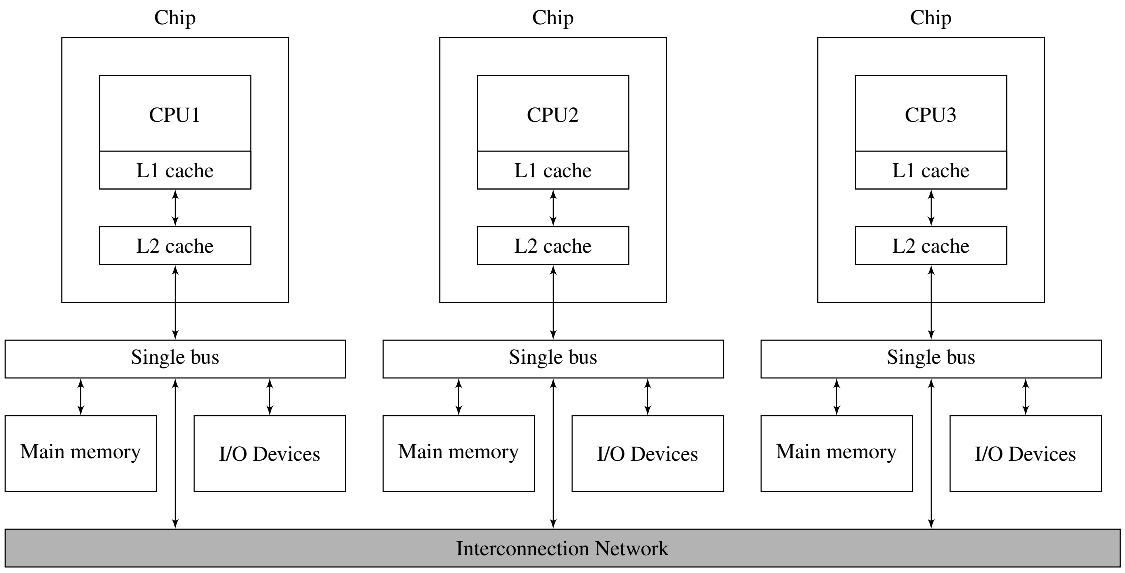


Figure 2.3: Illustration of a multiprocessor architecture according to the distributed memory model.

Note that, from the scheduling point of view, the main difference between the shared and distributed memory models concerns the costs associated with task migration, the process of transferring a ready (or runnable) task from one processor to another. In a distributed memory model, task migration requires the transfer of the task context and the memory contents that describe the state of the transferring task. Thus, task migration could come at the cost of a performance penalty and frequent task migration could contribute to the increase of intercommunication network traffic. However, if a task is accessing a huge amount of data stored in a remote processor memory, it could be more efficient, in terms of performance, to move such a task to the remote processor instead of transferring data to the task's processor. In a shared memory model, task migration instead requires only the transfer of the task context.

2.2.3 Sources of unpredictability

Designing and implementing a system to provide real-time timing requirements requires appropriate management of all sources of unpredictability in the system. At the hardware level, there are a few sources of unpredictability such as ILP (e.g. speculative execution and branch prediction), pipelines, bus access contention and caches. All these hardware features make the execution path of tasks non-deterministic, thus determining the execution time of a task becomes very difficult, and actually, this is a whole area of research [WEE+08].

Caches are a very useful hardware feature that is used to increase the performance of the processor by reducing the amount of time the processor needs to wait for data. From a real-time perspective though, caches are sources of unpredictability, in the sense that, the execution time of a task varies, depending on the data that is currently stored in the cache. The existence of distributed caches (such as when some cache levels are private to each processor and others to the chip) cause what is known as the *cache coherence problem*; that is, stored data in one processor cache might not be the most up-to-date version. Figure 2.4 illustrates, in a simplistic way, the cache coherence problem (this example was taken from [HP06]). Let us assume that at time t_0, caches of CPU A and B are empty and the content of memory location X is 1. At time t_1 CPU A reads the content of memory location X, and then its cache stores its value. Next, at time t_2, CPU B also reads the content of memory location X and its cache also stores its value. At time t_2, according to the informal definition presented by [HP06] ("*Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item.*") the memory system is coherent. But, at time t_3 CPU A stores 0 into memory location X. This operation updates the value of X in the cache of CPU A and also in main memory, but not in the cache of CPU B. If CPU B reads the value of X, it will receive 1. This problem can be addressed using two approaches: snooping- and directory-based protocols [HP06]. Cache coherence protocols are intended to manage these issues in order to maintain the memory system coherent.

Time	**Event**	**Cache contents for CPU A**	**Cache contents for CPU B**	**Memory contents for location X**
t_0				1
t_1	CPU A reads X	1		1
t_2	CPU B reads X	1	1	1
t_3	CPU A stores 0 into X	0	1	0

Figure 2.4: The cache coherence problem for a single memory location (X), read and written by two processors (A and B).

Usually, real-time system designers consider the worst-case execution time of a task by taking into account the delay incurred by fetching data from main memory; that is, assuming that the respective information is not inside the caches. However, this is not enough to solve the unpredictability caused by caches. For instance, cache misses due to task preemptions are not considered.

Interrupts are another source of hardware-related unpredictability that is very relevant to the context of this research. Naturally, processors have to interact with I/O devices (e.g. hard disk drives, graphics or network cards), which are usually much slower than the processor. Then, if a processor issues a request to one device, waiting for the device's response would imply that the processor is idle while waiting for the response. The usual approach to deal with this is: the processor issues the request, and continues to perform other work; when the device completes the request, the processor handles the device's response. However, given that it is typically not trivial to know beforehand how much time the device takes to handle the processor's request, it is hard to know when the device will have completed the request. One approach to this problem is *polling*. Periodically, the processor checks if the device has issued some response. This incurs overhead, because the polling occurs repeatedly at regular intervals, which might be much smaller than the response time of the device. Another, usually preferred, approach is based on *interrupts*. The processor is linked to the devices by interrupt lines, and whenever a device needs to notify a processor, it signals the respective interrupt line. Then, the processor acknowledges the interrupt issuer (in order to free the interrupt line), saves the context of the current executing task, loads the information about the Interrupt Service Routine (ISR) associated to that interrupt, and executes such ISR. When the ISR completes, the previously saved context of the current task is restored, and its execution is resumed from the point of the interruption. Note that, hardware devices are not the only source of interrupts, because software modules are also able to issue interrupts that are managed in a similar way.

Figure 2.5 shows a simplified illustration of Intel's approach [Int13] to handling interrupts on multi-core systems designated as Advanced Programmable Interrupt Controller (APIC). It is composed by the Local-APIC (LAPIC) and I/O APIC interconnected by a bus (which, depending on the processor's model, could be shared or dedicated). Each processor is equipped with a LAPIC. A LAPIC is able to generate and accept interrupts and is also a timer. A system has at least one I/O APIC that manages the external interrupt requests. Whenever a device issues an interrupt request, the I/O APIC receives that request and delivers the signal to the selected LAPIC via the bus. Then, LAPIC issues an interrupt to its processor. Another functionality is the possibility to send Inter-Processor Interrupts (IPIs). When a processor needs to signal some processor or processors, it has to issue a signal to the target LAPIC via the bus.

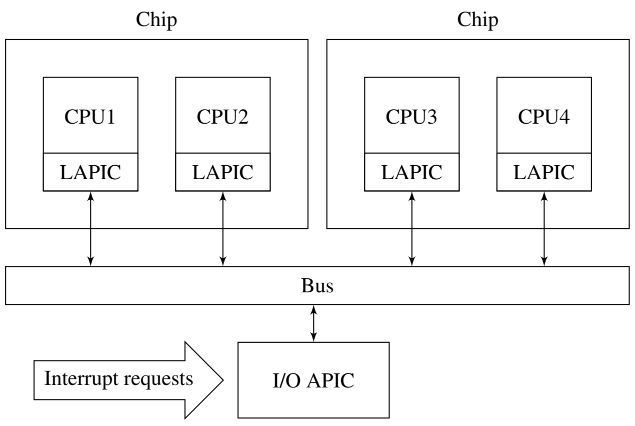


Figure 2.5: Illustration of the use of an Advanced Programmable Interrupt Controller (APIC) in multiprocessor systems.

Typically, the system designer does not know which is the frequency of all interrupts. Some interrupts occurs at a regular rate, but others occur in a unpredictable fashion. As it is intuitive, the interrupts contribute to the unpredictability of the system. Given the complexity of getting information about interrupts, most of the schedulability tests, simply, do not consider the cost of interrupts.

2.3 Software in multiprocessor systems

"*Without its software, a computer is basically a useless lump of metal*". This is the first sentence of the well known reference book "Operating Systems Design and Implementation" [TW08], which effectively conveys that a computational system (a computer) is composed of both hardware and software. It is common to classify the software of a computer into two broad categories: *application software* and *system software*. The terms *user-space* and *kernel-space* are also used to distinguish the running context of software in a computational system. Application software concerns with user-space, while system software concerns with kernel-space. Here, we focus on system software and the most important of such software is the so-called *operating system*. The operating system is responsible for managing the hardware as well as all other software; it controls and coordinates the use of the hardware among application and system software. For security purposes, it can also provide some protection level for executing application software.

Given its role, the operating system significantly affects the overall performance of the system. Hence, some operating systems were designed for specific domains, such as Real-Time Operating Systems (RTOSs), while others are intended to be General Purpose Operating Systems (GPOSs).

One of the most known and successful GPOS is Linux [Fou13], whose system architecture is illustrated in Figure 2.6. The term *Linux* typically refers to a system composed of the Linux kernel and a set of user-space tools and libraries. Every single day, 850 thousands of mobile phones, 700 thousands of televisions, and nine out of ten of the world's supercomputers run Linux [Fou13]. In the following subsections, we review the main features of the Linux kernel considered fundamental in the context of this work.

The main goal of this work is to provide better real-time support for multiprocessor systems, and for that purpose, the underlying operating system must be *predictable*, which implies that its latencies must be predictable or at least time bounded. Due to its success and open source nature, several relevant efforts towards enabling real-time computing using the Linux kernel have been undertaken [RTA12, Xen12, LR12, Kat12, SCH12, PR12], and later in this section we will describe these efforts.

One effort to introduce better real-time support in the Linux kernel – the PREEMPT-RT patch – aims at increasing the preemptibility of the Linux kernel and is an important advance towards better real-time support in the Linux kernel [PR12]. This effort is of special relevance for the research described in this dissertation, as a significant effort will be put into modifying the PREEMPT-RT-patched Linux kernel to support real-time slot-based task-splitting scheduling algorithms. Taking the PREEMPT-RT patch as a starting base for our work allows to leverage the efforts of a large community of kernel developers and produce a system with better real-time properties than the mainline Linux kernel.

Another type of efforts, also described in this section, have aimed to introduce to the Linux kernel support for several real-time scheduling algorithms (especially for multiprocessor systems) and some resource sharing protocols suited for real-time systems [LR12, Kat12, SCH12]. There exist several of these efforts, most of them from academia and used as testbed frameworks or proof-of-concepts for theoretical work.

We distinguish a third class of approaches to introduce real-time properties into the Linux kernel [Yod99, RTA12, Xen12], which are also described in this section. These are the micro kernel (or co-kernel) approaches, which make use of the Linux kernel for executing non-real-time tasks and other management operations and use a co-real-time kernel for real-time purposes.

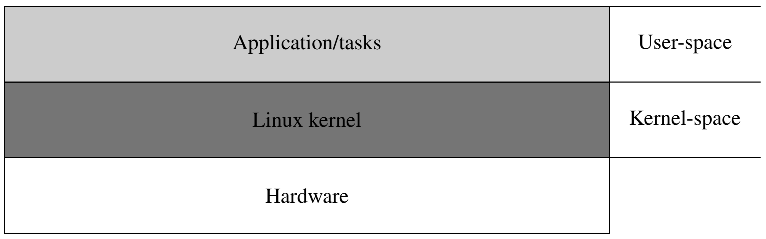


Figure 2.6: Linux-based computer system architecture.

2.3.1 Linux kernel overview

Linux is an open-source project, for which the development process follows a collaborative approach, with thousands of developers worldwide constituting the largest collaborative development project in history of computing. Due to its wide adoption, Linux is well positioned to take an important role enabling real-time computing in modern multiprocessor systems.

Since we are interested in leveraging Linux to support real-time slot-based task-splitting scheduling algorithms, in this section, we will overview the relevant features of the Linux kernel, needed to implement those scheduling algorithms. More specifically, we will see how timing is kept by the Linux kernel, and how the Linux kernel allows the implementation of a new task scheduler.

2.3.1.1 Clock and timers

Time management is an important issue in the Linux operating system [BC05, Mau08], as well as in other operating systems. There are a lot of kernel activities that are time-driven, such as balancing the processor's ready-queues (referred to as *run-queues* in the Linux documentation), process resource usage accounting, profiling, and so on. In the context of this work, it is important to keep accurate and precise timing to implement the slot-based task-splitting scheduling algorithms.

The Linux kernel time management system is based on some hardware devices that have oscillators of a known fixed frequency, counters, and comparators. Simpler devices increment their counters at fixed frequency, which is in some cases configurable. Other devices decrement their counters (also at fixed frequency) and are able to issue an interrupt when the counter reaches zero. More sophisticated devices are provided with some comparison logic to compare the device counter against a specific value, and are also able to interrupt the processor when those values match. Some examples of such devices are: (i) the Programmable Interval Timer (PIT), which is a counter that is able to generate an interrupt when it reaches the programmed count; (ii) the High Precision Event Timer (HPET), which consists of a counter and a set of comparators. Each comparator has a match register and can generate an interrupt when the associated match register value matches with the main counter value. The comparators can be put into *one-shot* mode or *periodic* mode. In one-shot mode, the comparator generates an interrupt once, when the main counter reaches the value stored in the comparator's register, whereas in the periodic mode the interrupts are generated at specified intervals; (iii) the Time Stamp Counter (TSC) is a processor register that counts the number of cycles since reset; and, finally (iv) the LAPIC, which is part of the processor chip. As mentioned before, the LAPIC manages all interrupts received by the respective processor either from external devices or internally generated by software. For that purpose, it is provided with mechanisms for queueing, nesting, and masking interrupts. Furthermore, it is able to send interrupts for external devices as well as to generate interrupts for the respective processor.

As would be expectable, the Linux kernel is able to interact with all of these devices. In order to get better time accuracy, the Linux kernel gathers timing information from these devices, associates a quality metric to each device and normalizes the values to the same time unit.

In multiprocessor systems time management could be even more complex. In a multiprocessor, the different processors can have different notions of time and this is an important problem for the scheduling algorithms under consideration in this work, as they assume a common notion of the time slots across processors. The assumption of this work is that this is something managed at the hardware level. This is a reasonable assumption as, for example, recent multiprocessor chips from Intel implement what is known as *invariant TSC*, which guarantees that the TSC is synchronized across all cores and runs at constant rate for all processors in a single package or mainboard [Int13].

Independently of the way that the time is managed, the Linux kernel employs a periodic timer interrupt to get the control of the system, called *tick*, which can be viewed as the kernel's heartbeat. The frequency of the this periodic timer interrupt (the tick rate) is defined by a kernel compilation option called `CONFIG_HZ`. In the kernel code, this macro is only referred to by `HZ`, an acronym for hertz. Depending on the value of `HZ`, the *resolution* of the tick (the smallest time difference between consecutive ticks) is more or less frequent. Unfortunately, these standard resolutions supported by the periodic timer are too coarse-grained for the scheduling algorithms under consideration in this work. From the panoply of options provided by Linux kernel the most suitable option for the implementation of the desired scheduling algorithms is the High-Resolution Timer (HRT) framework [GN06]. On one hand, HRT provides a nanosecond resolution and, on the other hand, the timers are set per-processor.

Next, we will describe how to implement a HRT. Conceptually, the use of a kernel timer encompasses three phases: (i) the initial phase is for setting up the timer, by specifying a function (generally called *timer callback*) to execute at timer expiration; (ii) the second phase is the activation of the timer, when its expiration time is specified; and (iii) the third phase occurs at timer expiration, when the specified function executes. Listing 2.1 shows a snippet of the required code for implementing a HRT in the Linux kernel. First, we have to define per-processor a variable of the type `struct hrtimer` and initialize that variable. For that purpose, we use the `DEFINE_PER_CPU` Linux kernel macro that defines, in this case, a variable of a type `struct hrtimer` on every processor, called `my_hrtimer`; variables defined in this way are actually an array of variables. Then, in order to initialize `my_hrtimer`, it is required to invoke, for each processor, the `my_hrtimer_init` function. Note that, it is not possible to execute this function on one processor to initialize a timer on another processor. The Linux kernel macro `smp_processor_id` returns the current processor number, between 0 and `NR_CPUS` (Linux kernel macro that defines the maximum number of processors). To get a particular processor's variable, the `per_cpu` macro may be used for. Then, the `hrtimer_init` function initializes a timer, `my_hrtimer`, to a given clock (in this case, using the monotonic clock) and the last parameter specifies if absolute or relative time values (relative to the current time) are used (in this case, absolute).

In the second phase, the timer is started in a per-processor basis; that is, the `my_hrtimer_start` function has to be executed on each processor. The timer expiration has to be specified.

Finally, when the timer fires, the `my_hrtimer_callback` function is invoked. Note that, the argument of the `my_hrtimer_callback` function is the timer itself. Two values could be returned by the `my_hrtimer_callback` function: `HRTIMER_RESTART` and `HRTIMER_NORESTART`. The timer is automatically restarted with the specified expiration at `next_expire` if it returns `HRTIMER_RESTART`, while the timer is not automatically restarted if it returns `HRTIMER_NORESTART`.

Note that, if the purpose of the timer is to trigger the invocation of the scheduler, then, the timer callback function must mark the currently executing task for preemption (invoking the Linux native `resched_task` function). Hence, after the execution of the timer callback the Linux scheduler is invoked to take scheduling decisions. Scheduling decisions are taken according to some specific scheduling policy. Later in this section, the Linux scheduling framework will be described; that is, how the scheduler is implemented in the Linux and consequently how the scheduling decisions are taken.

2.3.1.2 Tasks and run-queues in the Linux kernel

Before describing the Linux scheduling framework, let us devote some attention to the two fundamental, in the context of the scheduling, data structures of the Linux kernel: `struct task_struct` and `struct rq`. A Linux *process* is an instance of a program in execution [BC05]. From the Linux kernel point of view, a process is an instance of the `struct task_struct` structure (see Listing 2.2). This structure encompasses the run-state of process, `state`, process

```
/////////////////////////////////////////////////
//Initial phase
//variable declaration
DEFINE_PER_CPU(struct hrtimer, my_hrtimer);

void my_hrtimer_init()
{
  int cpu = smp_processor_id();
  struct hrtimer *timer = &per_cpu(my_hrtimer, cpu);
  hrtimer_init(timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
  timer->function = my_hrtimer_callback;
}

/////////////////////////////////////////////////
//Second phase
void my_hrtimer_start(ktime_t expires)
{
  int cpu = smp_processor_id();
  struct hrtimer *timer = &per_cpu(my_hrtimer, cpu);
  hrtimer_start(timer, expires, HRTIMER_MODE_ABS);
}

/////////////////////////////////////////////////
//Third phase
static enum hrtimer_restart my_hrtimer_callback(struct hrtimer *timer)
{
  ...
  if(cond){
    hrtimer_set_expires(timer,next_expire);
    return HRTIMER_RESTART;
  }
  return HRTIMER_NORESTART;
}
```

Listing 2.1: The required steps to implement a high resolution timer in the Linux kernel.

priority, `prio`, scheduling class, `sched_class`, and scheduling policy, `policy`, among other fields. Note that, in the context of this dissertation, the meaning of a process or a task is the same.

```
struct task_struct {
  volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
  ...
  int prio;
  ...
  const struct sched_class *sched_class;
  ...
  unsigned int policy;
  ...
};
```

Listing 2.2: The `struct task_struct` data structure.

At boot time, the Linux kernel creates a ready-queue (or run-queue) for each available processor. Each processor ready-queue is an instance of the `struct rq` data structure (see Listing 2.3). Some fields of that structure are for management purposes, while others are for storing the ready tasks currently assigned to the respective processor. The ready tasks are stored in sub-ready-queues according to the scheduling class, `cfs` and `rt`, of the tasks (these fields are discussed later in the next subsection). Such management information is about the number of ready tasks, `nr_running`, a pointer to the currently running task, `curr`, as well as a pointer to the idle task, `idle`,

```
struct rq {
  spinlock_t lock;
  ...
  unsigned long nr_running;
  ...
  struct cfs_rq cfs;
  struct rt_rq rt;
  ...
  struct task_struct *curr, *idle;
  ...
};
```

Listing 2.3: The `struct rq` data structure.

and a locking variable (called `lock` of the type `spinlock_t`; as it name implies if the lock is not immediately acquired, it spins until acquiring the lock) that can be used to lock the ready-queue for updating its state (for instance, when a task is enqueued or dequeued from the respective ready-queue). This ready-queue lock variable plays an important role in the scheduling framework. It is used to serialize the access to the ready-queue of each processor. Then, whenever this lock is acquired by one processor only that processor is allowed to manage that ready-queue. In other words, only the processor holding that lock can change the state of that ready-queue. In a multi-processor system, whenever a task migrates from one processor to another processor, the source processor needs to lock both ready-queues, in order to atomically dequeue from the ready-queue of the source processor and enqueue in the ready-queue of the target processor. When multiple ready-queue locks are required, it is important to always obtain those ready-queue locks in the same order to avoid deadlocks. In the source code of the Linux kernel, the following comment can be found: "*Locking rule: those places that want to lock multiple runqueues (such as the load balancing or the thread migration code), lock acquire operations must be ordered by ascending runqueue*". Therefore, Linux requires that ready-queue locks are always acquired in order of increasing memory addresses. Then, whenever a processor holds a ready-queue lock it may only attempt to lock a ready-queue with higher address. But, when a processor holds a ready-queue lock that needs to acquire a second, lower address ready-queue lock, it first must release the lock that it already holds and then acquire both locks in ascending order. Due to this rule, the state of the ready-queues may change in between lock acquisitions. Considerable complexity in the Linux scheduler is devoted to detecting and reacting to such concurrent state changes.

2.3.1.3 Linux modular scheduling framework

The introduction of scheduling classes (in the Linux kernel version 2.6.23) made the core scheduler quite extensible. The kernel scheduling framework consists of a scheduler (implemented by means of the `schedule` function) and various scheduling classes (see Figure 2.7).

The scheduler (or dispatcher) is the part of the kernel responsible, at run-time, for allocating processors to tasks for execution and the scheduling classes are responsible for selecting those tasks. The scheduling classes encapsulate scheduling policies. These scheduling classes are hierarchically organized by priority and the scheduler inquires each scheduling class in a decreasing priority order for a ready task. Currently, Linux has three main scheduling classes: Real-Time (RT), Completely Fair Scheduling (CFS) and Idle. In this system, the scheduler first inquires the RT scheduling class for a ready task. If RT scheduling class does not have any ready task, then it inquires the CFS scheduling class. If CFS does not have any ready task, then it reaches the Idle scheduling class, used for the idle task: every processor has an idle task in its ready-queue that is executed whenever there is no other ready task.

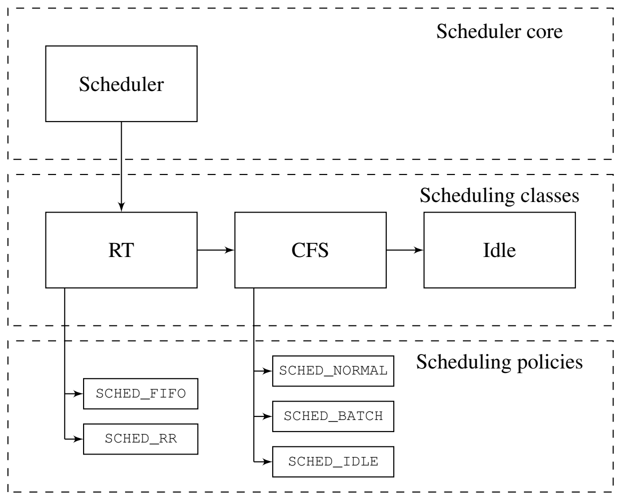


Figure 2.7: Scheduling framework architecture.

Any scheduling class is implemented as an instance of the `struct sched_class` data structure, and according to the rules of the Linux modular scheduling framework, each scheduling class must implement a set of functions specified by this data structure. Listing 2.4 shows the most important contents of the `rt_sched_class`, which implements the RT scheduling class (described in the next subsection), in the context of this dissertation. The first field of the `rt_-sched_class` structure, `next`, is a pointer to the next `sched_class` in the hierarchy. In that case, it stores the address of the `fair_sched_class` variable that implements the CFS scheduling class. The other fields are hooks to functions for handling specific events.

Let us first explain how a task is associated to one scheduling class, and then continue with the description of `sched_class` data structure. While it is possible to change the scheduling class of a task during its life cycle, at any time instant it belongs to only one scheduling class. A task is associated to one scheduling class through the `sched_class` pointer, a `struct task_-struct` field (see Listing 2.2). This pointer contains the address of the corresponding `sched_-class` variable that it is currently associated with. Thus, whenever a task becomes ready to be executed, the respective function pointed to by `sched_class->enqueue_task` is invoked. As a default behaviour, it inserts the newly ready task into the respective processor ready-queue. Then, the respective function associated with `sched_class->check_preempt_curr` is invoked for checking if the currently executing task must be preempted (according to the scheduling policy of the newly ready task), and if it is the case, then, the latter is marked to be preempted. This operation (marking the currently executing task to be preempted) triggers the invocation of the scheduler.

Whenever a task is no longer ready, the function pointed to by `sched_class->dequeue_task` is invoked. As a default behaviour, it removes the task from the respective processor ready-queue.

The scheduler can be periodically invoked at every expiration of the timer tick through the execution of the respective timer callback function, which invokes the `scheduler_tick` function. In turn, the `scheduler_tick` function invokes the `sched_class->task_tick` function of the scheduling class of the currently executing task. This gives the opportunity to implement time-driven scheduling policies.

```
static const struct sched_class rt_sched_class = {
         .next                    = &fair_sched_class,
         .enqueue_task            = enqueue_task_rt,
         .dequeue_task            = dequeue_task_rt,
         ...
         .check_preempt_curr      = check_preempt_curr_rt,
         .pick_next_task          = pick_next_task_rt
         ...
         .pre_schedule            = pre_schedule_rt,
         .post_schedule           = post_schedule_rt,
         ...
         .task_tick               = task_tick_rt,
         ...
  };
```

Listing 2.4: The `rt_sched_class` definition.

The scheduler is implemented through the `schedule` function. This function is, basically, executed whenever the currently executing task is marked to be preempted or whenever a task completes or voluntarily yields the processor (such as when a task waits for signal to occur or wants to sleep). For instance, the completion of a task is done by invoking the `exit` system call (even if this is not explicit in the programmer code). The `exit` system call performs a set of cleanup operations for freeing the task resources and then it invokes the `schedule` function. Listing 2.5 presents a snippet code of the `schedule` function. The first step is therein to disable preemptions. Next, it retrieves the ready-queue of the current processor (the processor executing this code). For that purpose, it obtains, via `smp_processor_id` macro, the identifier of the current processor, and then, via the `cpu_rq` macro, it retrieves the ready-queue, `rq`, of that processor. The currently executing task is pointed to by the `rq->curr` pointer. In the context of the code of the schedule function, whose purpose it to get the next task to be executed, the *current* task is then marked as the *previous* executing task, via the `prev = rq->curr` assignment.

The next step is to lock the current processor ready-queue. After these steps, the `pre_-schedule` function is executed, which invokes the `sched_class->pre_schedule` function of the current task pointed to by `prev`. This is the hook for the scheduling class to execute any logic before the scheduler calls `pick_next_task`. Currently, only the RT scheduling class implements this function (see Subsection 2.3.1.4 below). The next task selected for executing is retrieved by the `pick_next_task` function. This function iterates over the linked list of the scheduling classes, starting by the highest-priority scheduling class. For each scheduling class, it invokes the function associated to the `pick_next_task` hook of each scheduling class and the iteration is broken whenever that function returns a non-null pointer. Note that, in the last case the idle task is selected. Next, the scheduler can switch the previous task (pointed to by `prev`) with the next task (pointed to by `next`), by invoking the `context_switch` function. Note that, this function unlocks the ready-queue. When the switching of tasks is done, the scheduler calls the `post_schedule` function that invokes the function pointed to by `sched_class->post_-schedule` of the next task (now pointed to by `rq->curr`). This function locks the ready-queue. Again, only the RT scheduling class (described next) implements this function and it is a hook that allows the RT scheduling class to push real-time tasks of the current processor. As a final step, the preemption is again enabled.

```
static void __sched __schedule(void)
 {
        ...
         preempt_disable();
         cpu = smp_processor_id();
         rq = cpu_rq(cpu);
         prev = rq->curr;
         ...
         raw_spin_lock_irq(&rq->lock);
         ...
         pre_schedule(rq, prev);
         ...
         next = pick_next_task(rq);
         ...
         rq->curr = next;
         ...
         context_switch(rq, prev, next); /* unlocks the rq */
         ...
         post_schedule(rq);
         ...
         preempt_enable_no_resched();
         ...
 }
```

Listing 2.5: The `schedule` function.

2.3.1.4 The real-time scheduling class of Linux

As mentioned before, each processor holds a ready-queue of all ready tasks currently assigned to it. Actually, the ready-queue of each processor has one *sub-ready-queue* for each scheduling class, as depicted by Listing 2.3. The sub-ready-queue of the CFS scheduling class is an instance of the `struct cfs_rq` data structure while the sub-ready-queue of the RT scheduling class is an instance of `struct rt_rq` data structure. This allows each scheduling class to organize per-processor their ready tasks in an appropriate manner. As depicted in Figure 2.8, tasks in the RT scheduling class are organized by priority level. Inside of each priority level, the RT scheduling class implements two scheduling policies: `SCHED_FIFO` and `SCHED_RR`. `SCHED_FIFO` is based on the First-In-First-Out (FIFO) heuristic: whenever a `SCHED_FIFO` task is executing, it continues until preempted (by a higher priority task) or blocked (e.g., by an I/O operation). `SCHED_RR` implements the Round-Robin (RR) heuristic: a `SCHED_RR` task executes (if it is not preempted or blocked) until it exhausts its time-slice. At the same level of priority, the RT scheduling class favours `SCHED_FIFO` tasks over `SCHED_RR` tasks.

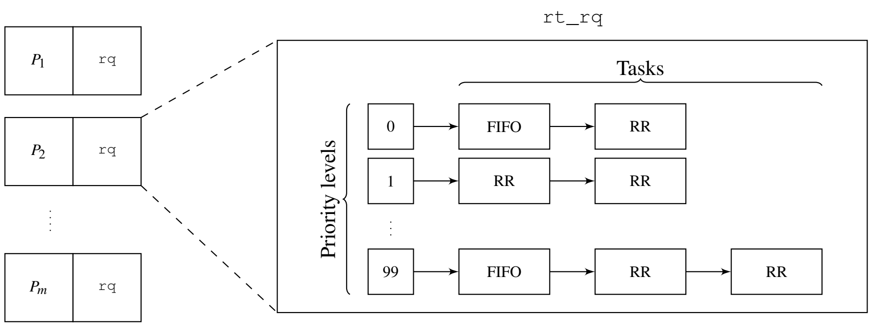


Figure 2.8: RT scheduling class ready-queue.

Each processor, at any time instant, is executing one task stored in its ready-queue. This may result in unbalanced workloads across processors. In order to balance the workload across processors, the RT scheduling class adopts an active push/pull strategy as follows: whenever the scheduler inquires the RT scheduling class (through the `pick_next_task` function), it first tries to pull the highest-priority non-executing task from the other processor ready-queue using the `pre_schedule` function and, after selecting the next running task, it checks if it can, using the `post_schedule` function, push the (freshly) preempted task to another processor ready-queue that is executing a task with lower priority than the preempted task. Observe that, moving tasks between two ready-queues requires locking *both* ready-queues and this might introduce considerable overheads.

2.3.2 The Linux PREEMPT-RT patch

There exist several important sources of unpredictability in the mainline Linux kernel, namely: (i) interrupts are generated by the hardware often in a unpredictable manner and when an interrupt arrives, the processor execution switches to handle it; (ii) multiple kernel threads running on different processors in parallel can simultaneously operate on shared kernel data structures, requiring serialization of the access to such data; (iii) disabling and enabling preemption features used in many parts of the kernel code can postpone some scheduling decisions.

The unpredictability in the mainline Linux kernel can cause arbitrary delays to the real-time tasks running on the system. To address these problems, a kernel developer community [Mol04a, McK05, RH07] has been working on the PREEMPT-RT patch [PR12]. This patch (that aims to derive a fully preemptible kernel) adds some real-time capabilities to the Linux kernel.

The following subsections illustrate the different sources of unpredictability, and how the PREEMPT-RT patch deals with: priority inversion; large non-preemptive sections; locking mechanisms; and interrupt management.

2.3.2.1 Priority inversion and Priority inheritance

Let us illustrate the classical example of *priority inversion* (see inset (a) of Figure 2.9). Consider a task set composed by three tasks: τ_1, τ_2, and τ_3. The priority of each task is according to their indexes. Lower index implies higher priority, thus, τ_1 is the highest-priority task while τ_3 is the lowest-priority task. Tasks τ_1 and τ_3 share some resource (graphically illustrated by a black rectangle). Task τ_3 becomes ready at time instant t_0, and immediately it starts executing (because it is the only ready task). At time t_1, task τ_3 locks the shared resource. At time t_2, task τ_1 becomes ready, preempts task τ_3, and executes until t_4, when it is blocked on due to the shared resource that is held by the lowest-priority task τ_3. Meanwhile, at time t_3, task τ_2 becomes ready. Since task τ_1 is blocked, the current highest-priority task is task τ_2. Thus, task τ_2 executes until its completion (time t_5). At time t_5, the only task that is ready to be executed is task τ_3; then it executes until freeing the resource, which occurs at time t_6. At this time, task τ_1 becomes ready and preempts task τ_3. Task τ_1 executes until its completion that occurs at time t_7, and task τ_3 restarts its execution until its completion at time t_8. As can be seen from inset (a) of Figure 2.9, the highest-priority task τ_1 is prevented from being executed by the lowest-priority task τ_3. This is could be harmful if the execution time of task τ_2 is unbounded, thus indefinitely extending this blocking time. This is what is known as *unbounded priority inversion*.

One approach to avoiding the priority inversion is the technique of *priority inheritance*. The PREEMPT-RT patch implements Priority Inheritance (PI). The underlying idea of PI is: "*a task that holds the lock on shared resource inherits the maximum priority of all tasks waiting for that resource, but only during its execution with the shared resource*". Afterwards, it returns to the original priority level. Thus, the highest-priority task can only be blocked by a lower priority task only during the intervals that the latter is executing holding the shared resource. Inset (b) of Figure 2.9 illustrates that effect. Independently of whether the execution time of task τ_2 is known or not, the highest-priority task τ_1 is only blocked during the execution with the shared resource of task τ_3. Therefore, the blocking time of the highest-priority task is bounded with PI.

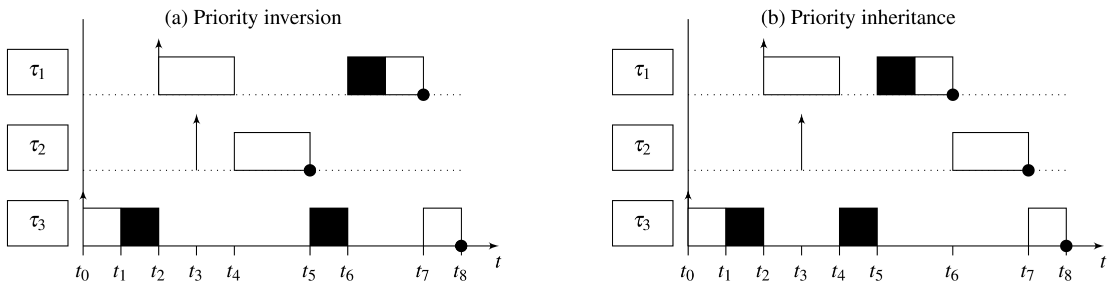


Figure 2.9: Illustration of the priority inversion and priority inheritance.

2.3.2.2 Fully preemptible kernel

The PREEMPT-RT patch reduces the Linux kernel latencies by reducing its *non-preemptible sections*. A non-preemptible section is a series of operations that must be performed without any preemption. One of the most intrusive mechanism is the Big Kernel Lock (BKL). The BKL is a system-wide locking mechanism used to protect concurrent accesses of non-preemptible sections from tasks running on separate processors. The BKL was implemented by a spin lock. Further, it was also recursive, which means that the holder of the BKL was allowed to re-acquire it again. Last and worst, the BKL holder could sleep. As expected, the BKL does not have many fans, according to [Mau08]: "*Although the BKL is still present at more than 1,000 points in the kernel, it is an obsolete concept whose use is deprecated by kernel developers because it is a catastrophe in terms of performance and scalability.*"

The implementation of the BKL [Mol04b] was modified by replacing the spin lock with a mutex (more details in the next subsection), which allows the preemption of the BKL holder. Kernel preemptibility is also achieved by replacing most kernel spin locks by mutexes, which support PI, and by transforming all interrupt handlers into preemptive kernel tasks, scheduled by the RT scheduling class.

2.3.2.3 Locking primitives

The Linux kernel employs locking primitives to protect critical sections. A *critical section* is a code path that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one task. In other words, it must be ensured that such a code path executes *atomically*; that is, operations complete without interruption as if the entire critical section were one indivisible instruction. Otherwise, if more than one task accesses a critical section at a given time this is generally referred to as *race condition*, because such tasks race to get there first. As it is intuitive, race conditions could be fatal in the literal sense of the word.

On uniprocessor systems race conditions can be protected by disabling preemptions during the execution of critical sections, then, since tasks may execute until the end of such critical sections without being preempted, at given time only one task may access to each shared resource. On multiprocessor systems, disabling preemptions is not enough to protect a shared resource, because enabling and disabling preemptions is performed on a per-processor basis. In other words, suppose that two tasks are executing on two different processors: disabling preemptions on each respective processor does not prevent such tasks from accessing the same shared resource. Consequently, a system-wide locking mechanism is required to prevent such an event. The Linux kernel provides some locking primitives that can be used to avoid race conditions, such spin locks, semaphores and mutexes.

The operation mode of a *spin lock* is as follows: whenever a task attempts to enter into a critical section protected by a spin lock, it immediately acquires the lock, if it is free, otherwise (as the name implies) it spins until the lock is free. Then, a waiting task performs a *busy waiting* by means of a loop for acquiring the lock. As it is intuitive, this approach should be only employed whenever the expected waiting time is small. Spin locks are considered fast compared to sleeping locks, because they avoid the context switch overheads (from switching the processor from one task to another). However, the use of spin locks to protect large sections (as unfortunately happens within the Linux kernel) and, especially, the use of nested spin locks can cause large and, potentially, unpredictable latencies.

The PREEMPT-RT patch converts most spin locks into sleeping locks denoted as *mutex*es a name which comes from *mut*ually *ex*clusive). Before describing mutexes, let us discussing the more general locking mechanism denoted by the term *semaphore*. A semaphore is a locking mechanism that can control the concurrent access to a shared resource. Semaphores can be either binary or counting. In the former case, they protect a shared resource from multiple accesses; that is, only one task can use such a resource. In the latter case, multiple accesses are allowed; that is, many tasks can use the shared resource at a given time. Unlike what holds for spin locks, a task holding a semaphore (binary or counting) may sleep. Semaphores, with the property of allowing more than one task into a critical section, have no concept of an owner. Indeed, one task may even release a semaphore acquired by other another task. A semaphore is never linked or associated to a task.

A mutex, on the other hand, always has a one-to-one relationship with the holder task. A mutex may be owned by one, and only one, task at a time. This is a key to the PREEMPT-RT patch, as it is one of the requirements of the PI algorithm; that is, a lock may have one, and only one, owner. This ensures that a priority inheritance chain follows a single path, and does not branch with multiple lock owners needing to inherit the priority of a waiting task.

2.3.2.4 Threaded ISRs

In the mainline Linux kernel, the interrupts are the event with the highest-priority. Whenever an interrupt fires, the processor stops the execution of the current task to execute the ISR associated with that interrupt. Suppose that the currently executing task is the highest-priority task in the system and a low-priority task has requested hard disk I/O operations. What happens in this case is that the execution of the highest-priority task will be greatly affected by the low-priority task. The hard disk I/O operations are, usually, managed under Direct Memory Access (DMA), which allows data to be sent directly from the hard disk to the main memory out of the processor control; however, the processor is notified whenever an operation ends. This kind of event comes in a unpredictable manner, therefore, it may unpredictably prevent the execution of the highest-priority task as well as of all other tasks.

PREEMPT-RT creates a kernel thread (or task) for each interrupt line and whenever some device fires a request, the respective kernel task is woken up; that is, it becomes ready to be executed. Thus, it is scheduled according to the RT scheduling policies. It has a priority level (50 by default) and therefore it causes the preemption of the lower-priority tasks and is preempted by higher-priority tasks. Despite of the benefits of this approach, there are some cases where an ISR is not converted into a kernel task. For that purpose, those ISRs have to be flagged as `IRQ_NODELAY`.

2.3.2.5 Scheduling policies

The RT scheduling class, as implemented in the PREEMPT-RT patch, supports the same scheduling policies as the mainline Linux kernel: `SCHED_FIFO` and `SCHED_RR` scheduling policies. While these scheduling policies are appropriate for uniprocessors, they are not adequate for multiprocessor systems because (i) their performance is poor on multiprocessors (since there exist task sets were a system with a load a little over 50% fail to meet deadlines) and (ii) they adopt an active push/pull approach for balancing tasks across processors. Such push/pull operations require locking multiple processor ready-queues, which cause an additional source of unpredictability.

The scheduling algorithm suitable for multiprocessor systems must be supported by the RT scheduling class, but the PREEMPT-RT patch does not add any scheduling algorithms suitable for multiprocessor systems. This is an important omission in the PREEMPT-RT patch, which is addressed in this work.

2.3.3 Real-time scheduling add-ons

During the last years some works, mostly, from the academia have provided the Linux kernel (some directly in the mainline and others using PREEMPT-RT-patched Linux kernel) with support for some algorithms suitable for real-time systems.

The LITMUS$^{\text{RT}}$ framework [LR12], whose name stands for LInux Testbed for MUltiprocessor Scheduling in Real-Time systems, focuses on real-time scheduling algorithms and synchronization protocols for multiprocessor systems and its primary goal is to provide a useful experimental platform for applied real-time systems research. To this end, the mainline Linux kernel is modified and distributed as a patch file, which requires the Linux kernel to be compiled with it.

The first prototype of LITMUS$^{\text{RT}}$ was developed back in 2006 based on the Linux 2.6.9 kernel version [CLB$^+$06], and since that time many releases have been produced [BBC$^+$07, BCA08, BA09c, Bra11]. The current version is based on the mainline Linux 3.0.0 kernel version. The project was initially focused on partitioned and global scheduling algorithms. More recently, it also incorporated clustered scheduling algorithms. The current LITMUS$^{\text{RT}}$ version also supports semi-partitioned scheduling algorithms [BBA11].

The implementation approach followed for LITMUS$^{\text{RT}}$ is based on scheduler plugins that implement the respective scheduling algorithms. Architecturally, LITMUS$^{\text{RT}}$ is based on a generic scheduling class (in Section 2.3.1.3 the Linux modular scheduling framework is described) called `litmus_sched_class` and positioned on the top of the Linux hierarchy of scheduling classes. This scheduling class does not implement any particular scheduling algorithm. It is a thin wrapper that delegates all scheduling decisions to the currently active scheduler plugin. The idea behind the scheduling plugins is to have the possibility of switching the scheduling algorithm at run-time.

The REal-time SCHeduler (RESCH) is a loadable suite for Linux [KY08a, KRI09, Kat12]. The primary objective of RESCH is to facilitate academic research on real-time systems for practical use, and to make them available in the real world. From the implementation point-of-view, it uses a different approach as it does not require any kernel patch. Consequently, there is no need to recompile the Linux Kernel. The RESCH core is implemented as a character device driver. Note that, developing a device driver module is similar to developing any user-space application. In order to activate RESCH, it is only required to install the respective device driver module, which is also similar to the installation of a user-space application.

As with the LITMUS$^{\text{RT}}$ framework, RESCH also supports a variety of scheduling algorithms: partitioned, global and semi-partitioned. Initially, such scheduling algorithms were implemented using the native real-time scheduling class of Linux using the POSIX-compilant `SCHED_FIFO` scheduling policy (described in Section 2.3.1.4). Nowadays, it is instead designed and implemented using SCHED_DEADLINE [SCH12].

Originally, SCHED_DEADLINE [FTCS09] only supported partitioned-EDF. Nowadays, global- and clustered-EDF [LLFC11] are also supported. The SCHED_DEADLINE implementation is compliant with POSIX interfaces and follows the rules defined by the Linux modular scheduling framework (described in Section 2.3.1.3). SCHED_DEADLINE is implemented as a scheduling class (called `dl_sched_class`) that is positioned on the top of the Linux scheduling class hierarchy, thus, becomes the highest-priority scheduling class.

Contrarily to LITMUS^{RT}, SCHED_DEADLINE does not use a shared global ready-queue to implement the global-EDF scheduler. Actually, in terms of implementation, there is no difference among the different categories of supported scheduling algorithms by SCHED_DEADLINE. It uses a ready-queue per-processor (as the Linux native real-time scheduling class) that is suitable for partitioning. Note that, under this approach, each processor can only execute tasks stored into its own ready-queue. Therefore, to support global and cluster scheduling, tasks are allowed to migrate among processors. This migration mechanism is, essentially, supported by *pull/push* operations (also used by the real-time scheduling class of mainline Linux; Section 2.3.1.4 describes such operations).

2.3.4 Micro kernel-based approaches

A micro kernel-based approach is used in some Linux-based RTOSs [Yod99, RTA12, Xen12] to support the real-time system requirements. Under this approach, a software layer called micro kernel is created and placed between the Linux kernel and the hardware (see Figure 2.10). Thus, the micro kernel interfaces the hardware with the rest of the software. Therefore, it is in a privileged position to control the entire system if it is provided with the required mechanisms for handling interrupts and scheduling tasks, for example.

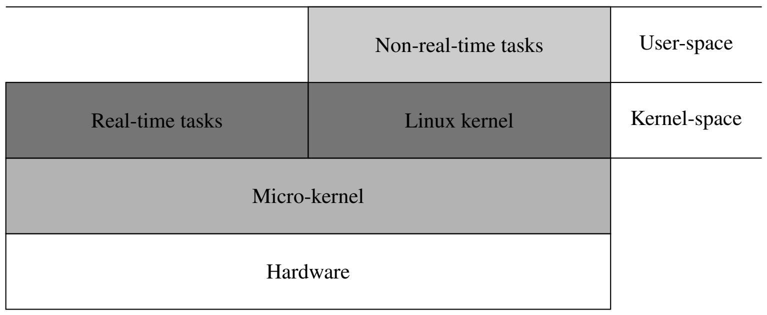


Figure 2.10: Computer system architecture with the micro kernel layer.

The first implementation of such an approach was RTLinux [Yod99], which stands for Real-Time Linux. RTLinux handles the interrupts from peripherals and dispatches the interrupts to

Linux kernel when there are no ready real-time tasks. Whenever the Linux kernel disables interrupts, RTLinux queues the interrupts to be delivered after the Linux kernel has enabled the interrupts once again.

The RTLinux implements real-time tasks as kernel modules: kernel modules are pieces of code, which may be loaded and unloaded into the kernel and do not require a kernel compilation. Hence, such tasks execute in kernel-space like the Linux kernel. In order to guarantee the priority of the real-time tasks, RTLinux considers the Linux kernel as the lowest-priority task in the system, which may do its normal operations whenever there are no real-time tasks running.

For scheduling real-time tasks, RTLinux implements its own scheduler as a module, fully preemptive and based on a fixed-priority scheme. However, given the modular nature of the scheduler, other scheduling policies can be easily added. The mainline Linux scheduler is not involved and given overall system control by RTLinux, Linux cannot delay or interrupt (or preempt) the execution of any real-time task.

Implementing real-time tasks as kernel modules presents two drawbacks though: first, it is very difficult to debug such tasks, and second, when executing in kernel-space, a software bug in a real-time task could be catastrophic for the entire system.

These drawbacks are addressed by a more recent RTOS, called RTAI (that stands for Real-Time Application Interface)[RTA12]. RTAI uses the same approach as RTLinux, since it also considers the Linux kernel as the lowest-priority task in the system, but it provides a complete Application Programming Interface (API) for developing real-time tasks, called LXRT (which stands for LinuX-RT). These tasks execute in user-space context. In essence, for each real-time task, LXRT creates a real-time agent that executes in kernel-space context. Whenever, a real-time task starts executing, the real-time agent disables interrupts and preemptions, removes the task from the Linux ready-queue and adds it to the RTAI scheduler queue. This way, such a task is protected from the other Linux processes.

More recently appeared ADEOS (which stands for Adaptive Domain Environment for Operating Systems). ADEOS consists of a resource virtualization layer placed on the top of the hardware and below the kernel layer [Ade13]. It intends to provide the possibility of sharing the hardware among several operating systems, referred to as domains (see Figure 2.11), which are ordered by priority. Each domain can request to ADEOS to be notified of some important events, such as interrupts, system call invocations, task-switching and task completions, just to mention a few. An event pipeline is formed according to the domain's priority. Incoming events are pushed into the head of the pipeline and flow down to its tail. This way, the events are handled first by the highest-priority domain and then by the next one and so on until reach the lowest-priority domain. The event propagation is constrained to some rules. For instance, a domain can forward or stop an event. For critical operations, a domain can stall events in its own pipeline stage; this is equivalent to disabling events for the subsequent domains. The events can be *unstalled* later, and then they continue the flow to the subsequent domains.

Xenomai [Xen12] is a more recent implementation for supporting real-time system requirements. Xenomai makes use of ADEOS for its own purposes. As illustrated in Figure 2.11, a

Xenomai system is composed by three domains: Xenomai, the interrupt shield, and the Linux kernel. In the Xenomai's jargon, the Xenomai domain (the highest-priority domain) is said to to be the primary execution mode, while the Linux kernel domain (the lowest-priority domain) undergoes the secondary execution mode of real-time tasks. The interrupt shield domain, as its name implies, provides a mechanism for managing the interrupt flow across domains.

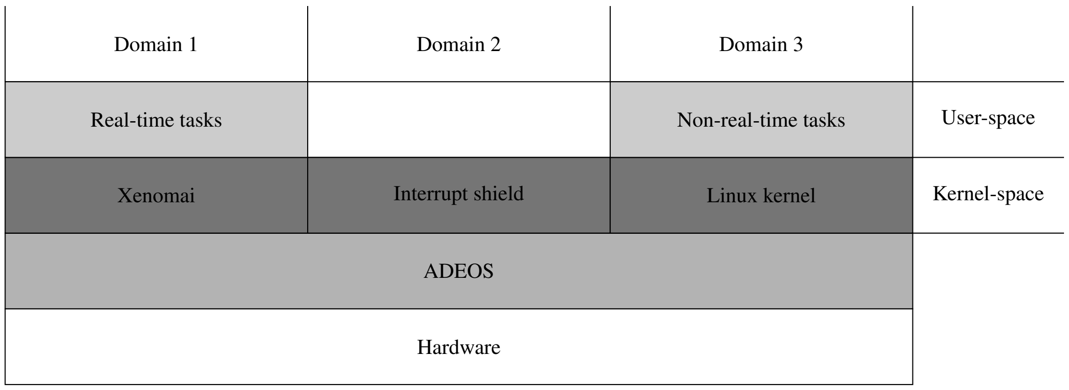


Figure 2.11: Xenomai computer system architecture.

Real-time tasks can be executed in the primary context mode as well as in the secondary execution mode; that is, a real-time task starts its execution on Xenomai, but it can jump to the Linux kernel whenever it invokes a Linux system call and can then return once again to Xenomai. Whenever this happens, the real-time task is removed from Xenomai's ready-queue and is inserted into the Linux ready-queue, more specifically into the queue for the RT scheduling class. As mentioned before, the RT scheduling class organizes its tasks according to their priority level and, for each level, it uses two scheduling policies: `SCHED_FIFO` and `SCHED_RR`. In this case, such tasks are scheduled according to the `SCHED_FIFO` scheduling policy. However, this implies the use of a common priority scheme by the Xenomai and the Linux kernel. Hence, to guarantee the priority compatibility, the real-time tasks have a priority level in the same order of magnitude as the Linux RT scheduling class (from zero to 99).

Xenomai is provided with mechanisms for guaranteeing the predictability of the execution of real-time tasks. However, whenever a real-time task jumps to the Linux kernel, it is out of its control, therefore Xenomai can no longer guarantee the predictability of that execution. One type of events that contributes to such unpredictability is interrupts. When a real-time task is being executed on the Linux kernel, it should not be perturbed by non-real-time Linux interrupt activities. To isolate those tasks from such events, Xenomai employs a mechanism that starves the Linux kernel from interrupts when a real-time task is being executed on it. For that purpose, Xenomai instructs the intermediate domain, the interrupt shield domain, to stall such interrupts during the time that such tasks are being executed on Linux kernel.

Another source of unpredictability are the long non-preemptible (and time unbounded) sections. Long non-preemptible sections could considerably delay scheduling decisions. Whenever a real-time task jumps to the Linux kernel, it is only executed after a scheduling decision has been taken. Therefore, the responsiveness of a system increases if such scheduling decisions are frequent, otherwise the responsiveness decreases. Hence, the probability of such a task experiencing long delays depends on the existence of long non-preemptible sections in the system. For this reason, Xenomai keeps an eye on the developments provided by the PREEMPT-RT patch [PR12], described earlier in this chapter, which aims for a fully preemptible Linux kernel.

One last source of unpredictability, stems from the priority inversion phenomena. Xenomai provides mechanisms for handling such phenomena, but the Linux kernel does not provide any mechanism. Once more, Xenomai uses the development provided by PREEMPT-RT patch, which implements the PI resource sharing protocol to reduce priority inversion.

2.4 Summary

In this chapter, we have addressed the most important platform-related issues of multiprocessor systems. We first discussed some hardware issues of the multiprocessor systems, namely, those that contribute to unpredictability, such as memory caches and interrupts. We then addressed software-related aspects, and described the most important features of the mainline Linux kernel. This research aims at putting real-time slot-based task-splitting scheduling algorithms into practice, and, in this chapter, we described several enablers of our experimental work, such as the Linux kernel features that support the implementation of a real-time scheduler and the PREEMPT-RT patch. We note that the PREEMPT-RT patch does not yet proper support real-time multiprocessor scheduling policies, something addressed by this work. We have also reviewed several other approaches introducing real-time capabilities into Linux-based operating systems, namely those that use a co-kernel to guarantee the real-time system requirements, and others that incorporate real-time features into the Linux kernel.

Chapter 3

Background on real-time systems

3.1 Introduction

Initially, real-time systems were restricted to some very specific domains as military, avionics, and industrial plants, just to mention a few. Nowadays, real-time systems are becoming increasingly important and pervasive, as more and more industries, infrastructures, and even ordinary people depend on them. Indeed, the increase of real-time domains as well as the increase of real-time systems have been in an order of magnitude that human beings use them daily and often without knowing. Naturally, with the general proliferation of multi-core platforms, it is expected that real-time applications start to be massively deployed on such platforms. A key factor for that, among other reasons, is the considerable boost in processing capacity in a relatively cheap, small, and low power consuming chip. Therefore, they offer an opportunity to maximise performance and execute more complex and computing-intensive tasks whose stringent timing constraints cannot be guaranteed on single-core systems.

In this chapter, we aim to provide the required background in real-time concepts for multiprocessor systems. We start by presenting and discussing the real-time concepts in general and then we focus on those specific to multiprocessor systems. Next, we review some relevant real-time scheduling algorithms for multiprocessor systems. After that, using a task set example, we go through the details of several real-time scheduling algorithms for multiprocessor systems. We finish this chapter with a historical perspective of the scheduling algorithms under consideration in this work.

3.2 Basic concepts

In this section, we discuss the real-time concepts used throughout this dissertation. First, we describe the system model and then we address the fundamental concepts of real-time scheduling.

3.2.1 System model

Consider a system composed by m processors. Each processor is uniquely indexed in the range P_1 to P_m and $U[P_p]$ denotes the utilization of processor P_p; that is, the current workload of processor P_p.

A real-time application is, typically, composed of a static set, τ, of *n tasks* ($\tau = \{\tau_1, \cdots, \tau_n\}$). Each task, τ_i, generates a sequence of *z jobs* ($\tau_{i,1}, \cdots, \tau_{i,z}$, where z is a non-negative number and potentially $z \to \infty$), and is characterized by a three-tuple (C_i, T_i, D_i).

The *worst-case execution time*, C_i, is the maximum time required by the processor to execute a job of task τ_i without any interruption. T_i defines the frequency at which jobs of task τ_i are released in the system and, according to the nature of T_i, the systems are classified in three broad categories: (i) *periodic*, jobs are released regularly at some known rate (called *period*); (ii) *sporadic*, jobs are released irregularly at some known rate (called *minimal inter-arrival time*); and finally, (iii) *aperiodic* jobs appear with irregular arrival times, typically, at unknown rate. Aperiodic tasks are out of the scope of this dissertation (the reader is referred to [AB98] for more details how to handle aperiodic tasks). The temporal constraint of each task, τ_i, is defined by its *relative deadline*, D_i, the size of the time window for executing a job of such task τ_i.

A task set, τ, is said to have *implicit deadlines* if the relative deadline of every task is equal to its period ($D_i = T_i$), *constrained deadlines* if the relative deadline of every task is less than or equal to its period ($D_i \leq T_i$), and *arbitrary deadlines* if there is no such constraint; that is, D_i can be less than, equal to, or greater than T_i.

For implicit deadline task set, the *utilization of task* τ_i, u_i, is given by the ratio between its worst-case execution and its period/minimal inter-arrival time:

$$u_i = \frac{C_i}{T_i} \tag{3.1}$$

The *utilization of the system*, U_s, is the sum of the utilization of all tasks normalised by the number of processors (m):

$$U_s = \frac{1}{m} \sum_{i=1}^{n} u_i \tag{3.2}$$

A task's *density* is an adaptation of its u_i for constrained or arbitrary deadline task sets and is computed as:

$$\lambda_i = \frac{C_i}{\min(D_i, T_i)} \tag{3.3}$$

A task set, τ, is said to be *synchronous* if the first jobs of all tasks are all released at the same time (usually this time instant is assumed to be time zero), otherwise it is said to be *asynchronous* and, in this case, it is necessary to define an *offset*, O_i, of the first job arrival of each task τ_i related to time zero. Typically, time zero is the time instant when the real-time system starts operating.

Each job $\tau_{i,j}$ (this notation means the j^{th} job of task τ_i) becomes *ready* to be executed at its arrival time, $a_{i,j}$, and continues until its *finishing* (or completion) time, $f_{i,j}$. The *absolute deadline*,

$d_{i,j}$, of job $\tau_{i,j}$ is computed as: $d_{i,j} = r_{i,j} + D_i$. Therefore, a deadline miss occurs when $f_{i,j} > d_{i,j}$. As mentioned before, we do not consider aperiodic tasks; then, the time difference between the arrivals of two consecutive jobs by the same task τ_i must be equal to T_i (for periodic tasks) or at least equal to T_i (for sporadic tasks).

Figure 3.1 illustrates the relation among the timing parameters of the job $\tau_{i,j}$. The execution of the job $\tau_{i,j}$ is represented by a gray rectangle and the sum of all execution chunks ($c^x_{i,j}$), which correspond to the execution time of that job, cannot exceed C_i.

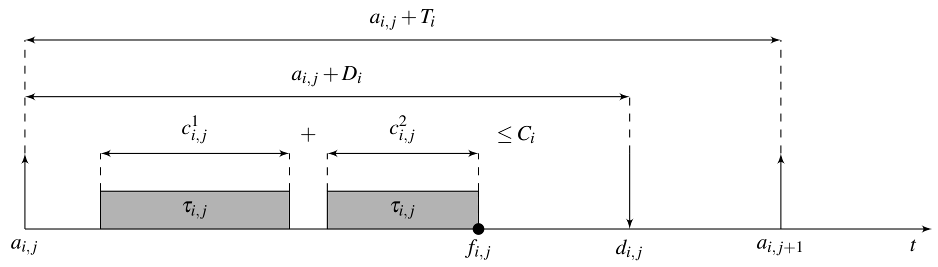


Figure 3.1: Illustration of the job timing parameters.

According to its criticality level, a real-time task can be classified as: *hard real-time* or *soft real-time*. A task is said to be a hard real-time task if it is not allowed to miss any job deadline, otherwise undesirable or fatal results will be produced in the system. On the other hand, soft real-time tasks can miss some deadlines and the system is still able to work correctly. For example, a heart pacemaker[1] is a hard real-time system because a delay on electrical impulses may kill a person while a live audio-video system is categorized as a soft real-time system because missing a deadline results in degraded quality, but the system can continue to operate.

Real-time tasks can be categorized as *dependent*, if they interact with other tasks and its execution can be blocked by those tasks, or *independent*, if they do not.

3.2.2 Fundamental concepts of scheduling

According to [But04] the preemption is: "*the operation of suspending the executing task and inserting it into the ready-queue*". Thus, in a *preemptive* system, the execution of the running task can be interrupted in order to execute a *more important* task on the processor. On the contrary, in a *non-preemptive* system, a task that has started executing on the processor cannot be interrupted before the end of its execution.

A *scheduling algorithm* is a method for defining an order/sequence for a set of ready jobs to have access to processor or processors. Real-time scheduling algorithms should schedule ready jobs according to their demands such that their deadlines are met.

A scheduling algorithm is said to be *fixed-task* priority if priorities are assigned per-task before run-time and applied to all jobs of each task and do not change during run-time. One example is the Rate Monotonic (RM) [LL73], where priorities are assigned according to the periodicity of

[1]A heart pacemaker is a medical device that uses electrical impulses to regulate the beating of the heart.

each task. A scheduling algorithm is said to be *fixed-job* priority if the priority of a task might change during run-time but each job has a fixed priority. For example, the Earliest Deadline First (EDF) [LL73] scheduling algorithm assigns the highest-priority to the job with the earliest absolute deadline. Finally, a scheduling algorithm is said to be *dynamic-job* priority if the priority of a job might change during run-time. For example, the Least-Laxity-First (LLF) [Der74] scheduling algorithm is a dynamic-job priority scheduling algorithm since it assigns higher priority to a job with the least laxity, which is, at time t, a measure of urgency for a job, however, such urgency varies during the execution of the job.

A task set is said to be *feasible* if there exists some scheduling algorithm under which it is schedulable. A scheduling algorithm is said to be *optimal* if it is able to schedule any task set that is feasible.

Multiprocessor systems introduce an additional dimension to the scheduling algorithm classification known as the *migration* degree. According to the migration degree, multiprocessor scheduling algorithms have traditionally been categorized as *global* or *partitioned*. Global scheduling algorithms store all ready tasks (or jobs) in one global queue that is shared by all processors. At any moment, the m highest-priority ready tasks among those are selected for execution on the m processors. Tasks can migrate from one processor to another during its execution; that is, the execution of a task can be preempted on one processor and resume on another processor. Some of these scheduling algorithms are said to be *work-conserving*, because it is not possible to have any processor in idle state while there is a ready task waiting for execution. As a consequence of that, this category of scheduling algorithms naturally provides a good workload balance among processors which, in overload situations, may help prevent a deadline miss because there is no constraint restricting on which processor a task can be executed.

In contrast, partitioned scheduling algorithms divide the task set such that all tasks in a partition are assigned to the same processor thus, reducing a multiprocessor scheduling problem, at least conceptually, to a set of m uniprocessor problems. Since tasks are not allowed to migrate from one processor to another, generally, these kind of scheduling algorithms are non work-conserving, because it is possible to have a processor in an idle state when there is some ready task waiting for execution on other processor.

Partitioning implies assigning tasks to processors. Assigning tasks to processors is a bin-packing problem, which is known to be a NP-hard problem [CGJ97]. The main goal of bin-packing is to pack a collection of items with different sizes into the minimum number of fixed-size bins such that the total weight, volume, etc. does not exceed some maximum value. In the context of a real-time scheduling algorithm, each item is a task, τ_i, belonging to the task set, τ, the size of each item is the utilization of that task, u_i, each bin is a processor, P_p, and the size of each bin is the capacity of a processor, usually considered to be 100%.

There are several heuristics for this kind of problems. Examples of such heuristics are: Next-Fit (NF), First-Fit (FF), Best-Fit (BF), and Worst-Fit (WF). NF assigns tasks to the current processor and if one task does not fit on the current processor it moves on and continues packing on the next processor. FF assigns each task to the first (lowest-indexed) processor that can accept that task. BF assigns a task to the lowest-indexed processor that will have the smallest remaining capacity after the assignment. WF is the inverse of the BF heuristic, that is, it assigns a task to the lowest-indexed processor that will have the greatest the remaining capacity after the assignment.

In the last years, a new class of scheduling algorithms, called *semi-partitioned* or *task-splitting*, has been proposed. Under a semi-partitioned scheduling algorithm, some tasks are assigned to specific processors, as in partitioned schemes, while other tasks may migrate between processors, as in global schemes.

There is also an orthogonal variant of the global, partitioned, and semi-partitioned categories called *clustered* scheduling. Clustered scheduling [BB07] is a hybrid of global, partitioned, and semi-partitioned [AT06, BA09b] scheduling that groups processors according to some criterion. The most common criterion is the cache level [GCB13], that is, a cluster is formed out of the cores that share the same cache level. For example, tasks may be partitioned to clusters and the tasks within each cluster may be scheduled using some global or some semi-partitioned algorithm. This avoids migrations among cores on different chips that cause cache misses, which imply extra accesses to the main memory with the consequent performance degradation. From this point of view, partitioned scheduling uses clusters composed by only one processor, while global and semi-partitioned scheduling use a single cluster containing all processors in the system. However, there is an important aspect common to all these scheduling categories: at given time instant one task can be executed by only one processor; that is, *execution parallelism is not allowed*.

All of the above defined concepts are used to characterize the task set model, scheduling algorithm, and host platform. A *schedulability test* is formal tool for checking if a task set is *schedulable* (no deadline miss occurs) using a specific scheduling algorithm running on a specific platform or not. Note that, schedulability tests play an important role in real-time systems. In the words of Bertogna [Ber08]: "*Having a good scheduler without being able to prove the correctness of the produced schedule is almost useless in the real-time field. Since designers are interested in finding efficient scheduling algorithms at least as much as they are interested in proving that no deadline will be missed, the tightness of existing schedulability tests is a key factor when selecting which class of schedulers to adopt*".

A common way to characterize the performance of scheduling algorithms is by using the notion of a *utilization bound*. The utilization bound of a real-time scheduling algorithm is a threshold for the task set utilization such that all tasks meet their deadlines when scheduled by the algorithm in consideration if the task set utilization does not exceed that threshold. This concept is commonly used to compare scheduling algorithms and it is also an indicator of the expected performance of a scheduling algorithm.

So far, we have described the generic concepts of the real-time systems, in general, and specific for multiprocessor systems. We next review the relevant work of real-time scheduling algorithms for multiprocessor systems.

3.3 Review of relevant work on multiprocessor scheduling

In this section, we present a global picture of the most relevant work on multiprocessor scheduling algorithms. For a survey on real-time scheduling algorithms and related issues, the reader is referred to [DB11b]. Real-time systems are computing systems that need to compute the right result at the right time. Real-time scheduling algorithms were designed, analysed and used to schedule computer programs in the on-board computer during the first manned space flight to the moon [Liu69, LL73] during the 1960s. These algorithms, RM and EDF, were intended to be used in uniprocessor computer systems [LL73].

During the 1980s and 1990s, these algorithms were extended to take into account more realistic models of preemption overhead [KAS93], deadlines [Leh90], and synchronization [SRL90]. Today, these techniques are considered mature for uniprocessor systems. Unfortunately, real-time scheduling on multiprocessors did not experience the same level of success as it did on a uniprocessor. As early as in the 1960s, it was observed by Liu [Liu69], the inventor of RM and EDF, that: "*Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors*".

Later, it was found that RM and EDF were not fit for multiprocessors; deadlines can be missed although close to 0% of the computing capacity is requested [DL78, Dha77]. This is known as *Dhall's effect*, which occurs when a task set is composed by m low-utilization tasks and one high-utilization task: consider the following synchronous periodic task set composed by $n = m+1$ tasks ($T_i = 1$ and $C_i = 2\varepsilon$, $i \in [1,m]$ (ε is an arbitrarily small value) and $T_{m+1} = 1+\varepsilon$ and $C_{m+1} = 1$) with implicit deadlines, to be scheduled on m processors according to the EDF scheduling algorithm. All tasks are released at time $t = 0$, hence task τ_{m+1} has the lowest-priority due to it having the longest deadline (at time $1+\varepsilon$). Figure 3.2 shows the time-line execution of just one job of each task to illustrate Dhall's effect. The schedule produced by the scheduling algorithm is presented according to two equivalent perspectives: tasks (above) and processors (below). Task arrivals and deadlines are represented by upwards and downwards pointing arrows, respectively. Gray rectangles represent the execution of tasks and black circles state the end of their execution. For sake of simplicity, in the description of the resulting schedule the term *job* will be avoided in favour of the more generic term *task*, even if some job-related concepts are used.

The high-utilization task τ_{m+1} misses its deadline due to the interference of the m low-utilization tasks. By letting $\varepsilon \to 0^+$, the utilization of the m tasks becomes close to 0%, and task τ_{m+1} has a utilization of 100%. Therefore, the total utilization of the task set, U_τ, is just over 100%. Considering $m \to \infty$ the system utilization, U_s, decreases to almost 0%. Thus, a deadline can be missed even if U_s is almost 0%.

For this reason, researchers initially shunned global scheduling and instead explored partitioning algorithms [DL78, Dha77, AJ03] where a set of tasks are partitioned such that all tasks in one partition are assigned to its dedicated processor and are not allowed to migrate. In this

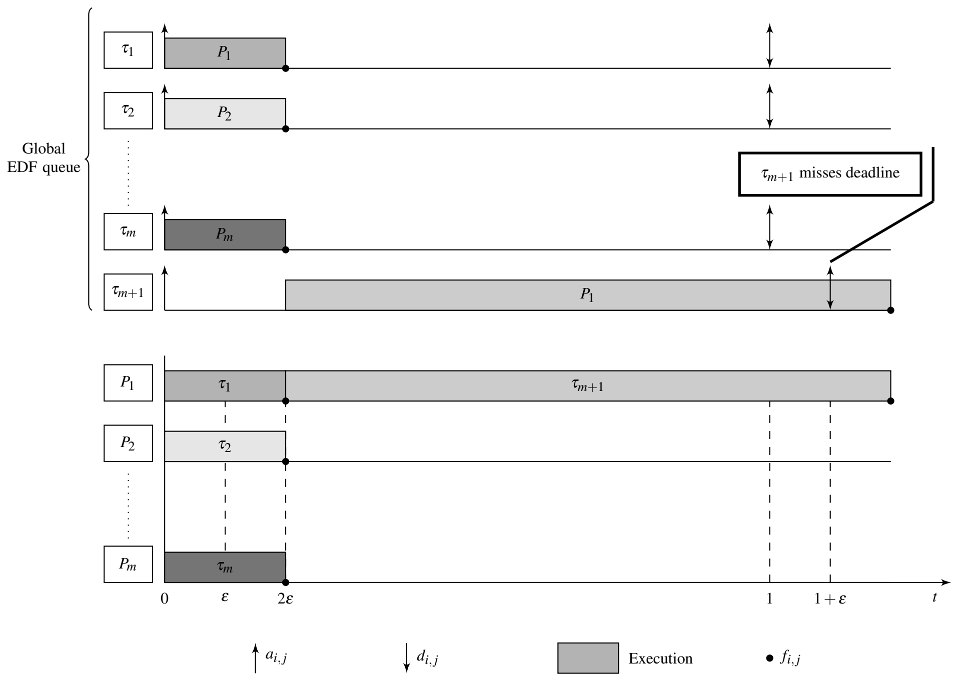


Figure 3.2: Illustration of Dhall's effect.

way, uniprocessor scheduling algorithms can be used on each core, but unfortunately, the utilization bound is limited to 50%, as the following examples shows: consider a task set composed by $n = m+1$ tasks each of which has a utilization factor barely higher than 50% ($u_i = \frac{1}{2} + \varepsilon$) to be scheduled on m processors. Assuming that tasks are assigned to processors according to the FF bin-packing heuristic, there is no processor with enough utilization capacity to accommodate task τ_{m+1}. Figure 3.3 illustrates the partitioning problem. Assuming $\varepsilon \to 0^+$, the worst-case utilization of any partitioned scheduling algorithm is limited by $(m+1)/2$ [LGDG00]. Further, with $m \to \infty$, the least schedulable utilization of any partitioned scheduling algorithm decreases to 50%.

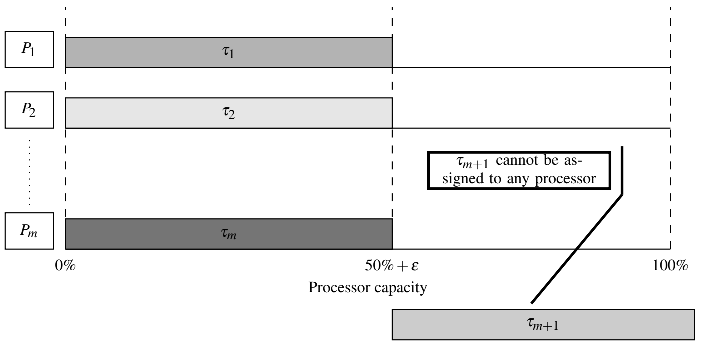


Figure 3.3: Illustration of the partitioning problem.

In the meantime, a class of algorithms called Proportionate Fairness (P-Fair) scheduling emerged [BCPV93, BCPV96]. As its name implies, under P-Fair scheduling the execution of tasks progresses proportionally over the time. For that purpose, these algorithms break a task into many unit-size sub-tasks that are scheduled with migration allowed at the sub-task boundary. Many P-Fair variants have been developed [BGP95, AS00, SA02, AS04]. These algorithms are optimal, however, at cost of many preemptions and migrations.

Recently, a class of scheduling algorithms denoted Deadline Partitioning Fairness (DP-Fair) provides the same optimality as P-Fair, but with fewer preemptions and migrations. DP-Fair algorithms [ZMM03, CRJ06, FKY08, LFS$^+$10] divide the time into time slots bounded by two consecutive task deadlines. Then, within each time slot a ready task executes during some amount of time proportional to the time slot length in accordance with some criterion, such as the task laxity, the task utilization, or the task density.

With the intent to reduce the number of preemptions and migrations incurred by global scheduling algorithms, multiprocessor scheduling algorithms with migration and static task priority were proposed like RM-US[$m/(3m-2)$] [ABJ01] and EDF-US[$m/(2m-1)$] [SB02]. Both algorithms employ a Utilization Separation (US) factor, in which, for instance, the EDF-US[$m/(2m-1)$] classifies as heavy all tasks that $u_i > m/(2m-1)$ or as light otherwise. Heavy tasks statically receive higher priority than light tasks. Light tasks are scheduled according to EDF scheduling algorithm. However, they suffered from the same limitation as partitioning, in terms of utilization bounds; that is, with $m \rightarrow \infty$ the utilization bound of the RM-US[$m/(3m-2)$] and the EDF-US[$m/(2m-1)$] tends to $m/3$ ($\approx 33.3\%$) and $m/2$ (50%), respectively. Despite this limitation, they generated an increasing interest [Lun02, AAJ$^+$02, AAJ03, Bak03, LL03] because these algorithms cause a low number of preemptions. Another approach to reduce the number of preemptions is the Fixed Priority until Zero Laxity (FPZL) scheduling algorithm [DB11a]. This algorithm combines fixed priority scheduling algorithms, like RM, with LLF. The main idea is to limit the job priority changes. Under this approach, jobs are only allowed to change the priority once. Tasks are scheduled according to the fixed priority algorithms, however, whenever a laxity of a task is zero it is given to it the highest-priority.

In the context of global fixed priority scheduling algorithms, Davis and Burns [DB11c] addressed the issue of finding schedulable task priority assignment. Their work highlights the importance of an appropriate choice of task priority assignment.

Consider again a task set composed by $n = m+1$ tasks each of which has a utilization factor barely higher than 50% ($u_i = \frac{1}{2} + \varepsilon$) to be scheduled on m processors. As mentioned before, this task set cannot be scheduled by any partitioned scheme. However, researchers [ABD05, AT06] observed that if the execution time of a task could be *split* into two pieces then it is possible to meet deadlines. For example consider, assigning task τ_{m+1} to two processors (for example P_1 and P_2) so that a task by τ_{m+1} executes $0.25 + \varepsilon/2$ time units on one of the two processors and $0.25 + \varepsilon/2$ time units on the other. This makes it possible to meet its deadline, assuming that the two *pieces* of task τ_{m+1} are dispatched so that they never execute simultaneously. This approach typifies semi-partitioning.

It should be noted that, contrarily to global scheduling algorithms where tasks can dynamically migrate across all processors, task migration rules in semi-partitioned or task-splitting scheduling algorithms are statically defined at the design phase. Hence, semi-partitioned scheduling

Table 3.1: Task set example.

Task	C_i	T_i	u_i
τ_1	9	10	0.900
τ_2	6	9	0.667
τ_3	4	7	0.571
τ_4	3	6	0.500

algorithms employ an off-line task-to-processor assignment algorithm, which also splits tasks as necessary, and a run-time task-dispatching algorithm. Many recent algorithms are based on this task-splitting idea and they differ in (i) how tasks are assigned to processors and split before run-time and (ii) how tasks (and in particular, split tasks) are dispatched at run-time.

In [ABD05], it is proposed that the second piece of a split task τ_i should arrive T_i time units later. This ensures that the two pieces of such a task do not execute simultaneously but unfortunately it requires that $D_i \geq 2T_i$ so this is only recommended for soft real-time tasks.

The remaining works that use task-splitting dispatching can be grouped according to two techniques: (i) *job-based task-splitting dispatching* and (ii) *slot-based task-splitting dispatching*. Job-based task-splitting dispatching [KY07, KY08b, KY09, LRL09, GSYY10, BDWZ12] splits a job into two or more sub-jobs and forms a sequence of sub-jobs. The arrival time of a sub-job is set equal to the absolute deadline of its preceding sub-job. Job-based task-splitting dispatching provides a utilization bound greater than 50% and few preemptions. The main drawback of job-based task-splitting dispatching is that utilization bounds greater than 72.2% have not been attained [SLB13].

Slot-based task-splitting dispatching [AT06, AB08, ABB08, BA09a, BA09b, BA11] sub-divides time into time slots whose beginning and end are synchronized across all processors; the end of a time slot of processor P_p contains a time reserve and the beginning of a time slot of processor P_{p+1} contains another time reserve, and these two reserves supply processing capacity for a split task. Slot-based task-splitting dispatching causes more preemptions than job-based task-splitting dispatching but, in return, it offers higher utilization bounds (higher than 72.2% and configurable up to 100%).

3.4 Comparison of scheduling algorithms

This section highlights the differences and similarities of the most significant real-time scheduling algorithms for multiprocessor systems. For that purpose, consider a preemptive system composed by three ($m = 3$) identical processors (P_1, P_2, and P_3) and a synchronous periodic task set composed by four ($n = 4$) independent tasks ($\tau_1, \cdots, \tau_4$) with implicit deadlines. Table 3.1 shows the parameters of each task. On the last column, the utilization of each task, u_i, is presented. The system utilization, U_s, is equal to 0.879 ($U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} u_i$).

We next present the schedule produced by different real-time scheduling algorithms (of the global, partitioned, and semi-partitioned categories) for the described system.

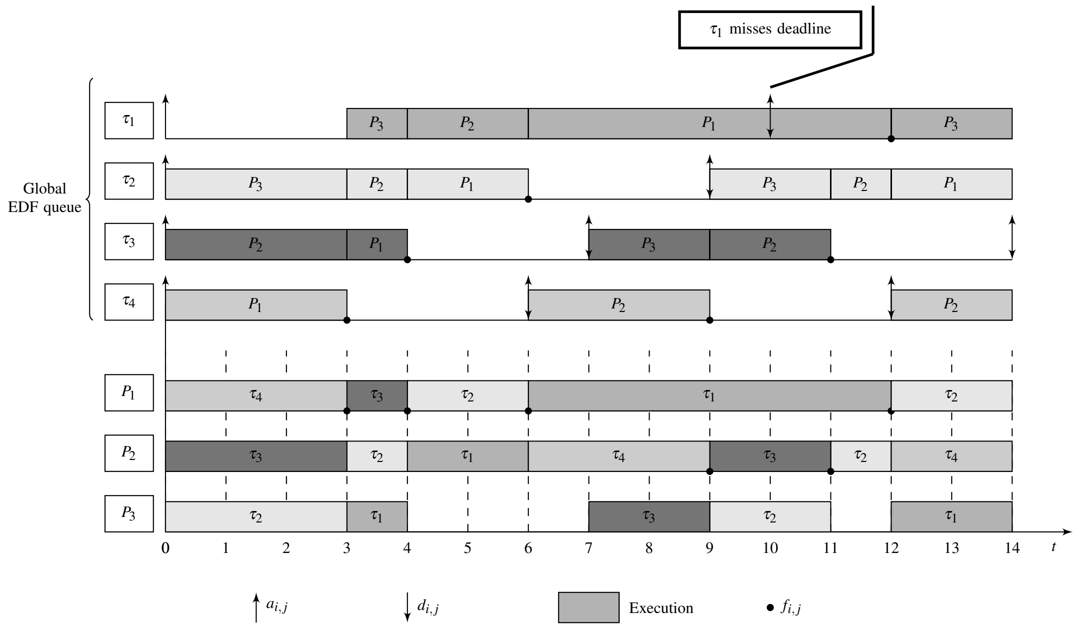


Figure 3.4: Global-EDF execution time-line.

3.4.1 Global scheduling

Under the Global-EDF scheduling policy, all ready tasks are stored into a global ready-queue and, at each time t, the m highest-priority ready tasks are executed on m processors.

For the sake of presentation, please note that in the next and also in the following figures, the highest-priority task is assigned to the lowest-indexed processor and the next highest-priority task is assigned to the next lowest-indexed processor, and so on. It should be noted that this could cause more preemptions and migrations than if some optimization heuristic was adopted. (Such as, for instance, keeping the task executing on the same processor whenever it is possible). For example, in Figure 3.4, at time 3, task τ_3 becomes the highest-priority task ready to execute and migrates from processor P_2 to processor P_1. Task τ_1 misses a deadline at time 10. Given that we will show that this task set can in fact be scheduled by other algorithms, this shows that Global-EDF is not optimal for multiprocessor systems.

The Earliest Deadline first until Zero Laxity (EDZL) scheduling algorithm for multiprocessor systems [Lee94] is a global scheduling algorithm that combines the features of two uniprocessor scheduling algorithms: EDF and LLF.

LLF [Der74] is a scheduling algorithm that assigns the highest-priority to the task with the least laxity. The laxity of a real-time task at time t is defined as the difference between the deadline and the amount of execution time remaining to complete. During execution, in addition to its C_i, T_i, and D_i parameters, a task τ_i is also characterized by its remaining execution time, $c_i^{rem}(t)$, and by its remaining time to absolute deadline, $d_i^{rem}(t)$, at time t. The Laxity, $L_i(t)$, of a task at time t is computed as $L_i(t) = d_i^{rem}(t) - c_i^{rem}(t)$ and provides a measure of urgency for this task. Whenever the $L_i(t)$ is zero the task must be scheduled to execute immediately, otherwise it will miss its deadline.

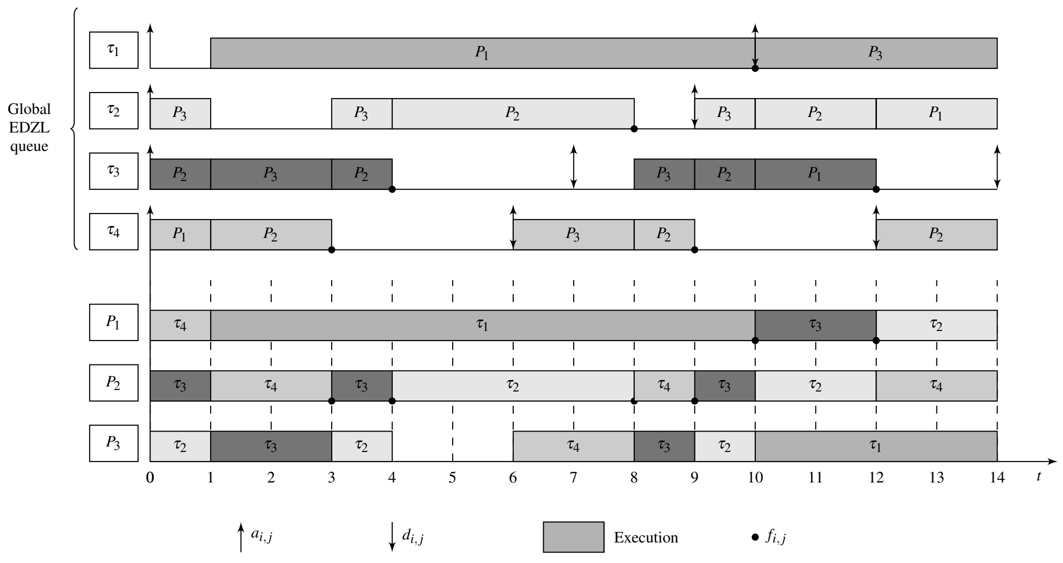


Figure 3.5: Global EDZL execution time-line.

The EDZL scheduling algorithm integrates EDF and LLF in such a manner that it behaves like EDF if there is no task with $L_i(t)$ equal to zero. When the $L_i(t)$ of some task becomes zero, the task with positive laxity that has the latest absolute deadline among the currently executing tasks is preempted in favour of the zero laxity task. Ties are broken by choosing the task with smallest execution time [Lee94]. Figure 3.5 shows the time-line execution of the task set presented on Table 3.1 when scheduled under EDZL. As it can be seen from Figure 3.5, no job misses the deadline. However, EDZL is not an optimal scheduling algorithm; that is, it is not able to schedule all feasible task sets.

Another approach is given by the P-Fair family of scheduling algorithms. The main idea of P-Fair scheduling algorithms is to provide a proportionate progress according to the task utilization. P-Fair scheduling algorithms are optimal. Figure. 3.6 presents a time-line execution considering the P-Fair scheduling algorithm [BCPV96], called P-Fairness (PF). This algorithm assures that within every interval [0, t) each task must be scheduled for $\lfloor u_i \cdot t \rfloor$ or $\lceil u_i \cdot t \rceil$ time quanta. For that purpose, PF breaks each task into an infinite sequence of quantum-length sub-tasks, τ_i^k, and each sub-task has a pseudo-release, $r_{\tau_i^k} = \left\lfloor \frac{k-1}{u_i} \right\rfloor$, and a pseudo-deadline, $d_{\tau_i^k} = \left\lceil \frac{k}{u_i} \right\rceil$.

The time interval $\left[r_{\tau_i^k}, d_{\tau_i^k} \right]$ denotes the window of the τ_i^k and each sub-task must execute within its associated window. The windows of two successive sub-tasks can partially overlap and the successor bit, $b_{\tau_i^k} = \left\lceil \frac{k}{u_i} \right\rceil - \left\lfloor \frac{k+1}{u_i} \right\rfloor$, measures the number of overlapping quantum-length units between two successive sub-tasks τ_i^k and τ_i^{k+1}. Different sub-tasks of a task are allowed to execute on different processors, but cannot do so simultaneously.

Assuming that, at time t, sub-tasks τ_i^k and τ_j^x are both ready to execute, τ_i^k has higher priority than τ_j^x (denoted $\tau_i^k > \tau_j^x$) according to the following rules, applied sequentially until any tie is broken:

(i) $d_{\tau_i^k} < d_{\tau_j^x}$,

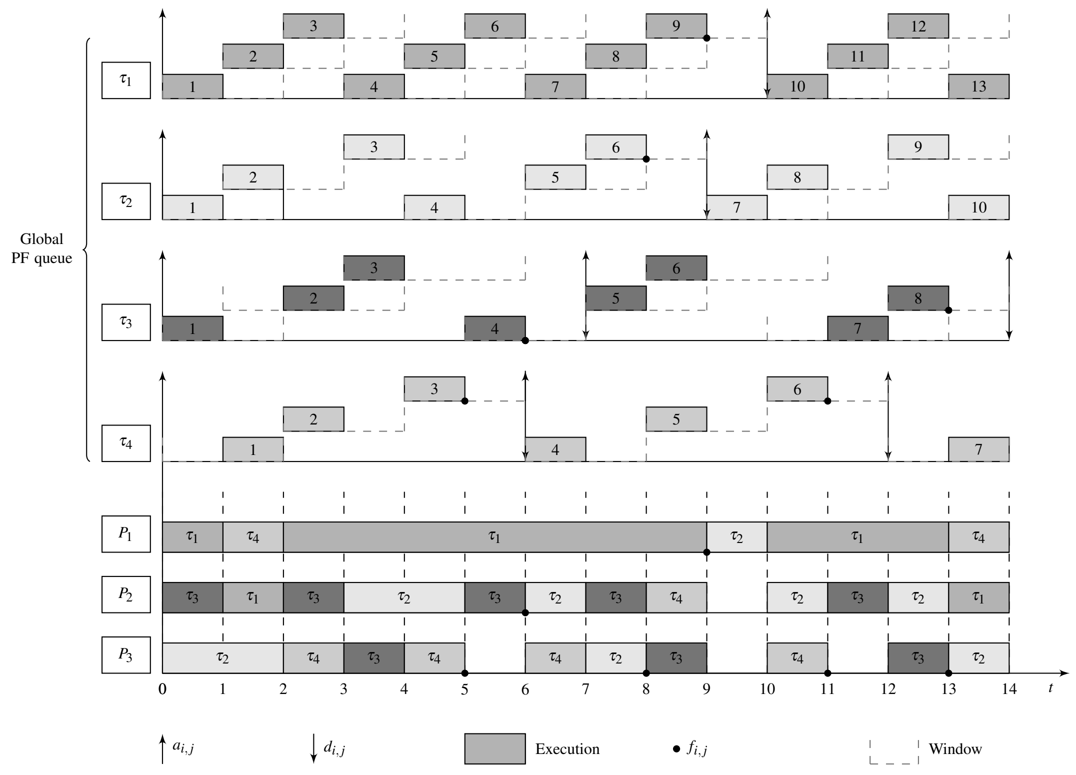


Figure 3.6: PF execution time-line.

(ii) $b_{\tau_i^k} > b_{\tau_j^x}$,

(iii) $\tau_i^{k+1} > \tau_j^{x+1}$.

If after applying all rules no sub-task has priority over the other, then the tie can be broken arbitrarily. Figure 3.6 shows the time-line execution of the task set presented on Table 3.1, where gray rectangles represent the execution of each sub-task and are labelled by its number. It also highlights two drawbacks of this algorithm that is, the higher number of task preemptions and migrations. Furthermore, another thing that should be noted is the computation complexity required, in order to take scheduling decisions.

DP-Fair scheduling algorithms are also optimal. DP-Fair divides the time into time slots demarcated by two consecutive absolute job deadlines. For each time slot, S^k, DP-Fair allocates the workload of all ready tasks and then schedules that workload. The workload is computed according to the length of each time slot. These workloads share the same deadline. Figure 3.7 shows the time-line execution of the task set example according to the DP-Wrap scheduling algorithm [LFS$^+$10], which typifies the DP-Fair approach. DP-Wrap divides the time into unequal time slots, then at the beginning of each time slot, S^k, it computes the workload of each ready task, $w_{\tau_i}^k = u_i \cdot S^k$, for that time slot. Then, it follows an adaptation of the NF bin-packing heuristic to distribute these workloads to processors. It iterates over the set of workloads and assigns each task workload, $w_{\tau_i}^k$, to lowest-indexed processor where it fits. Whenever the assigned workload exceeds the length of S^k, such workload is split between processors P_p and P_{p+1}. In this case, processor

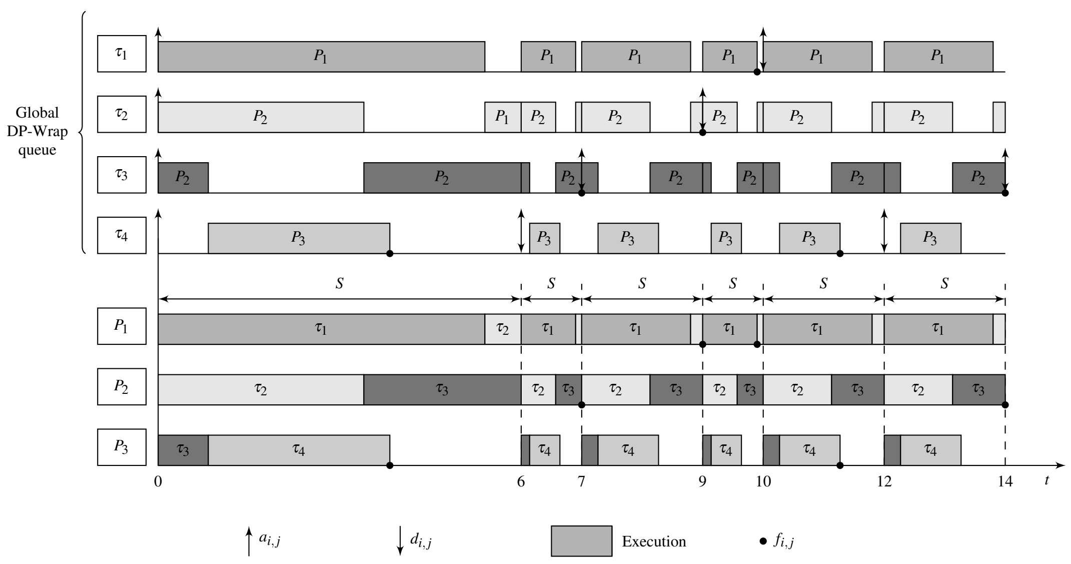


Figure 3.7: DP-Wrap execution time-line.

P_{p+1} is allocated to such workload in the beginning of the time slot while processor P_p is allocated to such workload in the end of the time slot. As it can never occur that $w^k_{\tau_i} > S^k$, this ensures that the execution parallelism of such workload is avoided.

According to [ZMM03], this approach can reduce the number of scheduling points up to 75% comparatively to P-Fair scheduling algorithms. However, it still requires a great deal of computation in order to take scheduling decisions.

3.4.2 Partitioned scheduling

Partitioned scheduling algorithms are composed by two algorithms. The off-line task-to-processor assignment algorithm and the run-time task-dispatching algorithm. The first one, assigns tasks to processors and the second one, schedules tasks to execute on the processor(s). In other words, partitioned scheduling algorithms statically assign tasks to processors and those are scheduled on each processor using a uniprocessor scheduling algorithm, like, for instance, RM or EDF.

Returning to the task set example presented on Table 3.1, it is easy to check that this task set cannot be partitioned onto three processors. Assuming that the assignment algorithm follows the FF bin-packing heuristic that assigns tasks one by one to the lowest-indexed processor where each fits, then, tasks τ_1 (with $U_1 = 0.900$), τ_2 (with $U_2 = 0.667$) and τ_3 (with $U_3 = 0.571$) are assigned to processors P_1, P_2, and P_3, respectively. Consequently, task τ_4 (with $U_4 = 0.500$) cannot be assigned to any processor, because none of them have enough capacity to accommodate this task. Nor it is possible succeed with any other bin-packing heuristic, as no two tasks can fit together on the same processor.

3.4.3 Semi-partitioned scheduling

Like partitioned scheduling algorithms, semi-partitioned scheduling algorithms are also composed of an off-line task-to-processor assignment algorithm and a run-time task-dispatching algorithm with the same purposes.

Figure 3.8 shows the time-line execution of the EDF-Window-constraint Migration (EDF-WM) scheduling algorithm [KY09]. Each task is assigned to an individual processor using any bin-packing heuristic. A task is split only when no individual processor has enough spare capacity to accommodate that task. From Figure 3.8 we observe that tasks τ_1, τ_2, and τ_3 are assigned to processors P_1, P_2, and P_3, respectively. None of them have sufficient capacity to entirely accommodate task τ_4. The execution of task τ_4 is then spread over processors with some spare processing capacity (utilization), starting from the lowest-indexed processor. Therefore, task τ_4 is split among processors P_1, P_2, and P_3. Consequently, task τ_4 migrates across those processors, but it does so in such a way that it never migrates back to the same processor within the same period, nor is it executed on more than one processor simultaneously. For that purpose, D_4 is divided into three (the number of processors on which that task τ_4 executes) equal windows. The length of each window is equal to two time units ($\frac{D_4}{3} = \frac{6}{3} = 2$), in this case. The beginning and end of these windows define the pseudo-releases and also the pseudo-deadlines related to each processor. For instance, on processor P_1 the relative pseudo-release and the relative pseudo-deadline are at time zero and two, respectively, while, on processor P_2 the relative pseudo-release and the relative pseudo-deadline are at time two and four, respectively, and so on.

An important issue is how to compute how much time task τ_4 can be executed on each processor, that is, which are the $C_4[P_1]$, $C_4[P_2]$, and $C_4[P_3]$. An obvious issue here is that the execution of task τ_4 on processors P_1, P_2, and P_3 cannot violate the timing requirements of the already assigned tasks. In [KY09] it is defined how to compute that parameter. According to this, $C_4[P_1]$, $C_4[P_2]$ and $C_4[P_3]$ are approximately equal to 0.5, 1.5 and 1 time units, respectively. The run-time task-dispatching algorithm schedules tasks on each processor under the uniprocessor EDF scheduling algorithm.

The Sporadic-EKG (S-EKG) scheduling algorithm [AB08], was designed to schedule real-time sporadic tasks – the name EKG, stands for "EDF with task splitting and k processors in a Group". It can be configured to achieve different levels of utilization bounds. It is categorized as semi-partitioned, since it assigns $m-1$ tasks to two processors and the rest to only one processor. S-EKG divides time into slots of length $S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i)$, where δ is a designer-set integer parameter that controls the frequency of migration of tasks assigned to two processors. $\mathrm{UB}_{\mathrm{S-EKG}}$ is the utilization bound of S-EKG scheduling algorithm and is determined by the chosen value for δ (in Chapter 4, we provide more details). A task whose utilization exceeds $\mathrm{UB}_{\mathrm{S-EKG}}$ (henceforth called *heavy task*) is assigned to a dedicated processor. Then, the remaining tasks are assigned to the remaining processors in a way similar to NF bin-packing. Assignment is done in such a manner that the utilization of processors is exactly $\mathrm{UB}_{\mathrm{S-EKG}}$, except for the last processor, where it could be less. Task splitting is performed whenever a task cause the utilization of a processor to

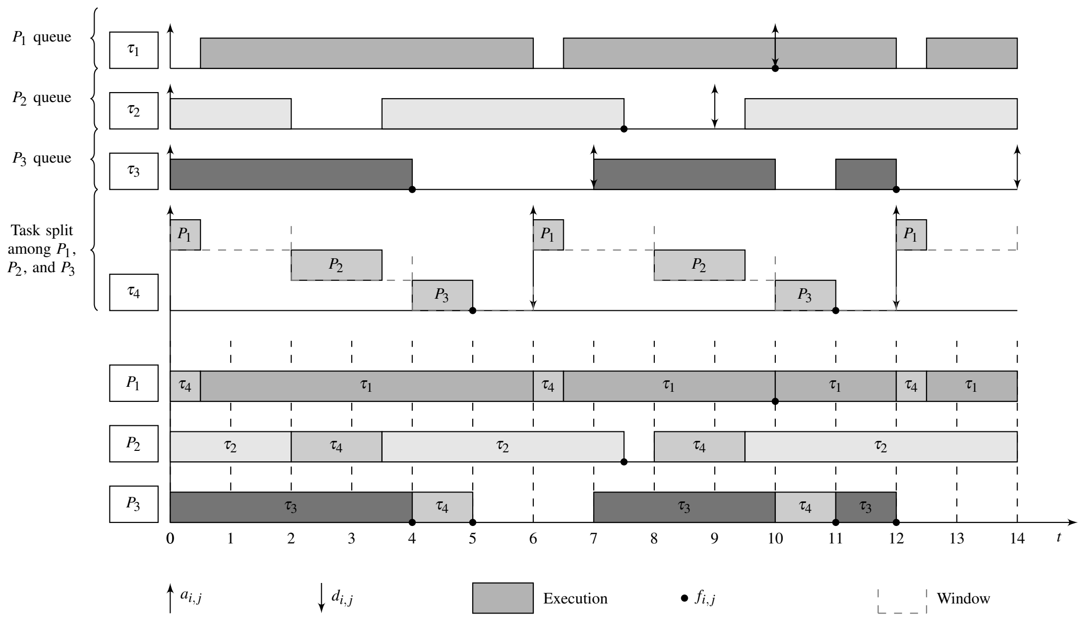


Figure 3.8: EDF-WM execution time-line.

exceed UB_{S-EKG}. In this case, this task (henceforth called a split task) is split between the current processor P_p and the next one P_{p+1}. Then, in these processors, there are time windows (called reserves) where this split task has priority over other tasks (henceforth called non-split tasks) assigned to these processors. The lengths of the reserves are chosen such that no overlap between them occurs, the split task can be scheduled, and all non-split tasks can also meet deadlines. The non-split tasks are scheduled under the EDF scheduling algorithm.

Figure 3.9 shows the time-line execution of the S-EKG scheduling algorithm for the task set example presented in Table 3.1 with a value of δ equal to four. According to [AB08], the utilization of the S-EKG algorithm for δ equal to four is approximately 88.9% ($UB_{S-EKG} = 4 \cdot (\sqrt{\delta \cdot (\delta+1)} - \delta) - 1 \approx 0.889$). Then, since τ_1 is a heavy task, it is assigned to a dedicated processor, P_1. τ_2 is assigned to processor P_2, but assigning task τ_3 to processor P_2 would cause the utilization of processor P_2 to exceed UB_{S-EKG} ($0.667 + 0.571 > 0.889$). Therefore, task τ_3 is split between processor P_2 and processor P_3. A portion of task τ_3 is assigned to processor P_2, just enough to make the utilization of processor P_2 equal to UB_{S-EKG}, that is 0.222. This part is referred to as $y[P_2]$ and the remaining portion (0.349) of task τ_3 is assigned to processor P_3, and is referred to as $x[P_3]$. Finally, task τ_4 is assigned to processor P_3.

On a dedicated processor, the task-dispatching algorithm is very simple, whenever the assigned task is ready to be executed, the processor executes that task. The time is divided into time slots and non-dedicated processors usually execute both split and non-split tasks. For that purpose, the time slot might be divided into at most three parts. The $y[P_p]$ and $x[P_{p+1}]$ time units are reserved for executing the split task shared by P_p and P_{p+1}. However, it is important to note that one split task executes one portion on processor P_p and the remaining portion on another processor P_{p+1}. This means that a split task will execute on both processors but not simultaneously. The remaining

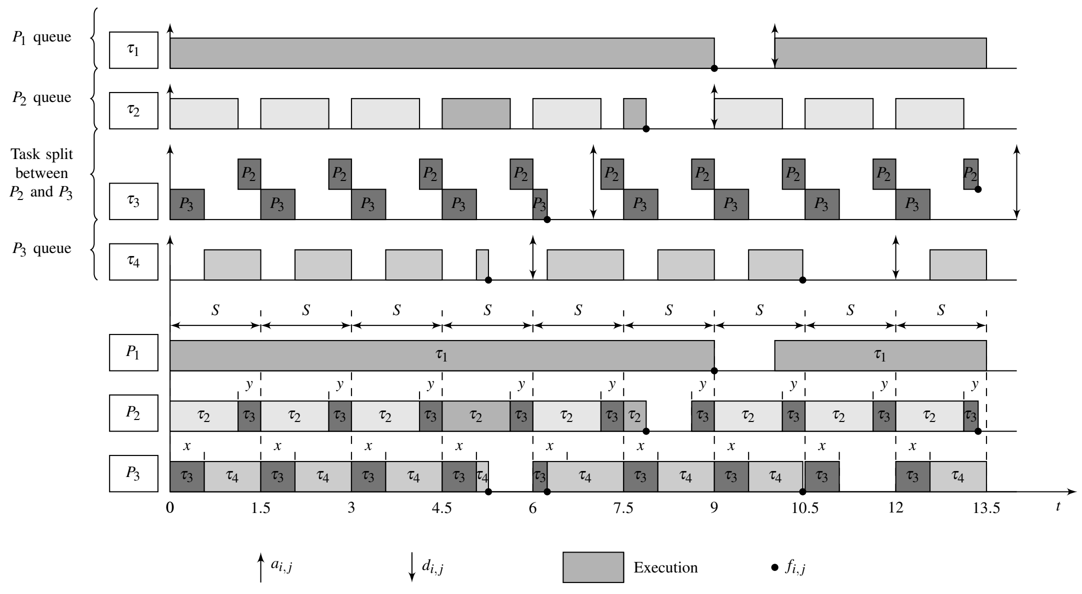


Figure 3.9: S-EKG execution time-line.

part of the time slot is reserved for executing non-split tasks, on each processor.

The run-time task-dispatching algorithm works over the time slot of each processor and, whenever the dispatcher is running, it checks to find the time elapsed in the current time slot: (i) if the current time falls within a reserve ($x[P_p]$ or $y[P_p]$), if the assigned split task is ready to be executed, then the split task is scheduled to run on processor, otherwise, the non-split task with the earliest deadline is scheduled to execute on the processor; (ii) if the current time does not fall within a reserve, the non-split task with the earliest deadline is scheduled to run on processor.

In the execution time-line illustrated in Figure 3.9, the time slot length is $S = 1.5$ because $\min_{\tau_i \in \tau}(T_i)$ is equal to six (the minimal inter-arrival time of task τ_4) and δ is equal to four. The split task τ_3 executes only within the $x[P_3]$ and $y[P_2]$ reserves. Outside its reserves, it does not use the processor, even if that is free. In contrast, the non-split tasks execute mainly outside the $x[P_p]$ and $y[P_p]$ reserves but potentially also within the reserves, namely, when there is no split task ready to be executed. There is one clear situation in Figure 3.9 that illustrates this. On processor P_3, within the time slot that starts at time six, task τ_4 executes on $x[P_3]$ reserve because the split task τ_3 has finished its execution.

The Notional Processor Scheduling - Fractional capacity (NPS-F) [BA09b] algorithm, uses a somewhat different approach from the scheduling algorithms previously presented in this section. While the previous scheduling approaches are based on the concept of a task, the NPS-F uses an approach based on bins. To each bin one or more tasks are assigned and there is a one to one relation between each bin and each notional processor, which is a sort of server. Then, each notional processor schedules tasks of the respective bin under the EDF scheduling policy. In turn, all notional processors are implemented upon the m physical processors, P_1 to P_m, by the means of time reserves. In Chapter 4, we present more details about the NPS-F scheduling algorithm.

Assume that there are four bins, b_1 to b_4, that are filled with the tasks of the task set presented

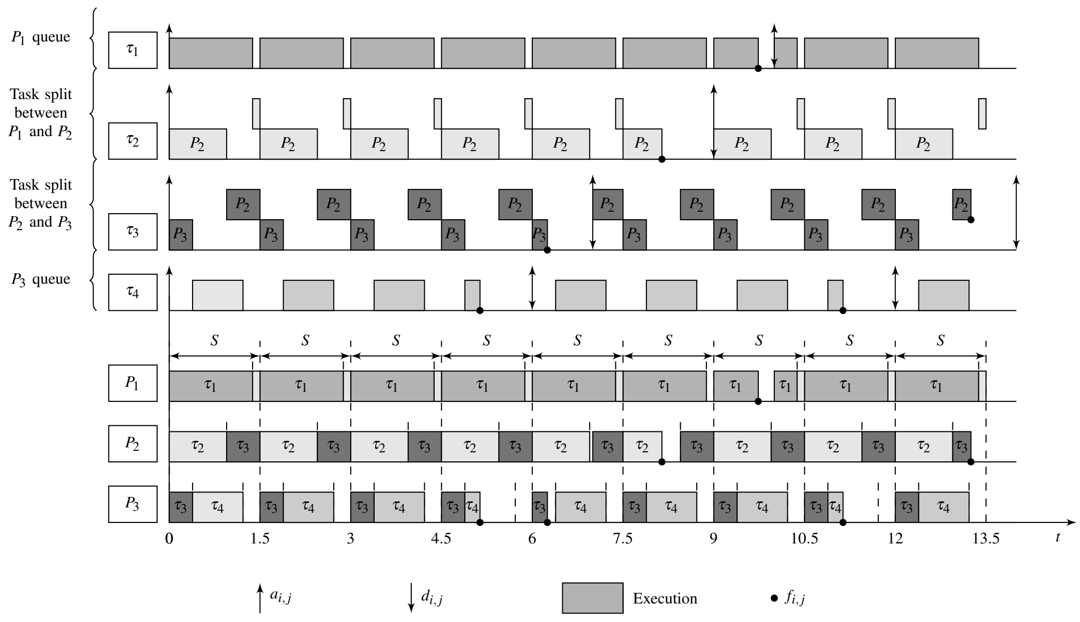


Figure 3.10: NPS-F execution time-line.

in Table 3.1 in such a manner that task τ_1 is in b_1, τ_2 is in b_2 and so on. Then, the utilization of each bin is equal to the utilization of the associated task. According to [BA09b], to ensure the schedulability of all tasks, the utilization of each bin (denoted by $U[b_x]$) must be inflated (the reasons for this are also explained in Chapter 4). After this, each bin is instantiated as to one notional processor, Ψ_q.

The notional processors are assigned to the physical processors as follows. Ψ_1 is assigned to processor P_1, since P_1 has some remaining capacity available (the capacity of each processor is one), Ψ_2 is assigned to P_1 and also to P_2. The remaining capacity available on processor P_2 is not enough for Ψ_3, thus Ψ_3 is split between P_2 and P_3. Finally, Ψ_4 is assigned to P_3. To finish the assignment, the length of reserves has to be computed. For that purpose, the time is divided into fixed length time slots $S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i)$. In this example, $\min_{\tau_i \in \tau}(T_i)$ is equal to six and, consequently, S is equal to $\frac{6}{4} = 1.5$, because δ is set equal to four (note that, in the context of the NPS-F, δ has the same purpose as in the context of the S-EKG). The notional processor reserve on each physical processor is computed by multiplying the notional processor processing requirements with S.

The task-dispatching algorithm is very simple (see Figure 3.10), tasks are only allowed to execute within their respective reserves; that is, within the reserves of the respective notional processors.

3.4.4 Discussion

Partitioned scheduling algorithms partition the set of tasks in the system such that all tasks belonging to a given partition are executed always on the same processor. Thus, after partitioning, the scheduling on a multiprocessor system reduces to multiple instances of the scheduling problem on a uniprocessor. However, we share the same opinion of Baker [Bak10] who states: "*A multiprocessor system composed by m processors cannot simply be broken down into m uniprocessor systems, because a multiprocessor system is not confined to processors. Cache and memory bus conflicts introduce other dimension to the scheduling problem that it is not present on uniprocessor systems*".

As a consequence of the partitioning such algorithms are not generally work-conserving, which in turn may cause a low utilization. This is because an idle processor cannot be used to run ready tasks which have been allocated a different processor. As it has been proven via construction of some *pathological cases*, their utilization bound is 50%.

Global scheduling algorithms try to overcome this limitation by allowing tasks to migrate from one processor to another. At any time instant, the *m* highest-priority runnable tasks in the system can be selected for execution on the *m* processors. Some scheduling algorithms of this class are work-conserving and present a utilization bound of 100%, but at a cost of many preemptions and migrations. Although recent work has made some progress towards reducing the number of preemptions and migrations [ZMM03, FKY08, LFS$^+$10] as well as scheduling overheads [KY11], the implementation of such algorithms requires the use of a global shared queue, which raises scalability issues because of the need to prevent race conditions in the access to that queue. Furthermore, some global scheduling algorithms require, at run-time, complex computations in order to take scheduling decisions

Semi-partitioned, or task-splitting, scheduling algorithms try to address these issues by limiting task migration. Typically, under task-splitting scheduling algorithms, most tasks execute only on one processor as in partitioned algorithms, whereas a few split (or *migratory*) tasks may execute on several processors, as in global scheduling algorithms. This approach produces a better balance of the workload among processors than partitioning (and consequently presents a higher utilization bound) and also reduces the contention on shared queues and the number of migrations, compared to global scheduling (by reducing the number of migratory tasks). Furthermore, the run-time task-dispatching algorithms benefits from the static nature of task-to-processor assignment performed pre run-time, i.e., off-line at the design stage.

Generally, task-splitting approaches can be classified as job-based or slot-based. Slot-based task-splitting scheduling algorithms present the highest utilization bound among scheduling algorithms that do not share a global queue. However, in [BBA11], a survey and experimental evaluation of semi-partitioned algorithms are presented, both in terms of scheduling potential and in terms of implementation practicality, the authors also attempt to quantify the deterioration of (theoretical) schedulability due to overheads and working set loss (especially in the case of memory-intensive workloads). In those experiments, this performance deterioration was more marked for slot-based (the NPS-F) than for job-based (the EDF-WM) scheduling algorithms, something which the authors largely attribute to the NPS-F design as stated in Observation 6 in [BBA11]: "*NPS-F schedulability is heavily constrained by the pessimistic assumptions made in the bin-packing heuristics of NPS-F's assignment phase, and by higher preemption and migration delays*".

In our view, some of the conclusions in [BBA11] regarding NPS-F are debatable. For instance, there exist many conceivable ways in which the various run-time overheads can be *safely* incorporated into the analysis. Some of these may be more penalising to one scheduling algorithm than to another, in terms of added pessimism. Indeed, it appears (although the exact approach is in fact not properly documented in [BBA11], hence our reservation) that the authors *model* all overheads as an increase to the execution time of each task. While this is a valid approach for some kinds overheads, it is entirely unreasonable and unnecessarily pessimistic for others. For example, the overhead incurred at the start of each reserve, in fact only affects one job per-time slot per-reserve; yet, in [BBA11] it is applied to *all* tasks in that reserve. By comparison, in the context of this dissertation, we formulate a much more efficient approach for incorporating that overhead.

Another point of concern is that of task migrations; the approach in [BBA11] relies on the existing Linux mechanisms for task migration, which introduce large delays, because of the fact that each task migration necessitates the simultaneous locking of *both* the destination and the source ready-queue. Even worse, the approach for performing task migrations in [BBA11] creates a scheduling dependency across processors, because it relies on IPIs (which, in the process, introduce additional delays); that is, the origin processor informs the target processor when it will free the migrating task. All in all, the evaluation in [BBA11] is an evaluation of NPS-F for the particular implementation (i.e. as emulated on top of LITMUS$^{\mathrm{RT}}$), hence the results have to be interpreted in that context. By comparison, our implementation, derived in the context of this dissertation, offers a much more efficient approach to task migration (without any need for ready-queue locking).

Finally, in [BBA11] the original form of NPS-F is examined [BA09b], which employs time slot boundaries synchronised across all processors. This variant suffers from the instantaneous migration problem earlier highlighted and deeply discussed in Section 4.5. Yet in [BA11], the staggered time slot approach is formulated, without any penalty in terms of schedulability, which eliminates or, in the worst-case, greatly reduces such effects.

In any case, we believe that the study presented in [BBA11], largely validates the semi-partitioned scheduling approaches for real-time systems and highlights the importance of the implementation for semi-partitioned scheduling algorithms. Still, we believe that the schedulability deterioration, observed therein in the case of slot-based task-splitting is mainly due to the implementation and *not* the migration model.

Since the scheduling algorithms under consideration in this dissertation are those classified as slot-based, in the next section, we present a historical perspective of this category of algorithms.

3.5 Historical perspective of slot-based scheduling algorithms

In 2006 the first slot-based task-splitting scheduling algorithm called EKG, was introduced (nowadays often retroactively referred to as “Periodic EKG” or “the original EKG”) [AT06]. Recall that, we consider slot-based task-splitting scheduling algorithms those that divide the time into time slots and, at design stage, map tasks to processors, and then, at run-time, schedule tasks. Due to this view, we do not consider DP-Fair scheduling algorithms as slot-based task-splitting. EKG was limited to the scheduling of periodic tasks only. Under this scheme, time is divided into time slots of unequal (in the general case) duration, with the time boundaries of a given time slot corresponding to the time instants of two consecutive job arrivals (possibly by different tasks) in the system. Most tasks are partitioned, but at most $m-1$ tasks (with m being the number of processors) are split – each, between a corresponding pair of successively indexed processors, P_p and P_{p+1}. Within each time slot, the first piece of a split task is executed at the end of the time slot on the first processor, P_p, utilized by that task, and the second piece is executed at the start of the time slot on the other processor, P_{p+1}. All other tasks are executed under EDF on their respective processors. The basic form of the algorithm has a utilization bound of 100%.

Clustered variants of EKG divide the system into m/k clusters of k processors each. Such clustering may be used to trade off utilization bound for fewer preemptions and migrations.

However, the original EKG suffered from the limitation that, by design, it could not handle sporadically arriving tasks. This was because split task budgets in each time slot were proportional to the task utilization and the time slot length. However, given that time slots were formed between successive job arrivals, it was necessary to know in advance the time of next job arrival in order to compute these budgets. With periodic tasks, this is not a problem, since arrival times are deterministic and may be computed in advance. However, with sporadic arrivals, this information is neither known in advance nor predictable.

This is why, in 2008, Andersson and Bletsas [AB08] came up with an adapted design that came to be known as Sporadic-EKG (S-EKG). In order to accommodate sporadic tasks, this algorithm *decouples* the boundaries from the time instants of job arrivals. Therefore, all time slots are of equal length. However, given that tasks can now arrive at *unfavorable* offsets relative to the time slot boundary, there is a penalty to be paid in terms of utilization bound: in order to ensure schedulability, processors can no longer be filled up to their entire processing capacity. Via a designer-set parameter, which controls the time slot length, S-EKG can be configured for a utilization bound from 65% to arbitrarily close to 100%, at the cost of more preemptions and migrations.

Later in the same year, Andersson *et al.* [ABB08] came up with a version of S-EKG, named EDF-SS (that stands for EDF scheduling of non-split tasks with Split tasks scheduled in Slots), which can handle arbitrary deadline tasks (whereas its predecessor was formulated in the context of implicit deadline tasks). However, due to different task assignment heuristics, one version does not dominate the other. Moreover, in part due to this *break* from the previous variant, no utilization bound above 50% has been proven for EDF-SS.

The three EKG variants discussed above share a basic design: at most $m-1$ tasks are split, each between two successively indexed processors. The first piece of a split task executes at the end of the time slot on the first processor used by that task and the second piece is executed at the start of the time slot on the other processor. However, a less prescriptive approach to splitting the execution time of tasks between processors, while at the same time maintaining a slot-based dispatching, was soon devised.

In 2009, Bletsas and Andersson [BA09a] presented NPS (that stands for Notional Processor Scheduling), rapidly superseded entirely by NPS-F [BA09b, BA11]. This algorithm (and its short-lived predecessor) employ a server-based approach. Each server (termed notional processor in the context of that work) serves one or more tasks employing an EDF scheduling policy. Under NPS-F, it is the execution time of these servers which is split – not directly that of the underlying tasks served. In principle, this allows improved efficiency in the utilization of a multiprocessor system. NPS-F has a utilization bound of 75% configurable up to 100% at cost of more preemptions and migrations. Compared to S-EKG, for corresponding configurations characterised by roughly the same number of preemptions, NPS-F has a higher utilization bound. However, a downside to splitting servers instead of tasks is that the number of migrating tasks is not bounded *a priori* and typically exceeds $m-1$.

3.6 Summary

In this chapter, we have addressed several issues related to real-time scheduling for multiprocessors. We presented an overview of the real-time theory, in general, and also specifically for multiprocessor systems. We discussed the categories of scheduling algorithm for multiprocessor systems highlighting the advantages and disadvantages of each category. We next focused on some relevant scheduling algorithms for multiprocessor systems. Finally, we presented a historical perspective of the sub set of scheduling algorithms that is the focus of this work: slot-based task-splitting algorithms.

Chapter 4

Slot-based task-splitting scheduling algorithms

4.1 Introduction

The real-time research community has produced many scheduling algorithms for multiprocessor systems. Some of them are, theoretically, perfect, because they have the ability to schedule any feasible task set; that is, they are optimal. However, when we jump from the theoretic field to practice field, we sometimes get disappointed. Some of those scheduling algorithms are not easy to implement and the amount of practice issues that needs to be addressed eliminates the theoretical benefits of those scheduling algorithms. The explanation is simple, many of them are based on a set of assumptions that have no correspondence with real systems. We believe that it is important to develop new scheduling algorithms but they cannot be detached from practical issues. For that purpose, real implementations play an important role, because they are fundamental for testing, debugging, evaluating, and also for learning.

In this chapter, we aim to provide all details of slot-based task-splitting scheduling algorithms and their implementation in a PREEMPT-RT-patched Linux kernel version. We present our assumptions about the hardware architecture for the host system. We next present a unified description of the slot-based task-splitting scheduling algorithms under consideration showing the generic principles of slot-based task-splitting and highlighting some particular features of specific slot-based task-splitting schemes. We address and discuss one practical issue of the slot-based task-splitting scheduling algorithms, the instantaneous migration problem. We describe the implementation of the slot-based task-splitting scheduling algorithms under consideration in this dissertation into the Linux kernel. Finally, the scheduling overheads incurred by these scheduling algorithms are described.

4.2 Assumptions about the hardware architecture

We assume identical processors, i.e. (i) all processors have the same instruction set and data layout (e.g. big-endian/little-endian) and (ii) execute at the same speed.

We also assume that the execution speed of a processor does not depend on activities on another processor (e.g. whether the other processor is busy or idle or which task it is busy executing) and also does not change at run-time. In practice, this implies that (i) if the system supports SMT – called hyperthreading by Intel – then this feature must be disabled and (ii) features that allow processors to change their speed (e.g. power and thermal management) must be disabled.

We assume that each processor has a local timer providing two functions: (i) one function allows reading the current real-time (i.e. not calendar time) as an integer; (ii) and another function allows setting up the timer to generate an interrupt at x time units in the future (x being configurable).

We assume that the memory system follows the shared memory model. Hence, the access time memory is uniform for any processor.

Finally, we assume that there is a perfect synchronization among all processor clocks; that is, the local clocks of all processors are synchronized.

4.3 Generic slot-based task-splitting

In this section, we present a generic, unified, high-level description of the slot-based task-splitting scheduling algorithms that covers the most relevant, either from utilization bound or implementation perspective, slot-based task-splitting scheduling algorithms: S-EKG and NPS-F. From this section onwards, whenever we refer to slot-based task-splitting algorithms, we are only considering algorithms where the time is divided into equal-duration time slots (more specifically, S-EKG [1] and NPS-F).

For example, we overlook the (original) EKG scheduling algorithm [AT06] whose applicability is limited to periodic task sets with implicit deadlines. That algorithm requires knowledge about future task arrivals to take scheduling decisions, which may be difficult to achieve in a real system. We also exclude DP-Fair [LFS$^+$10] algorithms such as DP-Wrap because under such approaches use time slots of unequal duration and which in fact, are constructed *dynamically* (i.e. on-the-fly), during the execution of the system. The arrival of a job may cause the current time slot to terminate early and its remainder to become a new time slot, with task execution time budgets having to be calculated anew, so as to accommodate the remaining execution times of all tasks (which, therefore, need to be tracked, at run-time). From a practical point of view, such dynamic time slot formation, the extensive associated *book-keeping* and the need to recalculate reserves (potentially at every new job arrival), considerably increase the complexity of the scheduler.

[1] We focus on S-EKG and not in EDF-SS, because latter is a version of the former that explores different bin-packing heuristics to get fuller processors.

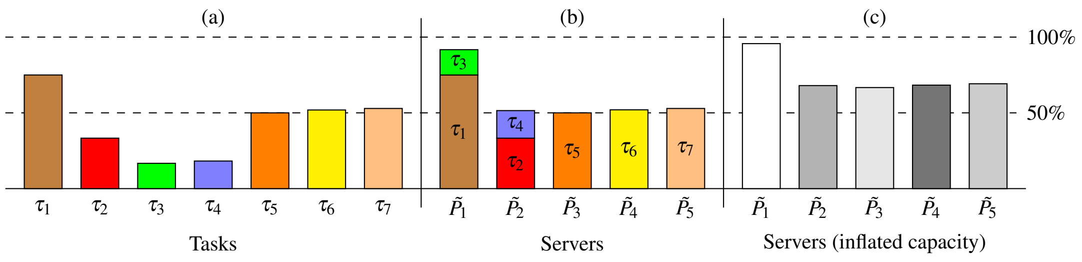


Figure 4.1: Task-to-server mapping.

A key concept of the generic slot-based task-splitting scheduling algorithm is that of a *server*. A server is a logical entity that provides computation services to tasks and has a maximum capacity equal to that of the underlying physical processors. Thus, in the generic template for a slot-based task-splitting scheduling algorithm, a task is first mapped to a server, which is then assigned to one or two processors. A processor may be allocated to multiple servers over time, but at *any time* a processor is allocated to only one server and one server is served by at most one processor. A time reserve is a time window during which a processor is exclusively allocated to a server, i.e. executes tasks of only that server. Therefore, time reserves on a processor must be non-overlapping. Furthermore, time reserves are periodic and we call their period, which is the same for all reserves, the time slot.

The scheduling of a set of tasks in the generic algorithm comprises two procedures, one that is performed off-line and another that is executed at run-time. The off-line procedure maps tasks to servers, determines the computation requirement of each server and allocates reserves to the processors in order to ensure that each server has the required capacity for executing its tasks. The run-time procedure is a task-dispatching scheduling algorithm that runs on each processor which uses EDF to choose the task of the server associated to the currently active time reserve.

We now describe the off-line procedure. The generic algorithm specifies a procedure composed of four steps and what is performed in each step, but it does not prescribe any particular algorithm for any of the steps. This is up to the specific scheduling algorithms. To illustrate the generic algorithm, we use an example. The figures illustrating its application were obtained by using the algorithms specified for NPS-F, later described in Subsection 4.4.2. The task set, τ, in our example is comprised of seven tasks, τ_1 to τ_7. Inset (a) of Figure 4.1 represents each task in that set by a rectangle whose height represents that task's utilization.

The first step of the off-line procedure is mapping tasks to servers, which we denote $\tilde{P}_q$. The generic slot-based task-splitting algorithm does not prescribe how tasks are mapped to servers. Each specific scheduling algorithm can use its own mapping. Inset (b) of Figure 4.1 shows the task-to-server mapping obtained by the FF bin-packing heuristic as is done under NPS-F. The server capacity is 100%.

The second step of the off-line procedure is to determine the (computation) capacity of each server. This is obtained by *inflating* the sum of the utilizations of the server's tasks. Utilization inflation is required to compensate for the time intervals during which a server may have ready

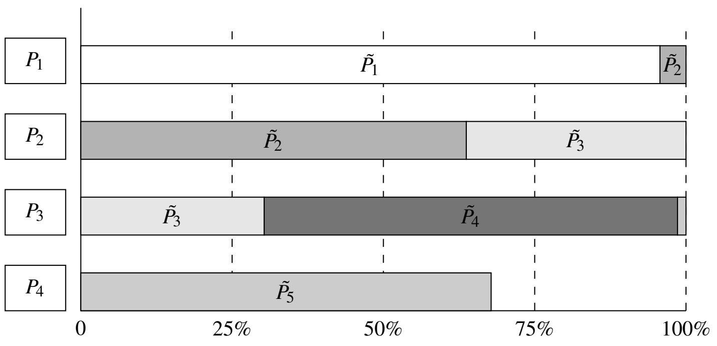


Figure 4.2: Server-to-processor assignment.

tasks, but none of them can be executed. Such a scenario may arise because none of the server's time reserves are active, and a processor executes tasks of only the server associated to its current time reserve. Several methods can be used to determine by how much to inflate a server capacity, some of these depend on the specific scheduling algorithm. In Section 5.4, we present one such method in the context of the new schedulability analysis. At this point, we assume that such a method exist, and illustrate its application in inset (c) of Figure 4.1.

The third step of the off-line procedure is to allocate processors to servers. Again, the generic algorithm does not prescribe how this allocation is done. Figure 4.2 illustrates the server-to-processor assignment obtained by applying the algorithm used in NPS-F to our running example. Servers $\tilde{P}_1$ and $\tilde{P}_4$ are assigned to only one processor each, and are, hence, classified as *non-split servers*; whereas servers $\tilde{P}_2$, $\tilde{P}_3$, and $\tilde{P}_5$ are *split servers* because they are assigned to two processors each.

The fourth and last step of the off-line procedure is to define the time reserves for each processor. Again, the generic algorithm does not prescribe how this is done. Figure 4.3 illustrates the reserves determined by the application of an algorithm used by NPS-F to our running example. In this case, all processors synchronize at the beginning of each time slot. On each processor P_p, the time slot can be divided into three reserves, at most: $x[P_p]$, $y[P_p]$, and $N[P_p]$. The $x[P_p]$ reserve occupies the beginning of the time slot and it is reserved for the split server shared by processor P_p and processor P_{p-1}, if any. The $y[P_p]$ reserve occupies the end of the time slot and it is reserved for the split server shared by processor P_p and processor P_{p+1}, if any. The remaining part, $N[P_p]$, is reserved for the non-split server assigned to processor P_p, if one exists.

Before describing of the run-time task-dispatching algorithm, let us highlight that it executes on a per-processor basis. The run-time task-dispatching algorithm is a two-level hierarchical scheduling algorithm. The top-level handles the time division into time reserves and identifies the currently active server for each processor according to the server-to-processor assignment defined by the off-line procedure. The low-level scheduler schedules tasks of the currently active server, on each processor.

For notational purposes, we consider a unlimited set of servers, which are equivalent to physical processors in terms of processing capacity, indexed in the range $\tilde{P}_1$ to $\tilde{P}_k$. The set of tasks that can be assigned to a server $\tilde{P}_q$ (denoted by $\tau[\tilde{P}_q]$) is limited by its processing capacity that is equal

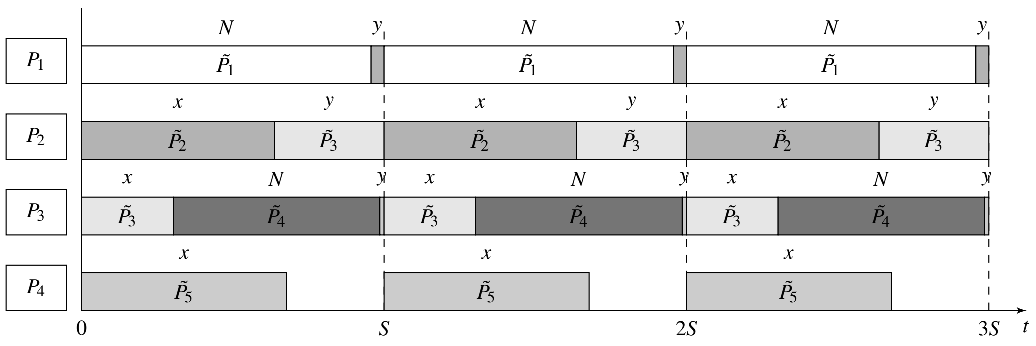


Figure 4.3: Run-time dispatching time-line produced by the slot-based task-splitting algorithm for the task set example.

to 1.0 (100%). The utilization of a server $\tilde{P}_q$, $U[\tilde{P}_q]$, is given by:

$$U[\tilde{P}_q] = \sum_{\tau_i \in \tau[\tilde{P}_q]} u_i \tag{4.1}$$

and the symbol $U^{infl}[\tilde{P}_q]$ is used to denote the inflated capacity of server $\tilde{P}_q$.

We have presented the generic slot-based task-splitting algorithm, in the next section we show how to correlate the S-EKG and NPS-F with this generic algorithm.

4.4 Overview of specific scheduling schemes

The specific algorithms under consideration differ in how they implement the specific mechanisms discussed earlier, for example, in how they map tasks to servers or in how they map servers to processors. Therefore, before introducing the implementation (in this section) and the new generalised schedulability analysis (in Chapter 5), it is worth highlighting, without going into too much detail, some of these aspects for S-EKG and NPS-F. Where deemed necessary, we also briefly discuss aspects of the original scheduling theory, which our new theory supersedes.

4.4.1 Discussion of S-EKG and its original theory

The S-EKG algorithm shares many features with the generic algorithm. Both are slot-based; both use an off-line procedure to map tasks to processors and a run-time algorithm that uses EDF to choose the running task. A major difference between the two is that S-EKG, as described in its original publication [AB08], does not explicitly use the concept of server. Instead, it assigns tasks to processors directly, employing a procedure similar to the NF bin-packing heuristic that we describe next: in S-EKG, the task-to-processor mapping procedure strives to ensure that the utilization of each processor is equal to $\text{UB}_{\text{S-EKG}}$ (the theoretical utilization bound of the algorithm). It iterates over the set of tasks. If a task has a utilization that exceeds $\text{UB}_{\text{S-EKG}}$, it assigns that task to a dedicated processor. Otherwise, it assigns the task to the next available processor whose utilization is lower than $\text{UB}_{\text{S-EKG}}$. In this case, if task τ_i cannot be integrally assigned to the current processor, P_p, without exceeding that bound, it is split between that processor and the next one, P_{p+1}, so that P_p ends up utilized exactly by $\text{UB}_{\text{S-EKG}}$ and P_{p+1} receives the remaining share of τ_i. Consequently, the number of split tasks is at most $m-1$ and there is at most one task split between each pair of successively indexed processors P_p and P_{p+1}. Furthermore, in a schedulable system, the utilization of every non-dedicated processor (except possibly the last one) is exactly $\text{UB}_{\text{S-EKG}}$.

S-EKG uses a designer-set integer parameter δ, which determines the length of the time slot according to Equation 4.2.

$$S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i) \tag{4.2}$$

This parameter also affects the utilization bound, $\text{UB}_{\text{S-EKG}}$, and the inflation factor, α, as follows:

$$\text{UB}_{\text{S-EKG}} = 4 \cdot (\sqrt{\delta \cdot (\delta+1)} - \delta) - 1 \tag{4.3}$$

$$\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta+1)} + \delta \tag{4.4}$$

Depending on the chosen value for δ, $\text{UB}_{\text{S-EKG}}$ varies from 65% (with δ equal to one) to arbitrarily close to 100% (for $\delta \to \infty$). (In Subsection 4.4.3 below, we reason further about this parameter.) Therefore, the value of δ can be used to trade-off the target utilization bound against the number of preemptions and migrations.

Although, the original description of S-EKG [AB08] does not use the concept of server, it is straightforward to equivalently map tasks to servers, which are then allocated time reserves as done in the generic algorithm. The rules to apply are as follows: (i) each task assigned to a dedicated processor is mapped to a server, which is then allocated exclusively the same dedicated processor; (ii) all non-split tasks that are assigned to one processor are mapped to a non-split server, which is then allocated to the same processor; (iii) each split task is mapped to a server that is split between the same processors that split task is assigned to.

With respect to the inflation of servers, under the original approach [AB08], each server is (safely, but inefficiently) inflated by the same amount $2 \cdot \alpha$ – in other words:

$$U^{infl:orig}_{S-EKG}[\tilde{P}_q] = U[\tilde{P}_q] + 2 \cdot \alpha \tag{4.5}$$

with α calculated according to Equation 4.4.

4.4.2 Discussion of NPS-F and its original theory

It is rather straightforward to formulate NPS-F as an instance of the generic algorithm. Indeed, NPS-F is based on the same concepts as the generic algorithm, and these concepts even have the same name, except for the servers, which were called *notional processors*, and gave the name to NPS-F. Furthermore, NPS-F's off-line procedure comprises exactly the same four steps.

Next we summarize the algorithms used by NPS-F, for each step of the off-line procedure. These are the algorithms that were used in the running example in Subsection 4.3 to illustrate the generic algorithm.

For the first step, the mapping of tasks to servers, NPS-F uses FF bin-packing heuristic. Inset (b) of Figure 4.1, in the Subsection 4.3, shows the task-to-server mapping obtained with NPS-F.

In the second step, the original paper on NPS-F used the following expression to inflate the capacity of each of the servers obtained in the first step:

$$U^{infl:orig}_{NPS-F}[\tilde{P}_q] = \frac{(\delta+1)\cdot U[\tilde{P}_q]}{U[\tilde{P}_q]+\delta} \tag{4.6}$$

where δ is an integer designer-set parameter, which is also used to set the length of the time slot like in S-EKG (see Equation 4.2).

The algorithm used by NPS-F to allocate processors to servers, in the third step, just iterates over the set of servers and assigns each server to the next processor that has yet some available capacity. If the processor's available capacity cannot accommodate the processing requirements of a server, the server is *split*. That is, the current processor's available capacity is allocated to partially fulfil the server's requirements, whereas the server's remaining requirements are fulfilled by the next processor.

Finally, the algorithm used by NPS-F in the fourth and last step is also straightforward. For each processor, it allocates one reserve per-server which uses it. Furthermore, the duration of each reserve is proportional to the processor capacity used by the corresponding server and is such that each server is periodic with a period equal to the time slot, S.

We end this subsection with the utilization bound determined by the original schedulability analysis:

$$\mathrm{UB}_{\mathrm{NPS-F}} = \frac{2\cdot\delta+1}{2\cdot\delta+2} \tag{4.7}$$

which ranges from 75% (for δ equal to one, which is the most preemption- and migration-light setting) to arbitrarily close to 100% (for $\delta \to \infty$). Note that, since δ controls the length of the time slot, S (see Equation 4.2), its value can be used to trade-off the target utilization bound against preemptions and migrations like in S-EKG.

4.4.3 Reasoning about the parameter δ

In all S-EKG and NPS-F publications, an explanation can be found, along the lines of "*δ is a designer-set integer parameter, which controls the frequency of migration of split tasks and can be used to trade off the target utilization bound against preemptions and migrations*". The purpose of this subsection is provide to the reader with the required intuition about this parameter. This is because many parameters of these algorithms are computed based on δ.

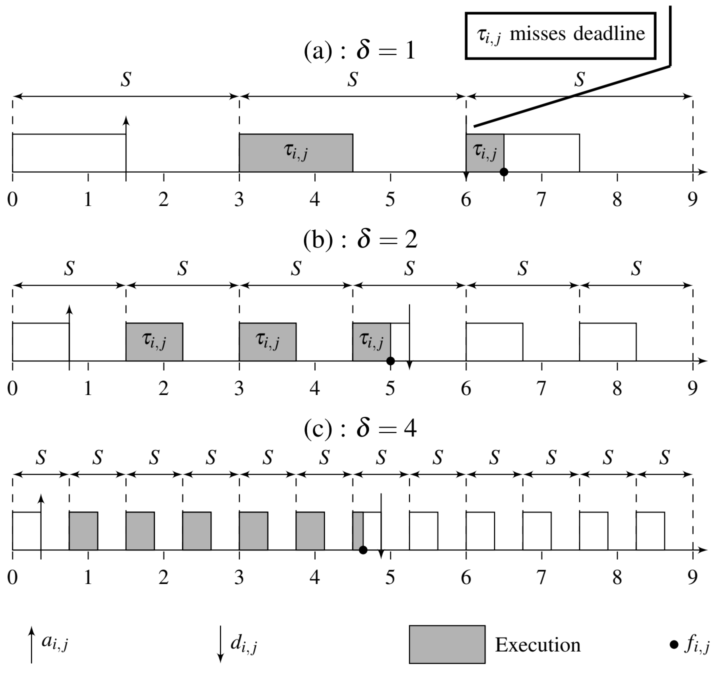


Figure 4.4: The intuition beyond δ parameter.

Originally, S-EKG and NPS-F compute S as follows: $S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i)$. This means that if the chosen δ is equal to one, this implies a S equal to $\min_{\tau_i \in \tau}(T_i)$, but if δ is equal to two, S is half of the $\min_{\tau_i \in \tau}(T_i)$, and so on.

The intuition beyond δ is illustrated by Figure 4.4. In the synthetic example presented in Figure 4.4, we assume that $\min_{\tau_i \in \tau}(T_i)$ is equal to three time units and the inflated utilization of the server is equal to 0.5 (the reserve for server $\tilde{P}_q$ is, graphically, represented by a white rectangle). Let us consider the sporadic j^{th} job of the task τ_i whose C_i is equal to two time units and whose T_i is equal to 4.5 time units. From a deadline miss point-of-view, the worst-case scenario for the job $\tau_{i,j}$ is when its arrival, $a_{i,j}$, happens on a time instant barely after the end of the associated reserve. The execution of the job $\tau_{i,j}$ is graphically represented by a gray rectangle labelled with the job identifier (except in inset (c), where the rectangles are not labelled because the identifier does not fit into the rectangle). As it can be seen from inset (a) of Figure 4.4, job $\tau_{i,j}$ misses its deadline; that is, $f_{i,j}$ is higher than $d_{i,j}$.

In contrast, in both insets (b) and (c), job $\tau_{i,j}$ meets its deadlines because the duration of the longest continuous interval that job $\tau_{i,j}$ is ready to be executed but cannot be executed decreases with the increase of the value of δ. A higher δ means a higher utilization bound, however, at cost of many preemptions and migrations due to reduction of the time slot length.

4.5 Instantaneous migration problem

In the original publications introducing the various slot-based task-splitting schemes under consideration [AB08, ABB08, BA09a, BA09b, BA11] a migrating server would migrate instantaneously

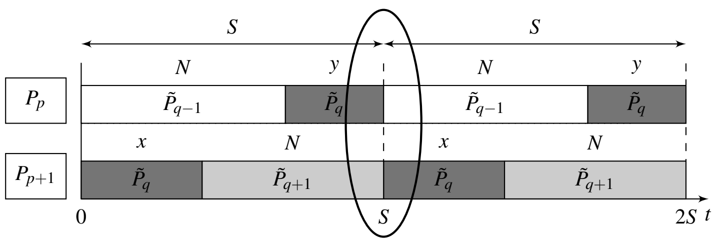


Figure 4.5: Illustration of the adjacent time slots that cause the instantaneous migration problem.

from one processor to another. For example, observe two consecutive time slots in Figure 4.5 where the reserves for the split server $\tilde{P}_q$ are temporally adjacent on processor P_p and processor P_{p+1}. Each server always selects the highest-priority task for executing, so, in practice, the executing task on processor P_p has to immediately resume its execution on its reserve on processor P_{p+1}. Due to many sources of unpredictability in a real-world operating system this level of precision is not possible. Consequently, this can prevent the dispatcher of processor P_{p+1} from selecting the task because processor P_p has not yet relinquished that task.

The result would be a waste of reserve time, possible endangering the schedulability of the system. This could be avoided if the reserve on processor P_{p+1} is available some time units later, by shifting the beginning of the time slot on processor P_{p+1} (see Figure 4.6).

In [SAT11] and [BA11] methods are proposed for calculating the optimal relative shifting for the time slots of processors that share split tasks in the context of the S-EKG and NPS-F, respectively. In the context of this dissertation, the method proposed by [BA11] was adopted, because is, in fact, generalizable for any slot-based task-splitting scheduling scheme. According to that formulation, for a split server $\tilde{P}_q$, which executes on two reserves $x[P_{p+1}]$ and $y[P_p]$, the optimal shifting is when the end of $y[P_p]$ is separated from the start of $x[P_{p+1}]$ by Ω (see Equation 4.8) time units. This means that the end of $x[P_{p+1}]$ is then also separated from the start of $y[P_p]$ by the same Ω time units. Note that, this does not impose any penalty for non-split servers.

$$\Omega = (S - y[P_p] - x[P_{p+1}])/2 \tag{4.8}$$

We denote this approach as the *staggered* version of the slot-based task-splitting algorithms.

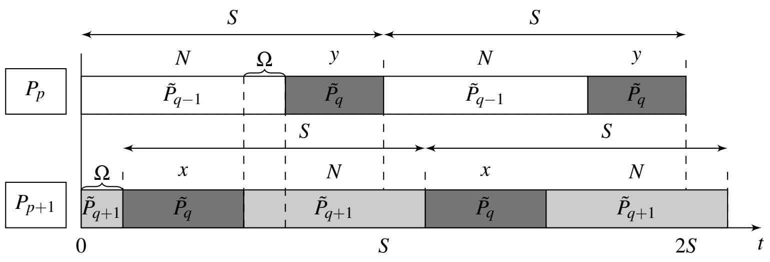


Figure 4.6: Illustration of the time slot shifting to solve the instantaneous migration problem. In spite of it is not shown in the figure there is no penalty for non-split servers; that is, the system supplies the same execution time to the non-split servers.

Note that, the Ω time units of shifting on each processor, $\Omega[P_p]$, are cumulative, and are computed as:

$$\begin{aligned}\Omega[P_1] &= 0 \\ \Omega[P_{p+1}] &= \Omega[P_p] + (S - y[P_p] - x[P_{p+1}])/2\end{aligned} \tag{4.9}$$

A collateral benefit from staggered approach is that the required inflation for split servers decreases. As explained in subsection 4.4.3, the reduction of the longest interval that a split server cannot execute its tasks (resulting from the use of the staggered offsets) helps with schedulability, which permits a reduction of the server's utilization inflation. In this case, split servers receive a *smoother* supply of processor time for executing their tasks.

Another weakness of the non-staggered (synchronized) time slots is that using synchronized time slots causes m task migrations at same instant. Assuming that all task migrations imply a cache miss event, this in turn implies m processors fetching data from the main memory. This will cause contention for the cache memory and also for the memory bus. We empirically evaluated these effects. For that purpose, we ran a set of experiments in a machine equipped with an Intel i7 processor [2] running Linux kernel 3.2.11 version (with the hyperthreading feature disabled). This processor is a 4-core with a L3 cache memory of 8 MB, shared by all cores.

The code of the task used to run this set of experiments is shown in Listing 4.1. This task receives two parameters: `cpu` and `time0`. `cpu` is the logical identifier of the processor that the task is assigned to. `time0` is used to define the initial absolute time instant of the experiment. There is an initial setup phase that comprises the following steps. The first one is setting the processor affinity of the task. Note that, with this feature the task is not allowed to migrate. The second step is to set the scheduling policy. The task is scheduled according the SCHED_FIFO policy (tasks scheduled according to such policy, in the Linux kernel, are given the processor for as long as they want it, subject only to the needs of higher-priority tasks) and it is the highest priority task in the system. The third step is to initialize an array of 32 MB of size. The last step is to put the task into the sleep mode until the absolute time of `time0` plus 20 seconds. After this initial phase, this task periodically reads (with a period of two seconds) a non-sequential array of 32 MB of size. It reads the array 5000 times and for each read it measures the time consumed for that read. At the end, it computes the average read time. Note that, given the reading pattern and the size of the array the processor has to constantly fetch task data from main memory.

We created seven task sets. The first task set is composed by only one task, running on processor P_1. Then, there is no contention neither for the L3 cache nor for the memory bus, because the task is the lone task in the system and has all resources available for it. The remaining six task sets are composed by four tasks each. For these task sets, we applied the partitioned approach; that is, each task was executed in only one processor and it was not allowed to migrate. In the second task set, we simulated the behaviour under synchronous time slots by setting the `time0`

[2] `http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html`

```
...
#define NSEC_PER_SEC    1000000000L
#define VEC_SIZE      4194304

int main(int argc, char* argv[])
{
  struct sched_param param;
  cpu_set_t cpu_set;
  struct timespec next_release,t,t1;
  unsigned int i,j,cpu;
  unsigned long long nj=0,x,sum_exec=0,release,time0;
  unsigned long long vector[VEC_SIZE];

  cpu = (unsigned int) atoi (argv[1]);
  time0=(unsigned long long)atoll(argv[2]);
  srand(SEED);

  CPU_ZERO(&cpu_set);
  CPU_SET(cpu, &cpu_set);
  sched_setaffinity(0, sizeof(cpu_set_t), &cpu_set);

  param.sched_priority = 1;
  sched_setscheduler(0,SCHED_FIFO,&param);

  for(i=0;i<VEC_SIZE;i++)
    vector[i]=i;
  for(i=0;i<VEC_SIZE;i++){
    j=rand()%VEC_SIZE;
    x = vector[i];
    vector[i]=vector[j];
    vector[j] = x;
  }

  release = time0 + 20 * NSEC_PER_SEC;
  next_release.tv_sec  = release / NSEC_PER_SEC;
  next_release.tv_nsec = release % NSEC_PER_SEC;
  clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &next_release,NULL);

  for(nj=0;nj < 5000;nj++){

    clock_gettime(CLOCK_MONOTONIC, &t);
    x = vector[0];
    for(i=1;i<VEC_SIZE;i++)
      x = vector[x];
    clock_gettime(CLOCK_MONOTONIC, &t1);
    sum_exec +=(t1.tv_sec * NSEC_PER_SEC + t1.tv_nsec)-(t.tv_sec * NSEC_PER_SEC + t.tv_nsec);

    release +=  2 * NSEC_PER_SEC;
    next_release.tv_sec  = release / NSEC_PER_SEC;
    next_release.tv_nsec = release % NSEC_PER_SEC;
    clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &next_release,NULL);
  }
  ...
  return 0;
}
```

Listing 4.1: The C language code of the task used to investigate the impact of the contention for shared cache and also for memory bus.

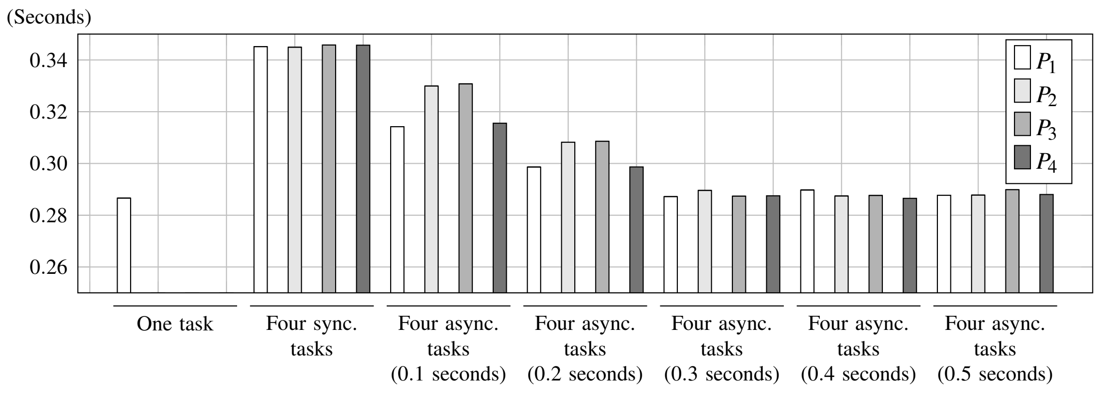


Figure 4.7: Experimental evaluation of the impact of the cache and bus memory contention on the reading time of an array. These experiments highlight the benefit of the staggered against the aligned versions.

equal to all tasks while in the third, fourth, fifth, sixth, and seventh we simulated the staggered time slots by setting `time0` with a shifting of 0.1, 0.2, 0.3, 0.4, and 0.5 seconds from the previous task, respectively.

Figure 4.7 plots the average array read time of each task. We use the average read time, because it is a measure of central tendency. As can be seen from that figure, the tasks being synchronised by their reads introduces a great penalty to the reading time. This is due to the fact that processors have to contend for the L3 cache and also for the memory bus. The impact of such effects decrease with asynchronous tasks. This becomes more clear when the shifting time is larger than the reading time, what happens for shifts larger than 0.3 seconds. For these task sets the average array read time is very close to the value measured when there is only one task. However, for a shifting of 0.1 and 0.2 seconds, tasks that execute on processors P_2 and P_3 present higher values than tasks that execute on processors P_1 and P_4. In fact, those tasks that execute on processors P_2 and P_3 do not execute alone, while tasks that execute processors P_1 and P_4 execute alone during some part of their reading time. In any case, the average array read time is smaller than for the synchronous task set.

4.6 Implementation of the slot-based task-splitting dispatcher

As mentioned before, the scheduling algorithms under consideration in this work require an off-line procedure that maps tasks to processors and a run-time task-dispatching algorithm. This section concerns with the run-time task-dispatching algorithm. The implementation of the task-dispatching algorithm has to be done inside the kernel of the operating system.

Before describing the implementation of the slot-based task-splitting dispatcher, let us present our reasons for choosing a PREEMPT-RT-patched Linux kernel version for implementing the dispatcher. In recent years, a kernel developer community (composed by Ingo Molnar, Thomas Gleixner and others) has been working on the PREEMPT-RT patch [PR12]. This patch (that aims to get a fully preemptible kernel) adds some real-time capabilities to the Linux kernel. The

PREEMPT-RT patch addresses sources of timing unpredictability in the mainline Linux kernel by making most of the Linux kernel preemptible, by implementing priority inheritance (to avoid priority inversion phenomena), and by converting interrupt handlers into preemptible kernel tasks. Further, this community is closely linked to the mainline Linux kernel, and indeed several features from the PREEMPT-RT patch have been introduced into the mainline Linux kernel. Other features remain in the PREEMPT-RT, because the mainline Linux kernel is a GPOS and some PREEMPT-RT features are not suitable for a GPOS. For instance, the high preemption level of the PREEMPT-RT patch decreases the scheduling latencies but increases the overhead caused by such events, because they happen more often.

However, it is a fact that real-time scheduling algorithms have been ignored by PREEMPT-RT community (even if we are not aware of the reasons for that). Therefore, we try to contribute by providing to PREEMPT-RT-patched Linux kernel version with suitable real-time scheduling algorithms for multiprocessor systems. Furthermore, our implementations aim to be used as a testbed framework for academic purposes. By the way, although the comparative evaluation between the Linux and the co-kernel approaches presented in [BLM$^+$08, BM10] claims better performance by the co-kernel approaches, none of that work claims that the Linux (mainline or PREEMPT-RT-patched) could not (be made to) support real-time systems.

During the work described in this dissertation, we, first, implemented S-EKG [SAT10b, SAT11] and subsequently we implemented both (S-EKG and NPS-F) scheduling algorithms [SBTA11] in a unified way; that is, using the same implementation for both. In all of the mentioned implementations, we used a scheduling class called `retas_sched_class` (ReTAS). The new scheduling class was added on top of the native Linux module hierarchy, thus making ReTAS the highest-priority scheduling class [ReT12]. However, such an approach cannot be employed in conjunction with the PREEMPT-RT patch, because important functionalities, such as timer interrupt handlers, needed by the ReTAS scheduling class, are implemented within the RT scheduling class and thus ReTAS cannot have a higher priority than the RT scheduling class implemented in the PREEMPT-RT patch. Thus, in this work, our focus is on the implementation where we added slot-based task-splitting scheduling algorithms to the RT scheduling class of the PREEMPT-RT-patched Linux 3.2.11-rt20 kernel version [SPT12]. Therefore, we added to the RT scheduling class support for scheduling algorithms that are suitable for real-time systems running in multiprocessor platforms.

4.6.1 Scheduler implementation

Recall that, the run-time task-dispatching algorithm of the scheduling algorithms under consideration, divides the time into equal-duration time slots. Each time slot is composed by at most three time reserves. The time slot is instantiated in a per-processor basis; that is, each processor divides the time into time reserves. Each processor reserve is allocated to only one server. Split servers have reserves assigned to different processors while non-split servers have one reserve. Each server has one or more tasks assigned to it that are scheduled according to the EDF.

From this description, we can view the task-dispatching algorithm as a two-level hierarchical algorithm. The top-level scheduling algorithm allocates processors to servers for the duration of

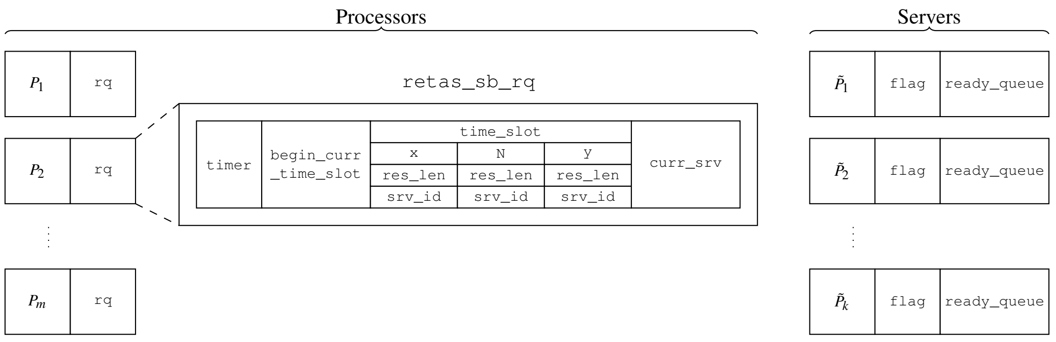


Figure 4.8: The required data used to implement the task-dispatching algorithm and its organization.

the respective server reserve(s) and ensures that for each server, at any time, there is at most one processor allocated to it. The low-level scheduling algorithm employs the EDF policy to schedule each server's tasks.

Before describing the implementation, let us highlight how the scheduler works in a multiprocessor system. As mentioned before, the scheduler is implemented through the `schedule` function. This function is invoked directly by the kernel code. In a multiprocessor system, this means that this kernel code is being executed on some processor. For instance, consider a task that is executing on processor P_p and completes. Upon the completion of the task, the `exit` system call is invoked that performs a set of clean-up operations for freeing the task resources and in turn it invokes the `schedule` function, which executes on processor P_p. Therefore, the scheduler works on per-processor basis.

As a rule-of-thumb, the implementation of the task-dispatching algorithm should avoid interactions across processors; that is, it should isolate the scheduling decisions on one processor from activities and decisions on other processors. Figure 4.8 shows the required data for implementing the task-dispatching algorithm. As can be seen from Figure 4.8, the data is divided into two parts: one part associated to processors and another associated to servers. For each processor, we added to the `rq` data structure a new data structure called `retas_sb_rq`. Next, we discuss about our implementation approach with respect to those data elements and our choice of Linux 3.2.11-rt20 for the implementation.

Each server is an instance of a new data structure called `server`, which stores the following information: (i) a field that uniquely identifies a server; (ii) an atomic variable that exposes the server state; and finally, (iii) a ready-queue, which is a priority queue wherein ready tasks are stored.

The Linux kernel defines one ready-queue per-processor. Such an arrangement is *natural* under partitioning, but potentially inefficient in the presence of migrating tasks. If tasks are stored locally on each processor (on a respective per-processor ready-queue), a task migration requires locking the ready-queues of both the origin and the destination processor. This may introduce big overheads, especially if the locking has to be serialised.

In the case of the slot-based task-splitting scheduling algorithms, where migrations involve the entire server, this would typically imply moving multiple tasks (hence, even higher overheads). Given that the frequency of migration may be high, it is obviously not the best approach. We opt instead for one ready-queue *per-server* (see Figure 4.8). Under this approach, all ready tasks assigned to a server are always stored on (i.e. inserted to/dequeued from) the same respective (per-server) ready-queue. Then, when a server migrates (i.e. with all its tasks) from processor P_p to processor P_{p+1}, we simply change the ready-queues used by processors P_p and P_{p+1}.

We implement server ready-queues as red-black trees. This data structure was deemed appropriate because: (i) the Linux kernel already implements red-black trees; (ii) they are balanced binary trees whose nodes are sorted by a key (which, in our implementation is the absolute deadline of each job); (iii) most operations take $O(\log n)$ time (good for performance if they are frequent).

In order to associate a task to a server all that is needed is a link between each task and its corresponding server. Under Linux, a *task* is an instance of a program in execution and the kernel uses one instance of a data structure (of type `struct task_struct`) for each task. Therefore, we simply added a field to that data structure, to hold the (unique) logical identifier of the respective server.

The top-level scheduler is responsible for the management of time reserves. A timer is used to trigger scheduling decisions at the beginning of each reserve. The Linux kernel offers high resolution timers with nanosecond resolution and timers are set on a per-processor basis. Each processor is configured with its time slot composition. The variable `begin_curr_time_slot` holds (as suggested by its name) the beginning of the current time slot and it is incremented by S (the time slot length). Observe that no synchronization mechanism is required for updates to this variable. The time slot composition is defined by an array of 2-tuples <`res_len`, `srv_id`> (see Figure 4.8). Each element of this array maps a reserve of length, `res_len`, to the respective server, `srv_id`. For optimization purposes, there exists a per-processor variable `curr_srv` that points to the current server. In other words, whenever the timer expires, at the beginning of one reserve, `curr_srv` variable is updated with the correspondent server address.

The low-level scheduler picks the highest-priority task from the server pointed to by `curr_-srv`. However, if the current server is a split server there is no guarantee that server is not also the current server of another processor. At design stage, we solved this problem in principle with the staggered time slot approach, but in practice, there is no absolute guarantee that this cannot happen.

To highlight the synchronization mechanism described below, let us assume a synchronized time slot scheme, which brings with it the instantaneous migration problem. In a such scheme, whenever a split server consumes its reserve on processor P_p, whichever task was executing at the time has to *immediately* resume execution on another reserve on processor P_{p+1}. However, due to many sources of unpredictability, the scheduler of processor P_{p+1} can be prevented from selecting the task in consideration for execution because processor P_p has not yet relinquished (the ready-queue associated with) that task. Two solutions could be used to solve this issue. One solution could be for processor P_{p+1} to send an IPI to P_p to relinquish (the ready-queue) of server $\tilde{P}_q$. Another could be for P_{p+1} to set up a timer x time units in the future to force the invocation of

```
static void
enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
  struct sched_rt_entity *rt_se = &p->rt;
  if(retas_policy(p->policy)){
    _enqueue_task_retas(rq,p);
    return;
  }
  ...
}
```

Listing 4.2: Changes to the `enqueue_task_rt` function.

its scheduler. We chose the latter option for two reasons: (i) we know that if a scheduler has not yet relinquished the server ready-queue it was because something is preventing it from doing so (e.g. the execution of an ISR); and, (ii) using IPIs in this manner introduces a dependency between processors, because the scheduling decision on one processor is dependent upon the reception of an interrupt sent by other processor.

In our implementation, to serialise the access to the ready-queue of a server, an atomic variable (called `flag`) per-server is used to state if a task of that server is being executed by a processor or not. Whenever a processor scheduler is invoked, it tries to increment by one that variable (of type `atomic_t`) using the `atomic_add_unless` kernel function. The `atomic_t` data type defined in the Linux kernel is simply an integer with a set of operations guaranteed to be atomic without any need for explicit locking. The increment operation can only succeed if the value of the `flag` is not one (i.e. if it is zero); otherwise, it fails. In the first case (success), by atomically incrementing the `flag`, the scheduler *locks* the server to that processor. In the second case (failure), the scheduler cannot pick any task from this server; it then sets up a timer to expire some time later, which will trigger anew the invocation of the scheduler.

4.6.2 Adding support for slot-based task-splitting scheduling algorithms to the RT scheduling class

Apart from the required code to manipulate servers (enqueue and dequeue of ReTAS tasks as well as getting the task with the earliest absolute deadline) and the time slot infrastructure, incorporating ReTAS into the RT scheduling class implies a set of modifications to some functions of the `rt_-sched_class` scheduling class. Those functions are `enqueue_task_rt`, `dequeue_task_-rt`, `check_preempt_curr_rt`, and `pick_next_task_rt`.

The function `enqueue_task_rt` (see Listing 4.2) is called whenever a RT task enters into a ready state. It receives two pointers, one for the ready-queue of the processor that is running this code, `rq`, and another to the task that is becoming ready, `p`. If the ready task is a ReTAS task (in which cases, it is also a RT task, with a priority level, but it is scheduled according to the `SCHED_NPS_F` or `SCHED_S_EKG` scheduling policies), then it is enqueued into the respective server ready-queue. When a RT task is no longer ready, then the `dequeue_task_rt` function is called to remove the task from the respective server ready-queue.

```
static void
check_preempt_curr_rt(struct rq *rq, struct task_struct *p, int flags)
{
 if (p->prio <= rq->curr->prio) {
   if(retas_policy(p->policy)){
     if(_check_preempt_curr_retas(rq)){
       resched_task(rq->curr);
       return;
     }
   }
 }
 ...
}
```

Listing 4.3: Changes to the `check_preempt_curr_rt` function.

As the name suggests, the `check_preempt_curr_rt` function (see Listing 4.3) checks whether the currently running task must be preempted or not (e.g. when a RT task wakes up). It receives two pointers, one for the processor ready-queue that is running this code, `rq`, and another to the woken up task, `p`. If the priority of the woken up task is higher than or equal to (lower `prio` values mean higher priority) the currently executing task (pointed to by `rq->curr`), it checks if `p` is a ReTAS task and if the current reserve is mapped to the server that task `p` is assigned to. If that is the case, then the currently running task is marked for preemption.

The `pick_next_task_rt` function also needs a small modification. This function selects the task to be executed by the current processor and is called by the scheduler whenever the currently executing task is marked to be preempted or finishes its execution. It first gets the highest-priority RT task, then it gets the highest-priority ReTAS task. Afterwards, it selects the highest-priority task between them (if any exists), otherwise it returns `NULL`, which forces the scheduler to inquire the CFS scheduling module.

So far, we have presented the generic slot-based scheduling algorithm and addressed and discussed all relevant details related to the operating system as well as the implementation of slot-based algorithms. In the next section, we identify all run-time overheads incurred by such

```
static struct task_struct *
pick_next_task_rt(struct rq *rq)
 {
  struct task_struct *p = _pick_next_task_rt(rq);
  struct task_struct *t = _pick_next_task_retas(rq);
  if(t){
    if(p){
      if(t->prio <= p->prio){
        return t;
      }
    }else{
      return t;
    }
  }
  ....
  return p;
}
```

Listing 4.4: Changes to the `pick_next_task_rt` function.

algorithms.

4.7 Scheduling overheads

In order to carry out an overhead-aware schedulability analysis (presented in the next chapter), we first need to identify the overheads that may be incurred at run-time because of the mechanisms used in the implementation of the scheduling algorithms. In this subsection, we provide an overview of the overheads that may arise in an implementation of a slot-based task-splitting scheduling algorithm. This overview is based on our implementation [SPT12] of S-EKG and NPS-F in the Linux kernel for the x86-64 architecture described above.

The overheads that a system may incur because of a scheduling algorithm are related to the following five mechanisms: (i) interrupts; (ii) timers; (iii) ready queues; (iv) context switching; and (v) caches. We examine the overheads of each mechanism in turn.

Most real-time systems interact with their environment and use interrupts whenever they need to react to environment events. We assume that the interrupt handlers, or ISRs, are implemented as tasks, as supported in the PREEMPT-RT Linux kernel [PR12]. Nevertheless, the occurrence of an interrupt suspends the execution of the currently running task to release a task that will service this interrupt. Furthermore, depending on the deadline of the released task, it may cause a preemption of the currently running task. A special kind of interrupt is the inter-processor interrupt, IPI. As its name suggests, these interrupts are generated by one processor and handled on another, and may be used by a processor to notify another of the occurrence of events. The processing of an IPI by the target processor is similar to that of an interrupt generated by the environment. Our algorithms only use the IPI in the implementation of split servers, more specifically, when a job whose priority is higher than that of all ready jobs of its server, arrives on a processor at a time instant that falls within the reserve of that server in the other processor. In this case, the newly arrived job should immediately start execution in the server's reserve on the other processor. We denote the delay incurred by the use of IPI in the dispatching of a task the *IPI Latency*, $IpiL$.

Timers are a per-processor mechanism of the Linux kernel designed for scheduling computations some time in the future. Our algorithms use timers to release tasks and also to trigger server-switches at the end of each time reserve. Timers are implemented using a priority queue and interrupts generated by some timer/counter device. They incur overheads related to the handling of these interrupts as well. Timer interrupts are different from other interrupts in that they are not handled by separate tasks, but immediately upon the occurrence of the interrupt. Thus, the expiration of a timer suspends the execution of the current task on that processor. Another *imperfection* associated with timers is that they do not measure time intervals in a precise way. We denote the delay incurred in the release of periodic tasks because of these imperfections the *Release Jitter*, $RelJ$.

The Linux kernel keeps the released tasks that are ready to run in queues, known as ready-queues. Therefore, when a task is released, the kernel moves the task to a ready-queue, and the dispatcher is invoked to select the next task to run, which may be either the task that was running before the release of the task, the released task or any some task that is ready to run. In the case of the slot-based task-splitting algorithms considered, all these data structures are either private to some processor or shared by two processors. Nevertheless, the release of a task requires some processing, which we call the *Release Overhead*, *RelO*.

A context switch (or task switch) occurs whenever the dispatcher decides to change the running task on a processor. This entails saving the state of the processor to some operating system data structure associated with the task being evicted, and restoring the state of the processor to the contents of the corresponding data structure associated with the task that was allocated to the processor. We use the *Context switch Overhead*, *CtswO*, to account for this overhead.

The worst-case execution time of a task is usually computed assuming that the task is executed without being preempted. However, when a task is preempted, it may incur additional costs when it is resumed because the cache lines with its data may have been evicted by other tasks and need to be fetched again from the main memory or from higher cache levels. Likewise, migrating one task from one processor to another requires the destination processor to fetch anew the cache footprint of the task. These costs are known as Cache-related Preemption and Migration Delays (CPMD). To incorporate the CPMD, we pessimistically assume that every preemption incurs the worst-case *CPMD Overhead*, *CpmdO*, cost. Furthermore, we do not distinguish between job preemption and job migration events. This simplification is not as pessimistic as it may seem because there is evidence [BBA10a, BBA10b] to suggest that, in a heavily loaded system, the CPMD costs of preemptions and migrations can be similar.

Although in this subsection we have discriminated the different sources of overheads associated with slot-based task-splitting scheduling algorithms, in the analysis devised in the next chapter, we sometimes lump in a single parameter, overheads of different sources that occur together in sequence. The reasons for this are two-fold. First, this leads to shorter expressions. Second, it simplifies the experimental measurement of the overheads and often leads to more precise experimental estimates of these overheads.

4.8 Summary

In this chapter, we presented a generic description of slot-based task-splitting scheduling algorithms and we also defined the terminology that is used in the remainder of this dissertation. We have shown how to correlate the particular algorithms with the generic one. We also clarified the impact of one of the most important parameters for slot-based task-splitting scheduling algorithms, called δ. We have identified one practical problem (the instantaneous migration problem) of slot-based task-splitting scheduling scheduling algorithms and also how it can be solved. Since the Linux kernel does not support any suitable mechanism for frequent task migrations, we detailed how to solve that issue. Moreover, since neither the mainline Linux kernel nor the PREEMPT-RT patch support any scheduling algorithm suitable for real-time multiprocessor systems, we described how we incorporated support for slot-based task-splitting scheduling in the PREEMPT-RT-patched Linux kernel. Note that, we follow a different approach to other related works; that is, we incorporate the slot-based task-splitting scheduling algorithms into the RT scheduling class itself. Finally, we identified and described the run-time overheads incurred by the slot-based task-splitting scheduling algorithms running in Linux.

Chapter 5

Slot-based task-splitting overhead-aware schedulability analysis

5.1 Introduction

Real-time multiprocessor scheduling has seen, in recent years, the flourishing of semi-partitioned scheduling algorithms. This category of scheduling schemes combines elements of partitioned and migrative scheduling for the purposes of achieving efficient utilization of the system's processing resources with strong schedulability guarantees and with low dispatching overheads. The sub-class of slot-based task-splitting scheduling algorithms, in particular, offers very good trade-offs between schedulability guarantees (in the form of high utilization bounds) and the number of preemptions/migrations involved. However, so far there did not exist unified scheduling theory for such algorithms; each one was formulated with its own accompanying analysis.

In this chapter, we formulate a unified schedulability theory for slot-based task-splitting applicable to scheduling algorithms under consideration in this dissertation, S-EKG and NPS-F. This new theory is based on exact schedulability tests, thus also overcoming many sources of pessimism in existing analysis. Furthermore, as a response to the fact that many unrealistic assumptions, present in the original theory, tend to undermine the theoretical potential of such scheduling schemes, we identified and modelled into the new analysis all overheads incurred by the algorithms in consideration. The outcome is a new overhead-aware schedulability analysis that permits increased efficiency and reliability. We provide the basic concepts that are used by the new theory developed, then, we present the new schedulability analysis. We end this chapter with a description how to implement the new theory.

5.2 Basis for the overhead-aware schedulability analysis

The scheduling algorithms under consideration in this dissertation assume that servers schedule their tasks according to the EDF scheduling algorithm. For preemptive uniprocessor systems, any periodic task set with implicit deadlines is schedulable under EDF scheduling policy if the condition $\sum_{\tau_i \in \tau} u_i \leq 1.0$ holds [LL73] and it is considered to be optimal [Der74] among all uniprocessor scheduling algorithms. In other words, if a task set is schedulable (no job miss deadlines) under any scheduling algorithm on a preemptive uniprocessor system is also under EDF.

The EDF schedulability analysis for sporadic task sets with arbitrary deadlines is based on the concepts of *processor demand* and the *demand bound function* (dbf) [BMR90]. The dbf gives a upper bound (over every possible time interval $[t_0,\ t_0+t)$ of length t) on the processor demand, defined as the aggregate execution requirement of all jobs released at time t_0 or later and whose absolute deadlines lie before time t_0+t.

In the context of pure partitioned scheduling with arbitrary deadline task sets (meaning that D_i is different of T_i for any tasks) the execution demand of a set of tasks assigned to processor P_p, $\tau[P_p]$, in a time interval of duration t is computed as:

$$\mathrm{dbf}(\tau[P_p],t) = \sum_{i \in \tau[P_p]} \max\left(0, \left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{5.1}$$

Similarly (in the context of pure partitioned scheduling), the *supply bound function* (sbf) gives a lower bound on the amount of execution time supplied to the task set assigned to processor P_p over a time interval of length t without any constraint and is computed as follows:

$$\mathrm{sbf}(\tau[P_p],t) = t \tag{5.2}$$

Intuitively, a partitioned real-time system is schedulable if following inequality holds:

$$\mathrm{dbf}(\tau[P_p],t) \leq \mathrm{sbf}(\tau[P_p],t) \tag{5.3}$$

on every processor P_p and for every interval length $t > 0$.

Consider a task set composed by only one task with the following parameters: worst-case execution time, C_i, equal to two; minimal inter-arrival time T_i equal to seven; and deadline, D_i, equal to five. Thus, it is an arbitrary deadline task. Inset (a) of Figure 5.1 shows the execution demand of that task in an interval of 18 time units against the processing capacity supplied by the system (assuming a uniprocessor system). As it is expected, the execution demand until the first deadline (time instant five) is two time units, increases to four on the second deadline (time instant 12) and so on.

Let us assume that the release jitter is non-negligible, which means that the release of such a task is delayed by $RelJ_i$. In practice, this means that the task is only ready for execution $RelJ_i$ time units after its arrival time. In other words, such task has to complete within $D_i - RelJ_i$ time units. In order to consider such overhead, the $\mathrm{dbf}(\tau[P_p],t)$ is changed to [Spu96, ZB08]:

$$\mathrm{dbf}(\tau[P_p],t) = \sum_{i \in \tau[P_p]} \max\left(0, \left\lfloor \frac{t-(D_i - RelJ_i)}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{5.4}$$

Inset (b) of Figure 5.1 plots the execution demand considering that $RelJ_i$ is equal to one. As it can be observed, the processing demand by the task is shifted one time unit earlier due to the reduction of the execution time window.

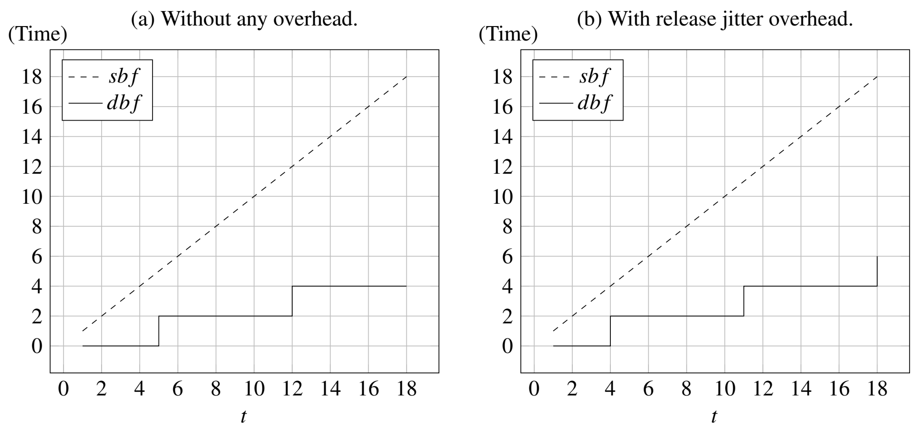


Figure 5.1: Illustration of the execution demand of a task against the processing supplied by the system.

Let us now consider that such task executes only inside of a reserve of length Res^{len} and periodically available every S time units. The time supplied for the execution of such a task on processor P_p over a time interval of length t is lower-bounded by [SBAT11]:

$$\text{sbf}(S, Res^{len}, t) = \left\lfloor \frac{t}{S} \right\rfloor \cdot Res^{len} + \max\left(0, \left(t - \left\lfloor \frac{t}{S} \right\rfloor \cdot S\right) - \left(S - Res^{len}\right)\right) \tag{5.5}$$

Consider that Res^{len} is equal to three and S is equal to five. Inset (a) of Figure 5.2 plots the time supplied (sbf) against the time demanding (dbf) in a time interval of 18 time units. Inset (b) of Figure 5.2 illustrates graphically the relation between the time supplied and the time demanding. As it is intuitive, the worst-case phasing for task execution occurs when the task arrives an infinitesimal amount of time after the respective reserve has been finished and the worst-case phasing for supplying is when the arrival time of a task coincides with the beginning of the reserve period.

The first term of Equation 5.5 ($\lfloor \frac{t}{S} \rfloor \cdot Res^{len}$) computes the amount of time supplied over the $\lfloor \frac{t}{S} \rfloor$ periods integrally contained within the time interval under consideration. In that case of $t = 18$, $\lfloor \frac{18}{5} \rfloor \cdot 3 = 3 \cdot 3 = 9$ time units. The second term ($\max(0, (t - \lfloor \frac{t}{S} \rfloor \cdot S) - (S - Res^{len}))$) computes the time supplied over the remaining *tail* (the last, incomplete period); in our example, during this tail, the time supplies $\max(0, (18 - \lfloor \frac{18}{5} \rfloor \cdot 5) - (5 - 3)) = \max(0, 18 - 3 \cdot 5 - 2) = \max(0, 18 - 15 - 2) = \max(0, 1) = 1$ time unit. Therefore, this system supplies $9 + 1 = 10$ time units in a time interval of 18 time units for executing such task.

Next, we use demand- and supply-based functions to derive the new schedulability analysis for slot-based task splitting scheduling algorithms.

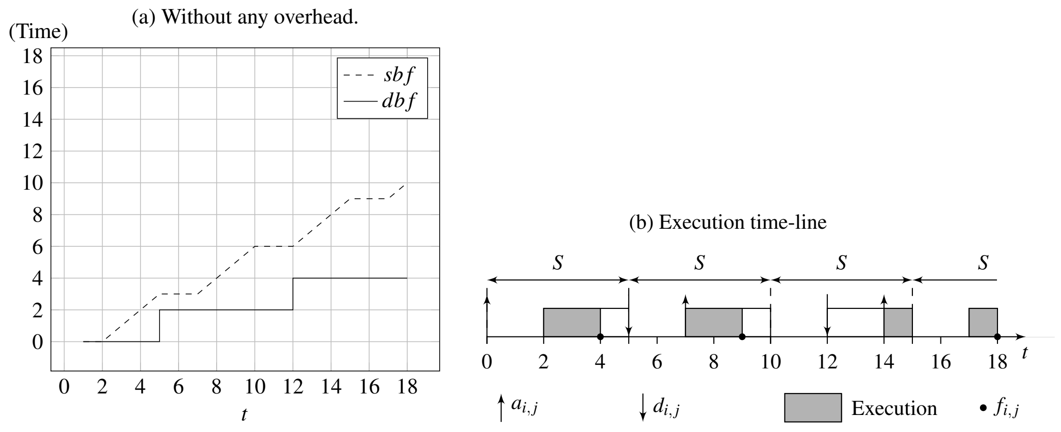


Figure 5.2: Illustration of the execution demand of a task against the processing supplied by the system considering time reserves.

5.3 New demand-based and overhead-aware schedulability analysis

The original schedulability analysis for slot-based task-splitting scheduling algorithms was based on utilization. While this simplifies the derivation of utilization bounds, it also entails pessimism. In [ABB08], the move towards processor-demand based analysis was not carried out in a way that would preserve the most useful theoretical properties (namely, the utilization bound) of previous work (S-EKG). Therefore, in [SBAT11], the authors present a schedulability analysis based on processor demand specific for the S-EKG scheduling algorithm.

In this section, a new schedulability analysis, based on processor demand, is introduced that can be applied to both S-EKG and NPS-F [SBT^{+}13]. This new schedulability analysis supersedes all previous utilization-based analyses. Further, it defines new schedulability tests that incorporate all real-world overheads incurred by one implementation of the S-EKG and NPS-F algorithms [SPT12].

The schedulability analysis that we develop in this section has two stages, which correspond to the two main stages of the task-to-processor mapping algorithm presented in Section 4.3. In the first stage, the analysis focuses on the schedulability of the tasks assigned to each server, assuming that each server is executed in isolation on a processor. The second stage examines whether there is enough capacity to accommodate all servers in the system.

We present each stage of the new demand-based overhead-aware schedulability analysis in its own subsection.

5.3.1 New demand-based schedulability test for mapping tasks to servers

In this subsection, we derive a schedulability test for the tasks mapped to a server based on demand-bound functions taking into account the overheads described in the Section 4.7. This leads to a new task-to-server mapping algorithm. For the purpose of the mapping of tasks to servers, we consider that a server is allocated a processor exclusively, i.e. it runs on a single processor that it does not share with any other server. Hence, we treat each server as a uniprocessor system.

Our analysis is based on the concept of demand-bound functions, which specifies a upper bound on the aggregate execution requirements of all jobs (of $\tau[\tilde{P}_q]$) over any possible interval of length t. Therefore the demand-based schedulability test for a server $\tilde{P}_q$ is given by:

$$\text{dbf}^{\text{part}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{5.6}$$

We use the text “part”, which stems from “partitioned”, as a superscript of all the dbfs of this stage to distinguish them from functions of the second stage.

Ignoring all overheads and assuming sporadic task sets with arbitrary deadlines, the $\text{dbf}^{\text{part}}(\tilde{P}_q, t)$ can be computed as:

$$\text{dbf}^{\text{part}}(\tilde{P}_q, t) = \text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{5.7}$$

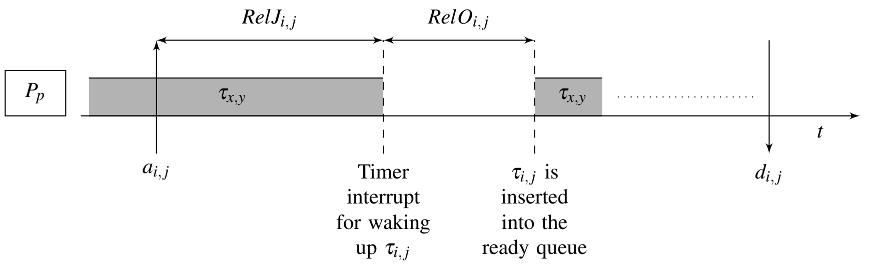
Next, we proceed by incorporating each source of overhead into the new overhead-aware schedulability analysis, one at a time. First, we consider the overheads caused by the release of tasks. We assume that all tasks are periodic, because it corresponds to the worst-case. For periodic tasks we need to take into account not only the release overhead, but also the release jitter caused by timers. Therefore, the effects of timers and task release will be considered together. Next, we consider the effects of context switching and CPMD. Finally, we incorporate the effect of interrupts other than those caused by timers.

Although other mechanisms may be used to release tasks, for sake of clarity, we assume that the task release is done by the use of timers. Therefore, the release of periodic tasks is affected by two of the overheads discussed in Section 4.7: the release overhead and the release jitter. Figure 5.3 graphically shows these two overheads for job $\tau_{i,j}$. (In all figures, the execution of a job is graphically represented by a rectangle labelled with the job identifier.) As illustrated, the effects of these two overheads are different. Whereas both overheads, the release jitter of job $\tau_{i,j}$, $RelJ_{i,j}$, and the release overhead of job $\tau_{i,j}$, $RelO_{i,j}$, reduce the amount of time that job $\tau_{i,j}$ has to complete its execution, only the release overhead actually requires processing time. Thus, we model the effect of these two overheads differently.

Let $RelJ$ and $RelO$ be the upper bounds on the release latency and on the release overhead, respectively. As shown in Figure 5.3, the release jitter decreases the amount of time available to complete a task, i.e., in the worst case, τ_i has $D_i - RelJ$ time units to complete. Therefore, we modify the $\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t)$ to:

$$\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - (D_i - RelJ)}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{5.8}$$

Concerning the release overhead, one way of modelling it could be by increasing the execution



$RelJ_{i,j}$
$RelO_{i,j}$
P_p
$\tau_{x,y}$
$\tau_{x,y}$
t
$a_{i,j}$
Timer interrupt for waking up $\tau_{i,j}$
$\tau_{i,j}$ is inserted into the ready queue
$d_{i,j}$

Figure 5.3: Illustration of release jitter and release overhead.

demand of a task accordingly. However, that approach does not work properly when multiple tasks are released too close together in time. The reason is that the release overhead contributes *immediately* to the processor demand – meaning that to model the processor demand correctly, it should be increased by $RelO$ time units at the time of the release, not at the deadline of the task released. Therefore, we instead model the release overhead as higher-priority interfering workload (as it is in reality). This way, we may compute the execution demand for releasing all jobs of $\tau[\tilde{P}_q]$ in a time interval $[0, t)$ as:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q],t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \cdot RelO \tag{5.9}$$

Modifying accordingly $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$, we get:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) = \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q],t) + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot C_i \end{aligned} \tag{5.10}$$

We now consider the context switching overhead, which is common to all schedulers. Every job causes *at most* two context switches: when it is released and when it completes – but not every job release causes a context switch. Therefore the number of context switches over a time interval of length t is upper bounded by twice the number of job releases during that interval. Let $CtswO$ be a upper bound on the context-switch overhead. We amend the derivation of the $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$, by increasing the execution demand of each job by twice $CtswO$, to:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) = \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q],t) + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot (C_i + 2 \cdot CtswO) \end{aligned} \tag{5.11}$$

In order to incorporate the cache-related overheads, i.e. the CPMD, we, pessimistically, assume that every preemption incurs the worst-case CPMD cost, $CpmdO$. Furthermore, we compute a upper bound on the number of preemptions for server $\tilde{P}_q$ in a time interval of length t as:

$$\mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q,t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \tag{5.12}$$

That is, we assume that every task that may be released in a time interval of length t, causes a preemption. Thus, the cumulative cost of CPMD over one interval of length t is:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) = \mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q,t) \cdot CpmdO \tag{5.13}$$

Intuitively, this increases the server execution demand. We therefore amend the expression of the $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q,t)$ to:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) \tag{5.14}$$

In contrast with the other overheads, and also with a recent approach [LAMD13], the cache-related overheads are not assigned to a particular task. Indeed, the jobs of some tasks may never be preempted, whereas the jobs of other tasks may be preempted several times. This is the reason why we do not incorporate the CMPD overheads in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$.

Finally, we consider the interrupt overheads. We assume that interrupt service tasks have higher priority than *normal* tasks. Thus, we model each sporadic interrupt as a task with worst-case execution time equal to C_i^{Int}, minimum inter-arrival time equal to T_i^{Int} and zero laxity ($C_i^{Int} = D_i^{Int}$). Periodic interrupts are also modelled as zero-laxity tasks, but T_i^{Int} represents their period and they are also characterized by a release jitter J_i^{Int}, which accounts for deviations from strict periodicity. For sporadic interrupts, we let J_i^{Int} equal to zero, since any variability in their arrival pattern is already accounted for by T_i^{Int}. Thus the interrupt execution demand for n^{Int} interrupts is then given by:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q,t) = \sum_{i=1}^{n^{Int}} \max\left(0, \left\lfloor \frac{t - D_i^{Int} + J_i^{Int}}{T_i^{Int}} \right\rfloor + 1\right) \cdot C_i^{Int} \tag{5.15}$$

Intuitively, the interrupt overhead increases the execution demand of a server. Thus, the $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q,t)$, incorporating all the overheads, becomes:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q,t) \tag{5.16}$$

Equation 5.16 can be used in a new schedulability test by the algorithm that maps tasks to servers. Algorithm 5.1 shows the pseudo-code of this algorithm. The algorithm iterates over the set of all tasks and, for each task τ_i, it checks whether it fits in one of the opened servers (subject to the constraints of the bin-packing heuristics used, e.g., NF or FF). For each server $\tilde{P}_q$ checked (q being the server index), it provisionally adds task τ_i to it, then it computes the length of the testing time interval t (computed as twice the least-common multiple of the T_i of tasks in $\tau[\tilde{P}_q]$) and finally, it applies the new schedulability test, by invoking the `dbf_part_check` function. If the test succeeds for some server $\tilde{P}_q$, then task τ_i is permanently mapped to it, otherwise, a new server is opened and task τ_i is added to it. The task set is considered unschedulable whenever the schedulability test fails for a server with only one task.

This algorithm applies only to NPS-F. In the case of S-EKG for reasons that will be explained later, the task to server mapping and the server to processor assignment are performed in a single step using the algorithm that is outlined in Subsection 5.4.2.1.

Algorithm 5.1: Pseudo-code of the new task-to-server mapping algorithm.

```
Input: set of $n$ tasks $\tau_i$, with $1 \leq i \leq n$
Output: set of $k$ servers, with $k \geq 0$ ($k = 0$ means failure)

$k \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
  $scheduled \leftarrow 0$
  for $q \leftarrow 1$ to $k$ do
    add_task_to_server($\tau_i, \tilde{P}_q$)
    $t \leftarrow 2 \cdot$ lcm_T($\tilde{P}_q$)
    if dbf_part_check($\tilde{P}_q, t$) then
      $scheduled \leftarrow 1$
      break
    else
      remove_task_from_server($\tau_i, \tilde{P}_q$)
    end if
  end for
  if $scheduled = 0$ then
    $k \leftarrow k + 1$ {add a new server}
    add_task_to_server($\tau_i, \tilde{P}_k$)
    $t \leftarrow 2 \cdot$ lcm_T($\tilde{P}_q$)
    if not dbf_part_check($\tilde{P}_k, t$) then
      $k \leftarrow 0$
      break {failure}
    end if
  end if
end for
```

To summarise, in this subsection we have developed a new overhead-aware analysis for schedulability testing in the task-to-server mapping stage. However, this test considers each server in isolation and it does not encompass all the scheduling overheads that may be incurred by servers when they share a processor with other servers. In the next subsection, we develop a new schedulability analysis for the processor-to-server allocation step.

5.3.2 New demand-based schedulability test for assigning servers to processors

To fully model all the overheads incurred by the use of periodic reserves, it is necessary to assign each server to one or more processors. Precisely modelling the impact of these overheads allows us to determine the processing capacity requirements of each server. In turn, this allows us to test whether or not all servers can be accommodated on the m physical processors.

With the server-to-processor assignment described in Section 4.3, non-split servers are allocated just one processor reserve whereas split-servers must be allocated two reserves. Because, each type of server incurs different overheads, we deal with each type of server separately.

5.3.2.1 Non-split servers

The approach we follow to check the schedulability of a server is to verify that the execution demand by all jobs assigned to a server (computed using the dbf) does not exceed the amount of time (computed using the sbf) that the system can provide for their execution, for every time interval of length t. Formally, we can express this schedulability test as:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}(\tilde{P}_q,t) \leq \mathrm{sbf}^{\mathrm{sb:non-split}}(\tilde{P}_q,t), \forall t > 0 \tag{5.17}$$

We use the superscript "sb" (an abbreviation for "slot-based") to distinguish the functions/variables used in this subsection from similar functions/variables used in the previous subsection. This superscript may be suffixed with either ":non-split" or ":split", depending on whether the function/variable applies to non-split servers or to split servers, respectively.

We develop an analysis that allows us to apply the schedulability test in Equation 5.17 to non-split servers in two steps. First, we revisit the analysis developed in Subsection 5.3.1 to take into account the effect of the reserve mechanism on the computing demand of a non-split server. Second, we factor into our analysis the effect of the reserve mechanism on the computing supply of a non-split server.

In Equation 5.16, we decomposed the demand of a server, $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q,t)$, into three components. The first, $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$, comprises the execution requirements induced by each task mapped to server $\tilde{P}_q$, including not only its execution time, but also overheads that may arise because of mechanisms used by the scheduling algorithm, i.e. timers, task releases and context switches. Clearly, these requirements are not affected by the use of reserves. However, now we also need to take into account the *Release Interference*, $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q,t)$, i.e. the overhead incurred by the release of tasks mapped to other servers that share the processor with $\tilde{P}_q$. Furthermore, as we explain below, the other two components are also affected by the use of reserves. Hence, in a first approximation, we have:

$$\begin{aligned}\mathrm{dbf}^{\mathrm{sb:non-split}}(\tilde{P}_q,t) = \\ \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) + \mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) + \mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{IntO}}(\tilde{P}_q,t) + \mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q,t)\end{aligned} \tag{5.18}$$

We now proceed with the development of the analytical expressions for the $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{XxxX}}$ parameters on the right-hand side of Equation 5.18.

The CPMD overheads now comprise not only the preemptions caused by tasks in the server, but also the preemptions incurred due to the reserve mechanism. In the worst case, the reserve mechanism preempts the last job that executes in the server's reserve. Thus, during an interval of duration S, a non-split server incurs at most one additional preemption due to the use of reserves:

$$\mathrm{nr}^{\mathrm{sb:non-split}}_{\mathrm{pree}}(\tilde{P}_q,t) = \left\lceil \frac{t+ResL}{S} \right\rceil + \mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q,t) \tag{5.19}$$

where *ResL*, the *Reserve Latency*, is an overhead akin to the release overheads that occurs at the beginning of a reserve and is explained later in this subsection.

Accordingly, the worst-case overall CPMD cost for that server in a time interval of length t is given by:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) = \mathrm{nr}^{\mathrm{sb:non-split}}_{\mathrm{pree}}(\tilde{P}_q,t) \cdot CpmdO \tag{5.20}$$

Taking into account interrupts with reserves is somewhat harder than in the case of a uniprocessor. Indeed, whereas on a uniprocessor a sporadic interrupt can be modelled as a sporadic interfering task, this is not the case with reserves. This is because reserve boundaries behave like temporal firewalls, and therefore an interrupt affects only the reserve that was active at the time the interrupt task is executed. Hence, each interrupt has to be modelled as a bursty periodic task. Given the complexity of such a formulation, we deal with it in Appendix A. Let $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{IntO}}(\tilde{P}_q,t)$ denote the amount of time required for executing all fired interrupts inside the reserves of $\tilde{P}_q$ in a time interval of length t, as determined in Appendix A.

Finally, we consider the release overhead, i.e. the processor time required to handle the release of jobs. On slot-based task-splitting scheduling algorithms, a server's tasks share the processor with other tasks whose servers are assigned to the same processor. Consistent with our implementation [SPT12] we assume that all jobs of a task are released on the processor(s) to which the task is assigned. As shown in Figure 5.4, non-split server $\tilde{P}_q$ can incur not only the release overheads of its own jobs, but also the release overheads of the jobs of both its immediate neighbour servers, $\tilde{P}_{q-1}$ and $\tilde{P}_{q+1}$.

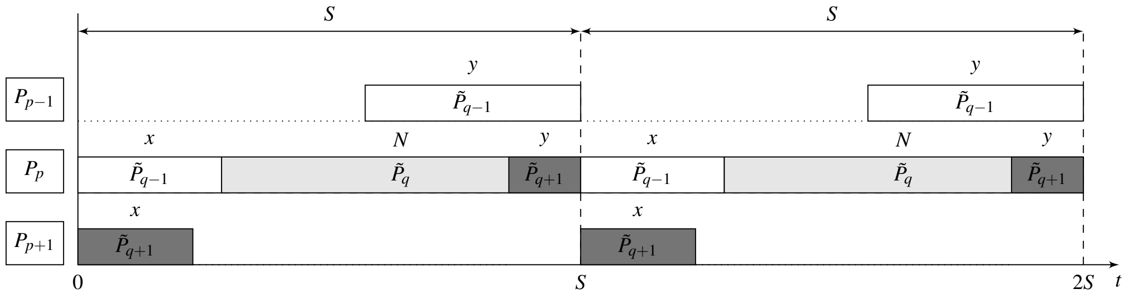


Figure 5.4: Illustration of the release interference for non-split servers. In this example server $\tilde{P}_q$ may suffer release interference from the arrivals of tasks served by $\tilde{P}_{q-1}$ and $\tilde{P}_{q+1}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_p).

Recall that the release overhead cost of all jobs of $\tau[\tilde{P}_q]$ in a time interval of length t is already accounted for in the derivation of $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$ (see Equation 5.10). Therefore, what remains is to incorporate the release interference, $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q,t)$, the release overhead cost from neighbouring servers, i.e. servers sharing the same processor:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q-1},t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q+1},t) \tag{5.21}$$

where $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_q,t)$ (see Equation 5.9) denotes the amount of time required to release all jobs assigned to server $\tilde{P}_q$ in a time interval of length t.

We now consider the effect of the reserve mechanism on the amount of time supplied to the execution of the tasks of a non-split server. In comparison with the analysis in Subsection 5.3.1, the amount of time supplied to the execution of a non-split server is reduced because of two factors. The first is the sharing of the processor with other servers. The second is the imprecision of the timers used to measure the duration of the reserves. We analyse the effect of each of these factors in turn.

In slot-based task-splitting scheduling algorithms, a non-split server $\tilde{P}_q$ is confined to execute within a single periodic reserve of length $Res^{len}[\tilde{P}_q]$, which is available every S time units:

$$Res^{len}[\tilde{P}_q] = U^{infl}[\tilde{P}_q] \cdot S \tag{5.22}$$

where $U^{infl}[\tilde{P}_q]$ represents the inflated processing capacity of server $\tilde{P}_q$. Thus, for any time interval of length t, only a fraction of such interval is supplied for the execution of a server. We model the unavailability of the reserve as an interfering *fake task* with attributes:

$$\begin{aligned} C^{fake} &= S - Res^{len}[\tilde{P}_q] \\ T^{fake} &= S \\ D^{fake} &= C^{fake} \end{aligned} \tag{5.23}$$

Hence, the supply-bound function for non-split servers can be expressed, in a first approximation, as follows:

$$\mathrm{sbf}^{\mathrm{sb:non-split}}(\tilde{P}_q,t) = t - \max\left(0, \left\lfloor \frac{t - D^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} \tag{5.24}$$

The second source of the reduction in the amount of time supplied to the execution of a non-split server is the processing time required to switch from one reserve to the next. The switch of reserves is also associated with a delay between the time at which the current reserve should end and the time at which it actually ends that is caused by timer drift, the execution of the timer callback, which performs the actions required to switch from a reserve to another, and the execution of the scheduler. To facilitate the experimental measurement of this parameter, we decided to group these three parameters in a single one that we call reserve latency. This is illustrated in Figure 5.5, which also shows that this parameter includes the time required to switch to the first job of the new reserve.

We model this reduction in the supply of processing time to the reserve as an increase in the execution demand of the fake task. Let *ResL* be a upper bound for the reserve latency. The

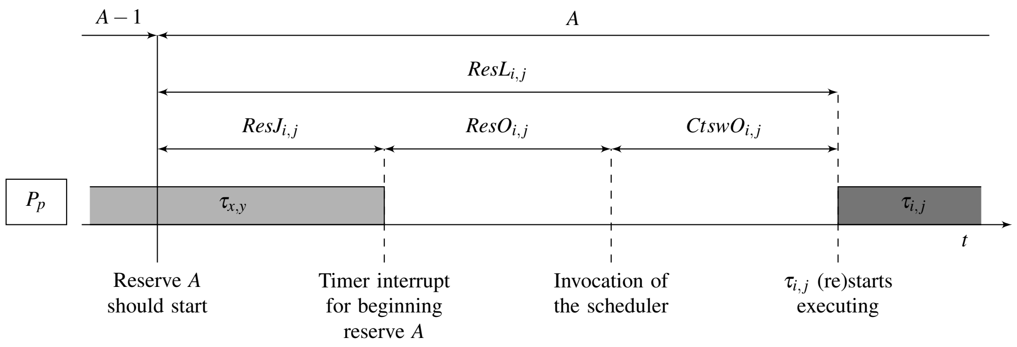


Figure 5.5: Illustration of the reserve overhead. In this example, the execution of job $\tau_{i,j}$, inside reserve A, is delayed by $ResL_{i,j}$, the overhead of that reserve.

expression for $\text{sbf}^{\text{sb:non-split}}(\tilde{P}_q,t)$ then becomes:

$$\text{sbf}^{\text{sb:non-split}}(\tilde{P}_q,t) = t - \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \tag{5.25}$$

Replacing this expression in Inequality 5.17 and moving some terms from the right-hand side to the left-hand side, we obtain the following schedulability test for non-split servers:

$$\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q,t) + \text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q,t) \leq t, \forall t > 0 \tag{5.26}$$

where $\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q,t)$ is given by Equation 5.18 and $\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q,t)$ is given by:

$$\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q,t) = \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \tag{5.27}$$

To complete the analysis of non-split servers, we provide an algorithm to compute the inflated utilization of server $\tilde{P}_q$, $U^{infl}[\tilde{P}_q]$. Indeed, evaluating $\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q,t)$ depends on $U^{infl}[\tilde{P}_q]$, via $Res^{len}[\tilde{P}_q]$ and C^{fake} (see Equations. 5.22 and 5.23). Furthermore, $\text{dbf}^{\text{sb:non-split}}_{\text{IntO}}(\tilde{P}_q,t)$ also depends on $U^{infl}[\tilde{P}_q]$, as shown in Appendix A.

In order to achieve the highest possible schedulability, we are interested in determining the minimum inflated utilization required for server $\tilde{P}_q$. For that purpose, we employ a "binary search" approach. We use the schedulability test developed in this section to determine an interval that is guaranteed to include the inflated utilization. This interval can be arbitrarily small. We start with the interval $[U[\tilde{P}_q], 1.0]$. Then we successively halve this interval in such a way that the inflated utilization is guaranteed to be in every generated interval. Algorithm 5.2 shows the pseudo-code for the `inflate_sb_non_split` function. In each iteration, it computes the current interval's midpoint and then applies the schedulability test, implemented in the `dbf_sb_non_split_-check` function to that utilization value. If the outcome of the test is positive, i.e. the server is schedulable with that utilization, the midpoint value computed becomes the upper bound of the interval in the next iteration, otherwise it becomes the lower bound. The algorithm converges rather rapidly, and in more or less ten iterations, it generates an interval that is less than 0.1% wide that contains the minimum inflated capacity of the server required for the server to be schedulable, according to the schedulability test in Inequality 5.26. In Section 5.5, we provide some details on the implementation of the `dbf_sb_non_split_check` function.

Algorithm 5.2: Pseudo-code algorithm of the `inflate_sb_non_split` function.

```
Inputs: $\tilde{P}_q$ {server to analyse}
        $\varepsilon$ {accuracy of desired estimate}
        $t$ {time interval for computing the demand bound function}
Output: $U^{infl}[\tilde{P}_q]$ {minimum inflated utilization, with an error smaller than $\varepsilon$,
                 that ensures schedulability of $\tilde{P}_q$}

$U_{min} \leftarrow U[\tilde{P}_q]$
$U_{max} \leftarrow 1.0$
while $U_{max} - U_{min} > \varepsilon$ do
    $U^{infl}[\tilde{P}_q] \leftarrow (U_{min} + U_{max})/2$
    if dbf_sb_non_split_check($\tilde{P}_q, t$) then
        $U_{max} = (U_{min} + U_{max})/2$
    else
        $U_{min} = (U_{min} + U_{max})/2$
    end if
    $U^{infl}[\tilde{P}_q] \leftarrow U_{max}$
end while
```

5.3.2.2 Split servers

In this subsection, we develop a schedulability analysis for split-servers similar to the one developed in the previous subsection. Again, we use a schedulability test based on the demand-bound and the supply-bound functions:

$$\text{dbf}^{\text{sb:split}}(\tilde{P}_q, t) \leq \text{sbf}^{\text{sb:split}}(\tilde{P}_q, t), \forall t > 0 \tag{5.28}$$

and we derive the expression for $\text{dbf}^{\text{sb:split}}(\tilde{P}_q, t)$, by revisiting the analysis developed in Subsection 5.3.1 to take into account the increase in the demand of processing time because of the reserve mechanism, and the expression for $\text{sbf}^{\text{sb:split}}(\tilde{P}_q, t)$, by accounting for the reduction in the amount of time supplied to the server because of the reserve mechanism.

Based on the arguments used in the previous subsection, we can express $\text{dbf}^{\text{sb:split}}(\tilde{P}_q, t)$ as follows:

$$\begin{aligned}&\text{dbf}^{\text{sb:split}}(\tilde{P}_q, t) = \\ &\quad \text{dbf}^{\text{sb:split}}(\tau[\tilde{P}_q], t) + \text{dbf}^{\text{sb:split}}_{\text{CpmdO}}(\tilde{P}_q, t) + \text{dbf}^{\text{sb:split}}_{\text{IntO}}(\tilde{P}_q, t) + \text{dbf}^{\text{sb:split}}_{\text{RelI}}(\tilde{P}_q, t)\end{aligned} \tag{5.29}$$

Like non-split servers, preemptions and migrations of tasks, interrupts and the release of tasks of servers that share processors with the split server also need to be taken into account, and amended specifically to split servers. However, unlike with non-split servers, we also need to amend $\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t)$, i.e. the processor demand of the server's tasks, assuming that they are executed in their own processor and accounting for the overheads incurred by the timers, the release of the server's tasks and the context switches between server's tasks. This is because the release of tasks of a split server may use IPI, which, as we show below, affects the components of the demand accounted for in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$. We now develop an expression for each term in Equation 5.29.

As described in Section 4.7, slot-based task-splitting scheduling algorithms may use IPIs to notify the dispatcher in another processor of the release of a task. As a result, the dispatching of a task may incur an IPI latency, $IpiL$. (Note that this parameter does not include the time required for context switching; this is already accounted for, as it will occur whether or not the release is via an IPI.) Figure 5.6 illustrates such a case. The arrival of a job of task τ_i assigned to a split server shared between processors P_p and P_{p-1}, for instance, occurs at a time instant t and is handled on processor P_p, but this time instant t falls inside the reserve of that server on the other processor, P_{p-1}. If this job is the highest-priority job of its server, P_p notifies P_{p-1} of the new arrival via an IPI. Clearly, the overhead caused by the IPI, $IpiL_{i,j}$, only delays the dispatch of job $\tau_{i,j}$ (and only if job $\tau_{i,j}$ is the highest-priority job of its server). Thus, the IPI latency, has an effect similar to the release jitter.

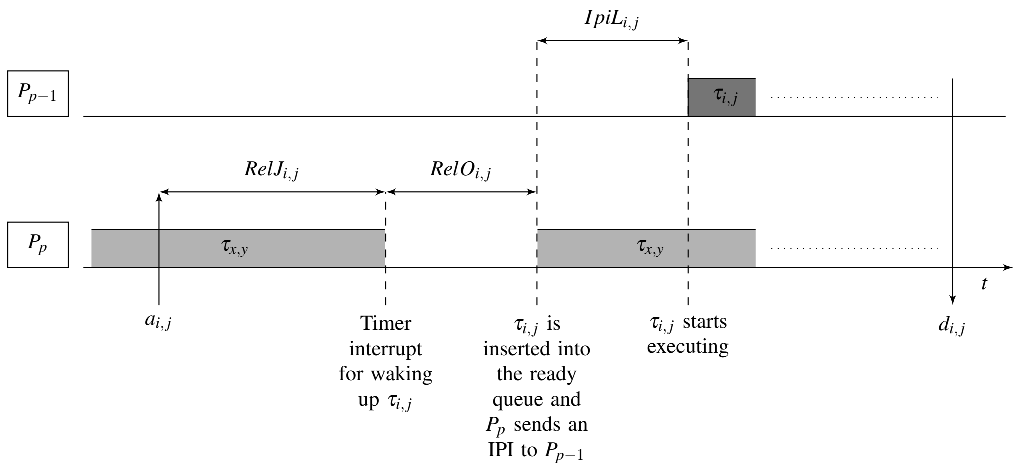

$IpiL_{i,j}$
P_{p-1}
$\tau_{i,j}$
$RelJ_{i,j}$
$RelO_{i,j}$
P_p
$\tau_{x,y}$
$\tau_{x,y}$
t
$a_{i,j}$
Timer interrupt for waking up $\tau_{i,j}$
$\tau_{i,j}$ is inserted into the ready queue and P_p sends an IPI to P_{p-1}
$\tau_{i,j}$ starts executing
$d_{i,j}$

Figure 5.6: Illustration of the IPI latency at the release of a split job. It does not include the time for context switching.

Let $IpiL$ be a upper bound for the IPI latency. Then, incorporating (in a pessimistic manner, since in practice not all jobs are affected) this effect is just like increasing the release jitter in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$, see Equation 5.11:

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{sb:split}}(\tau[\tilde{P}_q],t) = \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q],t) + \\ &\quad \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ + IpiL}{T_i} \right\rfloor + 1\right) \cdot (C_i + RelO + 2 \cdot CtswJ) \end{aligned} \tag{5.30}$$

The cost of the CPMD is more of a concern for split servers than for non-split servers, because tasks may actually migrate between two processors. Nevertheless, in our analysis we assume a worst-case CPMD overhead, $CpmdO$, which accounts for both. Hence, compared with modelling

CPMD overheads for non-split servers, the only difference is that other than EDF preemptions, split servers incur two additional preemptions per-time slot (*vs.* one for non-split servers), one for each reserve they use. Accordingly, $\mathrm{nr}^{\mathrm{sb:split}}_{\mathrm{pree}}(\tilde{P}_q,t)$ is calculated as follows:

$$\mathrm{nr}^{\mathrm{sb:split}}_{\mathrm{pree}}(\tilde{P}_q,t) = 2\cdot\left\lceil\frac{t+ResL}{S}\right\rceil + \mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q,t) \tag{5.31}$$

and the cost of the CPMD over a time interval of length t is:

$$\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) = \mathrm{nr}^{\mathrm{sb:split}}_{\mathrm{pree}}(\tilde{P}_q,t)\cdot CpmdO \tag{5.32}$$

The interrupt overhead for split servers is modelled as for non-split servers; that is, each interrupt is modelled as bursty periodic task. Given the complexity of such a formulation we deal with that in Appendix A. Let $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{IntO}}(\tilde{P}_q,t)$ be a upper bound on the amount of time required for executing all fired interrupts inside the reserves of $\tilde{P}_q$ in a time interval of length t.

Finally, we consider the release interference by servers that execute on the same processor. As illustrated in Figure 5.7, a split server, $\tilde{P}_q$, can incur the release interference of, at most, the previous two, $\tilde{P}_{q-1}$ and $\tilde{P}_{q-2}$, and also, at most, the next two servers, $\tilde{P}_{q+1}$ and $\tilde{P}_{q+2}$.

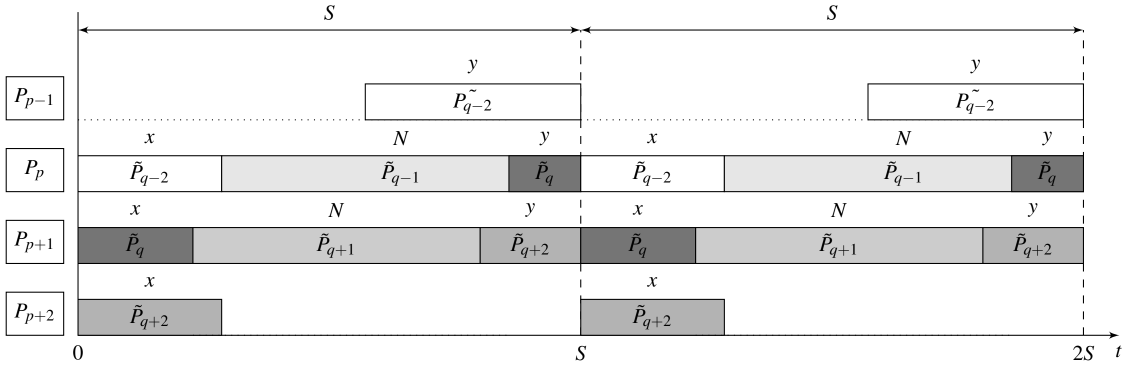


Figure 5.7: Illustration of the potential release interference exerted by neighbouring servers. In this example server $\tilde{P}_q$ may suffer release interference from the arrivals of tasks served by $\tilde{P}_{q-1}$ and $\tilde{P}_{q-2}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_p) but also from the arrivals of tasks served by $\tilde{P}_{q+1}$ and $\tilde{P}_{q+2}$ (if these occur at an instant when $\tilde{P}_q$ is mapped to processor P_{p+1}).

Thus, the release interference on $\tilde{P}_q$ by its neighbour servers is computed as:

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{RelI}}(\tilde{P}_q,t) = \\ &\qquad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q-1},t)+\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q-2},t)+\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q+1},t)+\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q+2},t)\end{aligned} \tag{5.33}$$

This concludes the analysis of the effect of the reserve mechanism on the processing demand by a split-server. Before, we analyse the effect of the reserve mechanism on the amount of time supplied to the server, $\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q,t)$, recall the need of staggered time slots, due to the instantaneous migration problem (see Section 4.5). In that description, the reserves of a split server $\tilde{P}_q$ on different processors P_p and P_{p+1} are temporally adjacent, as illustrated in Figure 4.5. In practice, because of the limitations in the measurement of the duration of a reserve, this layout requires explicit synchronization between the dispatchers on both processors to prevent simultaneous execution of the same task by both processors at the beginning of a time slot. This synchronization would lead to an additional overhead, which can be avoided by shifting the beginning of the time slot on processor P_{p+1} in time, i.e. by staggering the time slots in consecutive processors.

In [BA11], the authors showed that the time shift Ω given by $\Omega = (S - x[P_{p+1}] - y[P_p])/2$ is optimal with respect to utilization for a split server $\tilde{P}_q$ whose reserves are $x[P_{p+1}]$ and $y[P_p]$. With this value, the end of $x[P_{p+1}]$ is also separated from the start of $y[P_p]$ by the same Ω time units, as illustrated in Figure 5.8. Therefore, Ω is also optimal with respect to the reserve overhead tolerated, i.e., it is the time shift that provides the maximum protection against race conditions caused by the reserve jitter that may arise among schedulers of processors with reserves that are mapped to the same server.

Although this result was formulated in the context of NPS-F, it applies to any slot-based task-splitting scheduling algorithm. Therefore in our analysis, we assume that the two reserves of a split-server $\tilde{P}_q$ are Ω apart of each other.

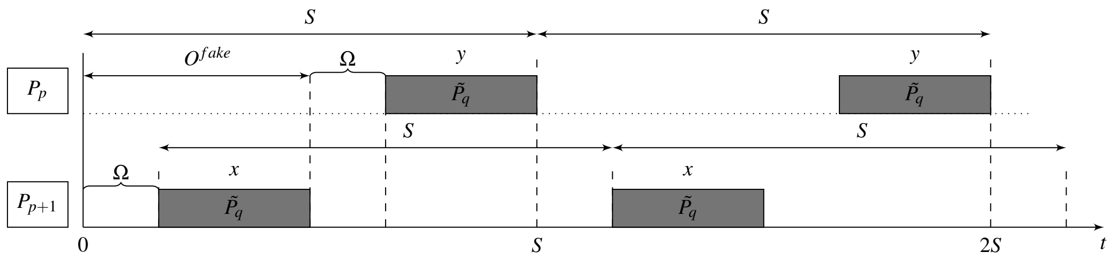


Figure 5.8: Illustration of the Ω and O^{fake} parameters.

We now proceed with the development of the reduction in the time supplied to execute the tasks of a split server because of the reserve mechanism.

Let $U_x^{infl}[\tilde{P}_q]$ and $U_y^{infl}[\tilde{P}_q]$ be the fractions of $U^{infl}[\tilde{P}_q]$ assigned to processors P_{p+1} and P_p, respectively. Then the durations of the reserves of $\tilde{P}_q$ are given by:

$$\begin{aligned} U^{infl}[\tilde{P}_q] &= U_x^{infl}[\tilde{P}_q] + U_y^{infl}[\tilde{P}_q] \\ x[P_{p+1}] &= U_x^{infl}[\tilde{P}_q] \cdot S \\ y[P_p] &= U_y^{infl}[\tilde{P}_q] \cdot S \end{aligned} \tag{5.34}$$

As in the analysis for non-split servers, we model the unavailability of the processor outside the reserves with fake tasks, now two per-time slot, each with the following parameters:

$$\begin{aligned}
C^{fake} &= \Omega \\
T^{fake} &= S \\
D^{fake} &= C^{fake}
\end{aligned} \tag{5.35}$$

Although the two fake tasks have the same arrival rate, they arrive at a relative offset. To account for the worst case, we assume that the first fake task arrives at $t = 0$ and the second task arrives at an offset of:

$$O^{fake} = \Omega + \min(U_x^{infl}[\tilde{P}_q], U_y^{infl}[\tilde{P}_q]) \cdot S \tag{5.36}$$

Thus we can express the amount of time supplied to the execution of tasks of the split server, $\mathrm{sbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t)$, as:

$$\begin{aligned}
&\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) = \\
&\quad t - \\
&\quad \max\left(0, \left\lfloor \frac{t - D^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} - \\
&\quad \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake}
\end{aligned} \tag{5.37}$$

Like the reserve of a non-split server, each of the two reserves of a split server incurs the reserve overhead. Let *ResL* be a upper bound for the reserve latency. Thus, to take into account this overhead, we do just as in the case of non-split servers, i.e. we add *ResL* to the execution demand of each of the two fake tasks, and $\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$ becomes:

$$\begin{aligned}
&\mathrm{sbf}^{\mathrm{sb:split}}(\tau[\tilde{P}_q], t) = \\
&\quad t - \\
&\quad \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) - \\
&\quad \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL)
\end{aligned} \tag{5.38}$$

Replacing this expression in Inequality 5.28 and moving some terms from the right-hand side to the left-hand side, we obtain the following schedulability test for split servers:

$$\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{5.39}$$

where $\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q,t)$ is given by Equation 5.29 and $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q,t)$ is given by:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q,t) = \\ &\quad \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) + \\ &\quad \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \end{aligned} \tag{5.40}$$

To complete the analysis of split servers, we provide an algorithm to compute the inflated utilization of server $\tilde{P}_q$, $U^{infl}[\tilde{P}_q]$. Indeed, evaluating $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q,t)$ depends on $U^{infl}[\tilde{P}_q]$, via $x[P_{p+1}]$, $y[P_{p+1}]$, Ω, C^{fake} and O^{fake}(see Equations 4.8, 5.34, 5.35, and 5.36). Furthermore, $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{IntO}}(\tilde{P}_q,t)$ also depends on $U^{infl}[\tilde{P}_q]$, as shown in Appendix A.

In order to achieve the highest possible schedulability, we are interested in determining the minimum inflated utilization required for server $\tilde{P}_q$. The algorithm we use for split servers is similar to that used for non-split servers, presented in Algorithm 5.2, except that it uses the function `dbf_sb_split_check`, which implements the schedulability test in Equation 5.28, rather than function `dbf_sb_non_split_check`. In Section 5.5, we provide some details on the implementation of these functions.

5.4 New server-to-processor assignment procedure

The application of the schedulability tests developed in the previous section raises two main issues. First, computing the inflation of the utilization of each server requires knowledge of whether or not the server is split, and of which servers it shares the processor with. However, this depends, among other things, on the inflated utilization of the server. Second, when a server is split between two processors, in marginal cases, the additional overheads incurred would necessitate the combined length of its two reserves to exceed S (which is a unworkable arrangement, given that they should not overlap in time). To prevent this undesirable outcome, we specify two assignment rules, which further exacerbate the first issue. Thus, in this section, we start by describing the assignment rules. After that, we address how to resolve the circularity associated with the first issue.

5.4.1 Assignment rules

The schedulability analysis formulated so far considers sporadic task sets with arbitrary deadlines. Therefore, the time slot length, S, cannot be computed according to the Equation 4.2, because it only consider task inter-arrival times, T_i. It has also consider the deadlines, D_i. Therefore, we amend Equation 4.2 to:

$$S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i), \forall \tau_i \in \tau \tag{5.41}$$

To prevent the creation of reserves too short to be useful, we add the following rule to the assignment algorithms that are presented below:

A1: Whenever a server would be split between two processors P_p and P_{p+1} in such a way that the length of the second reserve (i.e. on P_{p+1}) would be larger than the length of its only reserve had it been assigned to a single reserve on P_{p+1}, then the server should not be split, but rather assigned as non-split to P_{p+1}.

Figures 5.9 illustrates rule A1 using aligned time slots for reasons of clarity. Clearly, if the size of the non-split reserve is smaller than that of the second reserve, not splitting the server will lead to a lower computation demand by the server in both the first and the second processor. This means that there will be more computational resources for the remaining servers in the second processor. Although the computational resources not used on the first processor will not be used to satisfy the demand of the task set to schedule, they can be used by other (non-real time) tasks.

On the other hand, if the second reserve of the split server is shorter than the single reserve required if the server were not split, it must be the case that the first reserve is used for satisfying the demand of the server's tasks, and therefore, for the sake of improving the schedulability, the server should be split.

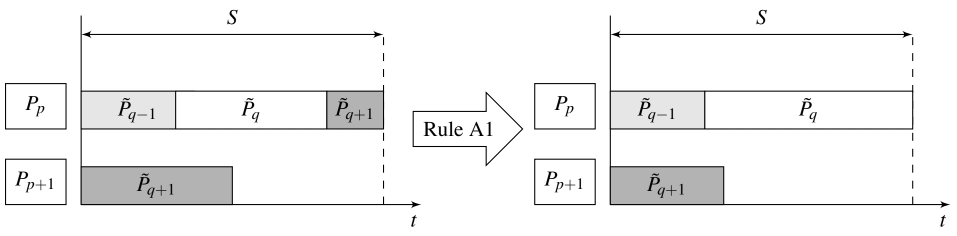


Figure 5.9: Illustration of the assignment rule A1.

Another issue concerns the case when the two reserves of a split server (possibly after application of rule A1) add up to almost S, or even surpass it. As a result, the schedulers on two processors might attempt to run the same task simultaneously. To prevent such a scenario, we specify the following rule:

A2: In cases where a server would be split such that $(U_x^{infl}[\tilde{P}_q] + U_y^{infl}[\tilde{P}_q]) \cdot S > S - ResL$, the server should instead become a *single server*.

A single server is assigned to a processor, utilizing its entire processing capacity, without being confined in a time reserve. This arrangement amounts to partitioning. Figure 5.10 illustrates rule A2.

5.4.2 Assignment procedure

Section 4.3 suggests that server-to-processor assignment is straightforward once the servers have been inflated. However, with the schedulability tests developed in the previous section, this is not so. The challenge is that server inflation depends on the assignment of servers to processors, because the release interference overhead depends on which servers are allocated the same processor.

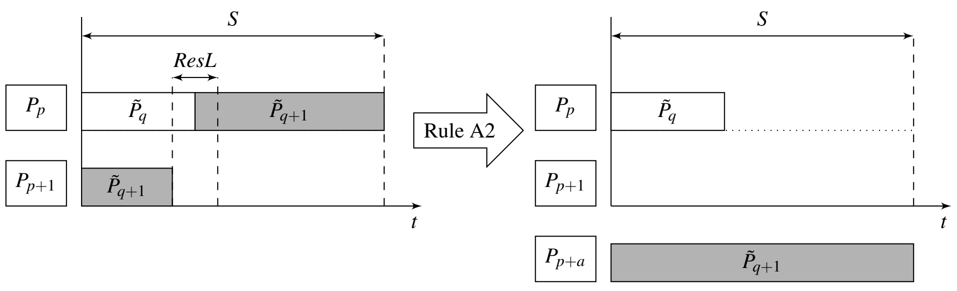


Figure 5.10: Illustration of the assignment rule A2.

Therefore, we have a circularity issue: inflation depends on the assignment, and the assignment depends on the inflation. For example, when we first inflate a server $\tilde{P}_{q-1}$, we do not yet know the servers that it will share a processor with it. We can assume that the next server, $\tilde{P}_q$, will share the processor with the server currently being analysed, but later, because of the application of rule A2, server $\tilde{P}_q$ may be allocated its own processor (as a single server), and therefore server $\tilde{P}_{q-1}$ will share the processor not with that server but with the one that follows it, i.e. server $\tilde{P}_{q+1}$, and it will have to be re-inflated.

The approach we use to overcome this issue is backtracking. To limit the amount of backtracking, we merge several steps of the generic algorithm in a single step. In the next two subsections, we illustrate the application of this approach to S-EKG and to NPS-F, respectively.

5.4.2.1 Task to processor assignment procedure in S-EKG

The distinctive feature of S-EKG is that the split servers, if any, have only one task. To ensure this, we merge the four steps of the generic algorithm in a single one. The full algorithm is somewhat complex, therefore, we just provide an overview of its main steps, which are illustrated in Figure 5.11.

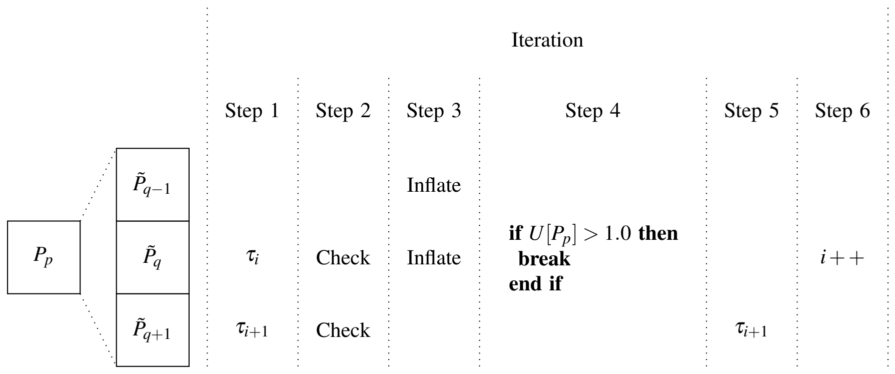


Figure 5.11: Illustration of the S-EKG task-to-processor mapping.

The algorithm starts by assigning empty servers (an *empty server* is a server without any task assigned to it) to the processors. All processors are allocated to a non-split server, one split server per-predecessor processor and one split server per-successor processor, so that the first and the last processors are allocated to only two servers, whereas the other processors are allocated to three servers. Then, it iterates over the set of tasks, two tasks at time, if available, and it assigns the tasks to the servers in an attempt to maximize the utilization of each server, subject to the constraint that each split server has at most one task. In the first step (Step 1), it provisionally assigns tasks τ_i and τ_{i+1} to $\tilde{P}_q$, the non-split server, and $\tilde{P}_{q+1}$, the split-server shared with the next processor, respectively, by invoking the `add_task_to_server` function. Then, it checks (Step 2) the schedulability of each server by invoking the `dbf_part_check` function. If some server with only one task is not schedulable, then the task set is also not schedulable. Otherwise, if the non-split server is not schedulable, the algorithm backtracks and assigns τ_i to $\tilde{P}_{q+1}$, and moves to the next iteration (where it will map tasks τ_{i+1} and τ_{i+2} to servers $\tilde{P}_{q+2}$ and $\tilde{P}_{q+3}$, respectively, and check their schedulability). If both servers are schedulable, it proceeds by inflating (Step 3) the capacity of the previous, $\tilde{P}_{q-1}$, and the current, $\tilde{P}_q$, servers by invoking the `inflate_-sb_split` and `inflate_sb_non_split` functions, respectively. It then checks (Step 4), if $U[P_p] = U_x^{infl}[\tilde{P}_{q-1}] + U^{infl}[\tilde{P}_q]$ is larger than 1.0. If yes, then it proceeds as in Step 2, when the non-split server is not schedulable. Otherwise, (Step 5) it assigns τ_i permanently to $\tilde{P}_q$, removes τ_{i+1} from $\tilde{P}_{q+1}$ server, and moves to the next iteration (Step 6), in which it will attempt to map task τ_{i+1} to server $\tilde{P}_q$ and task τ_{i+2} to server $\tilde{P}_{q+1}$.

For sake of simplicity, in this description we omitted many details, including those related to the application of rules A1 and A2.

5.4.2.2 New server-to-processor assignment for NPS-F

In the case of NPS-F, to limit the amount of backtracking, we keep the first step of the generic algorithm, i.e. the mapping of tasks to servers, separated and merge the remaining steps in a single one. The mapping of tasks to servers is performed in a first step, as described in Algorithm 5.1, and is never undone. The backtracking can affect only the assignment of servers to processors, and therefore their inflation and the definition of the reserves.

Algorithm 5.3 shows the pseudo-code of the new merged step. It assigns servers to processors (employing a NF bin-packing heuristic) and maps processor reserves to servers. The algorithm iterates over the set of servers created by the task mapping algorithm. First, it tries to assign each server as a non-split server. For that purpose, it inflates the current server by invoking the `inflate_sb_non_split` function, which considers the interference of the previous and the next server. If $U[P_p]$ (the utilization of the current processor already assigned to other servers) plus $U^{infl}[\tilde{P}_q]$ (the inflated utilization of the current server) is smaller than or equal to 1.0 (100%), the current server $\tilde{P}_q$ is assigned (non-split) to the current processor P_p and the algorithm moves to the next server. Otherwise, it will try to assign the current server, $\tilde{P}_q$, as a split server. Thus, it computes the inflation of the server by invoking the `inflate_sb_split` function, which considers the interference of the previous two and also the next two servers. If rule A1 applies, then the server is assigned as a non-split server to the next processor, and the algorithm moves to the next server. If rule A2 does not apply, then the current server $\tilde{P}_q$ becomes a split server and is assigned to both the current and the next processor, and the algorithm moves to the next server.

Otherwise, i.e. if rule A2 applies, the server is classified as a single server, moved to the end of the server list (and the servers renumbered, for ease of description of the algorithm), so that it is later allocated a dedicated processor. Furthermore, the algorithm is restarted, because servers that have already been assigned to a processor may have to be re-inflated. For example, server $\tilde{P}_{q-1}$, which was inflated assuming that $\tilde{P}_q$ would share the processor with it, will now share the processor with $\tilde{P}_{q+1}$. However, this then entails the possibility that $\tilde{P}_q$ was not sufficiently inflated (since the release interference from tasks on $\tilde{P}_{q+1}$ might be greater than what the schedulability test assumed). Thus backtracking is performed only when rule A2 is applied. Furthermore it is bounded to the number of servers. This is because application of rule A2 determines that the server will become a single server, and therefore will no more be subject to application of rule A2.

For the sake of ease of understanding, Algorithm 5.3 does not include some improvements that could make it more efficient or that could reduce the pessimism in the server inflation for some task sets. For example, when the algorithm applies rule A2 to a server, it moves it to the end of the servers list and restarts the assignment from the beginning. However, there is no need to backtrack all the way back to the beginning: it would be enough to backtrack until the highest numbered processor whose y-reserve mapping is not affected. Therefore the amount of work that has to be redone can be limited by slightly changing the algorithm. Yet another improvement on the speed of the algorithm is to prevent attempting assignments that will surely fail. For example, if the current processor has already been assigned a non-split server, the current server cannot be assigned as non-split in that processor (or else the two servers' contents would have been lumped together into one server, at the bin-packing stage). Therefore, in this case, the algorithm should try immediately to assign the server as a split server. Yet another example is the case where the sum of the size of the x-reserve, in terms of utilization, and the uninflated utilization of the server under analysis is larger than 1.0. Clearly, that server cannot be assigned to the N-reserve, and therefore the algorithm should try immediately to assign the server as a split server.

Algorithm 5.3 takes a pessimistic stance and considers that a non-split server always shares the processor with two other servers, and that a split server always shares the processors with four other servers, but this is the worst-case. In the best-case scenario, a non-split server may share the processor with only one more server, and a split server with two other servers. Thus, by assuming the best-case, it is possible to eliminate any pessimism from the algorithm (all pessimism is included in the functions that inflate the servers). However, this comes at the cost of additional backtracking, whenever an assumption is proved wrong. Still, it is possible to reduce the pessimism without adding backtracking by taking into account previous assignment decisions. For example, when inflating a non-split server and the x-reserve of the current processor is empty, the algorithm need not consider the interference of the previous server, because they do not share processors.

5.4.2.3 Effect of assignment rules on the schedulability analysis

As shown in Algorithm 5.3, the introduction of assignment rule A2 may lead to backtracking. Although, as we have argued, backtracking is limited, it can nevertheless be undesirable for some

Algorithm 5.3: Pseudo-code of the new server-to-processor assignment algorithm.

```
Input: set of $k$ servers
Output: reserves for the $m$ processors, allocating them to the input servers

$S \leftarrow \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$ {time slot}
$restart \leftarrow$ true
while $restart$ do
    $restart \leftarrow$ false
    $p \leftarrow 1$ {processor index}
    $q \leftarrow 1$ {server index}
    $U[P_p] \leftarrow 0$
    while $q \leq k$ and $Type[\tilde{P}_q] \neq$ SINGLE do
        $Type[\tilde{P}_q] \leftarrow$ NON-SPLIT
        $t \leftarrow 2 \cdot$ lcm_T$(\tilde{P}_{q-1}, \tilde{P}_q, \tilde{P}_{q+1})$
        $U^{infl}[\tilde{P}_q] \leftarrow$ inflate_sb_non_split$(\tilde{P}_q, t)$
        if $U[P_p] + U^{infl}[\tilde{P}_q] \leq 1.0$ then
            {server need not be split – we need not check rule A1, at this point}
            $U[P_p] \leftarrow U[P_p] + U^{infl}[\tilde{P}_q]$
            add_server_to_processor_N$(P_p, \tilde{P}_q, U^{infl}[\tilde{P}_q])$
        else
            {Server cannot be NON-SPLIT on $P_p$: try to split it}
            $U^{infl}_{tmp} \leftarrow U^{infl}[\tilde{P}_q]$ {Note that we are considering the interference by $\tilde{P}_{q-1}$, but that is not needed}
            $t \leftarrow 2 \cdot$ lcm_T$(\tilde{P}_{q-2}, \tilde{P}_{q-1}, \tilde{P}_q, \tilde{P}_{q+1}, \tilde{P}_{q+2})$
            $U^{infl}[\tilde{P}_q] \leftarrow$ inflate_sb_split$(\tilde{P}_q, t)$
            $U^{infl}_y[\tilde{P}_q] \leftarrow 1.0 - U[P_p]$
            $U^{infl}_x[\tilde{P}_q] \leftarrow U^{infl}[\tilde{P}_q] - U^{infl}_y[\tilde{P}_q]$
            if $U^{infl}_x[\tilde{P}_q] \geq U^{infl}_{tmp}$ then
                {Rule A1 – note that inflate_sb_non_split() always considers 3 servers, but in this case we need only consider 2}
                adjust_reserves$(P_p)$ {the y reserve becomes empty}
                $p \leftarrow p + 1$
                $U^{infl}[\tilde{P}_q] \leftarrow U^{infl}_{tmp}$ {non-split server inflated utilization, prev. computed}
                add_server_to_processor_N$(P_p, \tilde{P}_q, U^{infl}[\tilde{P}_q])$
                $U[P_p] \leftarrow U^{infl}[\tilde{P}_q]$
            else
                if $(U^{infl}_x[\tilde{P}_q] + U^{infl}_y[\tilde{P}_q]) \geq 1.0 - ResL/S$ then
                    {Rule A2}
                    $Type[\tilde{P}_q] \leftarrow$ SINGLE
                    move_to_last$(\tilde{P}_q)$ {so that split servers are assigned "neighbor" processors}
                    $restart \leftarrow$ true
                    break {start all over: inflation of other servers may have been affected}
                else
                    $Type[\tilde{P}_q] \leftarrow$ SPLIT
                    add_server_to_processor_Y$(P_p, \tilde{P}_q, U^{infl}_y[\tilde{P}_q])$
                    $U[P_p] \leftarrow U[P_p] + U^{infl}_y[\tilde{P}_q]$
                    $p \leftarrow p + 1$
                    add_server_to_processor_X$(P_p, \tilde{P}_q, U^{infl}_x[\tilde{P}_q])$
                    $U[P_p] \leftarrow U^{infl}_x[\tilde{P}_q]$
                end if
            end if
        end if
        $q \leftarrow q + 1$
    end while
    if $restart$ then
        continue
    end if
    $p \leftarrow p + 1$
    while $q \leq k$ do
        {handle SINGLE servers, if any}
        add_server_to_processor$(P_p, \tilde{P}_q)$
        $U[P_p] \leftarrow 1.0$
        $p \leftarrow p + 1$
        $q \leftarrow q + 1$
    end while
end while
```

task sets, because it could take too much time. In such cases, one can avoid backtracking at the cost of some pessimism, by amending Equations 5.21 and 5.33 (employed by the schedulability test), respectively, to:

$$\text{dbf}_{\text{RelI}}^{\text{sb:non-split}}(\tilde{P}_q,t) = \text{dbf}_{\text{RelO}}^{\text{part}}(prev\_server(\tilde{P}_q),t) + \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_A(q,t)},t) \tag{5.42}$$

and

$$\begin{aligned}&\text{dbf}_{\text{RelI}}^{\text{sb:split}}(\tilde{P}_q,t) = \\ &\quad \text{dbf}_{\text{RelO}}^{\text{part}}(prev\_server(\tilde{P}_q),t) + \text{dbf}_{\text{RelO}}^{\text{part}}(prev\_server(prev\_server(\tilde{P}_q)),t) + \\ &\quad \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_A(q,t)},t) + \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_B(q,t)},t)\end{aligned} \tag{5.43}$$

wherein *prev_server* denotes the previous server (not assigned a dedicated processor, i.e., not single server) and the server indexes q_A and q_B are computed as:

$$\begin{aligned}&q_A(q,t) \in \{q+1, \cdots, k\} : \\ &\quad \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_A(q,t)},t) \geq \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_p,t)\ \forall p \in \{q+1, \cdots, k\}\end{aligned} \tag{5.44}$$

and

$$\begin{aligned}&q_B(q,t) \in \{q+1, \cdots, k\} \setminus \{q_A\} : \\ &\quad \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_B(q,t)},t) \geq \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_p,t)\ \forall p\{q+1, \cdots, k\} \setminus \{q_A(q,t)\}\end{aligned} \tag{5.45}$$

That is, when inflating a server, rather than considering the release interference from the next server, we consider the maximum release interference that any of the servers not yet assigned may cause, thus taking a worst-case approach. Similarly for split servers, but in this case we need to consider the two largest values of the release interference that any of the servers not yet assigned may cause.

Note that the values of indexes $q_A(q,t)$ and $q_B(q,t)$ may change with the values of t. However, since both $\text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_A(q,t)},t)$ and $\text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_A(q,t)},t) + \text{dbf}_{\text{RelO}}^{\text{part}}(\tilde{P}_{q_B(q,t)},t)$ are non-decreasing functions of t, the applicability of the Quick-Processor Demand (QPA) EDF analysis, which is discussed in the next section, is not affected.

5.5 Schedulability tests

As discussed earlier, in slot-based task-splitting algorithms, overhead-aware schedulability testing has to be done at two stages: (i) during the task-to-server mapping and (ii) during the server-to-processor assignment. In our code, this testing is, respectively, performed by the C functions (i) `dbf_part_check` (implementing Equation 5.6) and (ii) `dbf_sb_non_split_check` or `dbf_sb_split_check` (implementing Equations 5.17 and 5.28, respectively).

All these functions check whether or not, at every instant within a time interval [1, t), where t is an argument of the functions, the supply of processor time satisfies the demand. Unlike what is possible for conventional uniprocessor EDF scheduling where certain techniques allow the safe use of much shorter intervals [GRS96, HBJK06, RCM96, Spu96], in our case, it is necessary to set t to twice the least common multiple of all T_is of the tasks of the server under consideration, including the fake task modelling the unavailability of the reserve(s). This can be a very big number, therefore the length t of this testing interval can be exceptionally long. This raises two difficulties. First, the value for t may exceed the range of a 64-bit integer. To overcome this limitation, we used the GNU Multiple Precision Arithmetic C-Library[1]. Second, a longer testing interval means many more iterations, in order to test for all integer values in the range [1, t). To speed up the analysis, we implemented the schedulability testing using Quick Processor-demand Analysis (QPA) [ZB09], which overcomes the need to test for all values in the interval [1, t). This technique works by identifying large sub-intervals within which no deadline misses may occur, and skipping them during testing. This way, for most cases, the analysis is significantly sped up. Algorithm 5.4 shows, in pseudo-code, how the QPA technique can be used with each of the schedulability tests we defined earlier (where dbf$^{\text{xxx}}$ stands for any of them).

Algorithm 5.4: Pseudo-code algorithm of the schedulability test functions.

```
Input: $\tilde{P}_q$ {server to analyse}
Returns: true if $\tilde{P}_q$ is schedulable, false otherwise

$t \leftarrow 2 \cdot \text{lcm\_T}(\tilde{P}_q)$
$d_{min} \leftarrow \min_{\tau_i \in \tau[\tilde{P}_q]}(D_i)$
while $\text{dbf}^{\text{xxx}}(\tilde{P}_q, t) \le t$ and $\text{dbf}^{\text{xxx}}(\tilde{P}_q, t) > d_{min}$ do
    if $\text{dbf}^{\text{xxx}}(\tilde{P}_q, t) < t$ then
        $t \leftarrow \text{dbf}^{\text{xxx}}(\tilde{P}_q, t)$
    else
        $t \leftarrow t - 1$
    end if
end while
return $\text{dbf}^{\text{xxx}}(\tilde{P}_q, t) \le d_{min}$ {true if $\tilde{P}_q$ is schedulable, false otherwise}
```

5.6 Summary

In this chapter, we proposed a unified schedulability analysis for slot-based scheduling algorithms. This new schedulability analysis is a processor demand-based analysis that supersedes the the original utilization-based analysis. The new schedulability analysis incorporates the overheads incurred by slot-based scheduling algorithms. It is used by the off-line task-to-processor assignment algorithm. It is more efficient and also more reliable than the previous analysis.

[1]Available online at `http://gmplib.org/`

Chapter 6

Reserve-based scheduling algorithm

6.1 Introduction

Given the increasing use of multiple processing units in computing systems, the problem of scheduling tasks with real-time constraints on such systems is particularly relevant. Therefore, although an optimal algorithm for this problem is not possible, neither from a theoretical [FGB10] nor from a practical perspective, we believe that there is always room for more scheduling algorithms.

In this chapter, we present the Carousel-EDF [SSTB13], a new hierarchical scheduling algorithm for a system of identical processors and its overhead-aware schedulability analysis based on demand-bound functions. We start with a description of the Carousel-EDF scheduling algorithm, then we ensue with a description of the implementation of Carousel-EDF in Linux kernel. We formulate the demand-based overhead-aware schedulability analysis for Carousel-EDF scheduling algorithm. We end this chapter with the derivations of the utilization bound and the preemption upper bound of the Carousel-EDF scheduling algorithm.

6.2 Basics of the reserve-based scheduling algorithm

Carousel-EDF is an offshoot of the generic slot-based task-splitting scheduling algorithm described in Chapter 4. It also relies on the concept of servers that are instantiated by the means of time reserves. However, contrarily to slot-based task-splitting scheduling algorithms, which migrate tasks at time slot and reserve boundaries, Carousel-EDF only migrates tasks at reserve boundaries. Consequently, the time is no longer constrained to the time slot boundaries, but it is only divided into time reserves instead. This approach preserves the utilization bounds of slot-based task-splitting scheduling algorithms, namely that of NPS-F, which is the highest among algorithms not based on a single dispatching queue and that have few preemptions. Furthermore, with respect to slot-based task-splitting scheduling algorithms, Carousel-EDF reduces up to 50% the worst-case number of context switches and the worst-case number of preemptions caused by the time division mechanism.

The Carousel-EDF scheduling algorithm comprises an off-line procedure and a run-time scheduling algorithm. The off-line procedure consists of (i) the partitioning of the task set into subsets that are mapped to servers; (ii) the sizing of the reserve of each server; (iii) the cycling sequence order of the server-to-processor assignment; and finally (iv) for each processor, the first server assigned to it and the size of the first reserve. The run-time scheduling algorithm is a two-level hierarchical algorithm, similar to the run-time scheduling algorithm under slot-based dispatching. The top-level scheduling algorithm allocates processors to servers for the duration of the corresponding reserves and ensures that for each server, at any time, there is at most one processor allocated to it. The low-level scheduling algorithm uses the EDF policy to schedule the tasks in a server during its reserves. In the next subsection, we describe the off-line procedure and in the following one the run-time task-dispatching algorithms.

6.2.1 Off-line procedure

Carousel-EDF employs the off-line procedure as in the slot-based task-splitting algorithm described in Section 4.3. Therefore, the first step of the off-line procedure is to partition the task set into subsets that are mapped to servers. Servers have unique identifiers in the range $\tilde{P}_1$ to $\tilde{P}_k$. The set of tasks in server $\tilde{P}_q$ is denoted $\tau[\tilde{P}_q]$. Recall that the utilization of server $\tilde{P}_q$, $U[\tilde{P}_q]$, is the sum of the utilizations of its mapped tasks, i.e. $U[\tilde{P}_q] = \sum_{\tau_i \in \tau[\tilde{P}_q]} u_i$. Because at any time a server can have at most one processor allocated to it, a server's utilization must not exceed 100%. Thus, the assignment of tasks to servers can be done by applying any bin-packing heuristic.

For the task set example presented in inset (a) of Figure 4.1 the outcome is thus equal to the presented in inset (b) of the same figure. Once the servers have been defined, the off-line procedure computes for each server the size of its reserve, i.e. the length of the time window during which a processor is allocated to a server. The reserves are periodic and we call this period the time slot, S, a term inherited from slot-based task-splitting scheduling algorithms. As in slot-based task-splitting scheduling algorithms, $S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$, where δ is an integer design parameter that allows a trade-off between the number of preemptions and the utilization bound of the algorithm.

Having determined the length of the time slot, S, the size of each reserve, $Res^{len}[\tilde{P}_q]$, can be computed by multiplying S with the inflated utilization of that server, which we denote $U^{infl}[\tilde{P}_q]$. The reason for inflation is the same as for slot-based task-splitting scheduling algorithms. Server utilization inflation is required because the tasks of a server can run only when a processor is allocated to that server. Thus, a server may have ready tasks, but cannot execute them. By inflating the utilization of a server, we ensure that the tasks deadlines are met in spite of that delay. Below, in Section 6.4 we present the server inflation algorithm. The last step defines the cycling sequence order of the server-to-processor assignment as well as the initial phasing of each processor. This assures that at any time instant there is a one-to-one relationship between processors and active servers. In other words, at any time instant, each processor is allocated to only one server and a server can be, at most, assigned to one processor (see Figure 6.1).

6.2.2 Run-time task-dispatching algorithm

The top-level hierarchical scheduling algorithm allocates processors to servers. Each processor is allocated for the duration of the corresponding reserve in a round-robin fashion. Furthermore, the schedule of each processor is staggered so that each server is assigned to at most one processor at any time and is periodically assigned, with period S, to some processor. As a result, each server moves from one processor to the next also in a round-robin fashion.

Figure 6.1 illustrates the run-time dispatching time-line produced by the Carousel-EDF for the example shown in Figure 4.1. In this example, we have four processors and five servers. At time zero, processor P_1 is allocated to server $\tilde{P}_1$. When that server's reserve expires, processor P_1 is allocated to server $\tilde{P}_2$, and so on until it is allocated to the last server $\tilde{P}_5$. Processor P_1 is allocated again to server $\tilde{P}_1$ at time $4S$; that is, at the next integer multiple of S after the reserve of the last server (in this case, $\tilde{P}_5$) expires on P_1. This ensures not only that every reserve has period S, but also that at any time there is at most one processor allocated to any server.

The cycling of servers (that inspired the name *Carousel*) through processors is also clear in Figure 6.1. Server $\tilde{P}_1$ is assigned to processor P_1 at time zero. At time S, it is assigned to processor P_4, then, at time $2S$, to processor P_3, and so on (it will return to processor P_1 every $4S$ time units). This ensures not only that every reserve has period S, but also that at any time there is at most one processor allocated to any server. Therefore, when the sum of the duration of all reserves is not a multiple of the time slot, there is a time gap between the end of the last reserve and the beginning of the first that we name *empty reserve*.

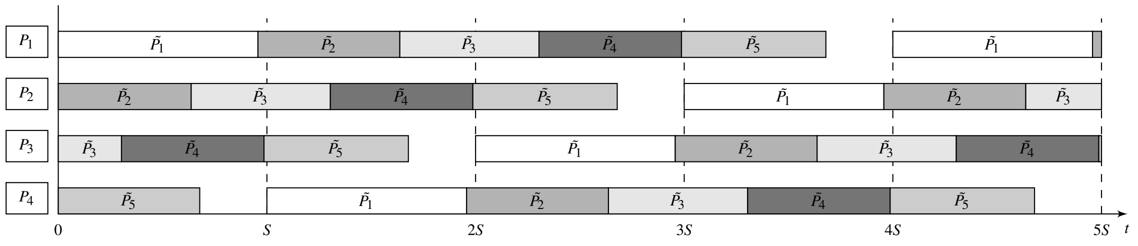


Figure 6.1: Run-time dispatching time-line produced by the Carousel-EDF for the task set example presented in Section 4.3.

The low-level scheduler uses EDF to schedule the tasks in a server. Because each reserve instance occurs always at a single processor, the low-level scheduler operates on a local basis, i.e. there is no need of coordination among the low-level schedulers of the different processors.

6.3 Implementation of the Carousel-EDF task-dispatching algorithm

In this section, we provide some details regarding the implementation of Carousel-EDF's top- and low-level scheduler. We also describe a new implementation technique for the release of tasks. This technique eliminates the task release interference across reserves. This is important in order to better understand the overhead-aware schedulability analysis of Carousel-EDF that we develop in the following section.

Following the same implementation approach of the slot-based task-splitting scheduling algorithm described in Section 4.6, we added the Carousel-EDF scheduler to the RT scheduling class of the PREEMPT-RT-patched Linux 3-2.11-rt20 kernel version [SSTB13]. Therefore, the Carousel-EDF implementation shares many issues with the slot-based task-splitting scheduling algorithm implementation. Actually, the changes introduced to the RT scheduling class are, basically, the same. Therefore, in this section, we opt for a high-level description of the Carousel-EDF implementation.

6.3.1 Scheduler implementation

In the context of this work, our implementations have followed one rule-of-thumb, which we believe is important for the implementation of scheduling algorithms for multiprocessor systems, that is: the interactions among processors must be reduced.

The implementation of the Carousel-EDF scheduler shares many issues with the slot-based scheduler described in Section 4.6. The Carousel-EDF scheduler implementation is also split into top- and low-level scheduler implementation. The implementation of the Carousel-EDF scheduler is supported by three data structures (see Figure 6.2): the per-processor `retas_rb_rq`; the `carousel` array; and `server`. Next, we discuss about our implementation approach with respect to each such data element.

We model each server as an instance of a new data structure called `server`, which stores the following information: (i) a field that uniquely identifies a server; (ii) a ready-queue; (ii) an atomic variable that exposes the server state, called `flag`; (iv) the length of its reserve, `res_len`; and finally, (v) the release-queue (whose purpose is explained later in this section). Tasks that are ready to run are kept in a ready-queue, ordered by their deadlines. However, rather than using a system wide ready-queue, our implementation uses a per-server ready-queue like the slot-based scheduler implementation described in Section 4.6. The purpose of `flag` is the same as in the slot-based scheduler implementation. Its main goal is to ensure that a task of a server is executed by only one processor at any time instant.

Carousel-EDF scheduling guarantees that at any time there is at most one processor allocated to any server, this eliminates contention in the access to the respective ready-queue. This is assured by the `carousel` array; that is, an array, which implements a circular queue, that stores the server identifier according to the cycling sequence order (in Section 6.4, we describe the algorithm to create such a server cycling sequence order). Note that, the server cycling sequence order is the same for all processors. The difference consists in the assignment of the first server to each processor.

For each processor, we use a timer for handling the server reserves, a variable to store the index of the carousel array that indicates the current allocated server, `curr_car_idx`, and a pointer to the current allocated server, `curr_srv`.

The top-level scheduler work as follows. At the setup each processor is configured with the first allocated server through the index of the carousel array, `curr_car_idx`, and the length of the first reserve (which in many cases is different from the server reserve length defined in `res_-len`). Indeed, due to this initial phasing (which is described in Section 6.4) of every server it is assured that any server is not allocated on two processors simultaneously. The timer is used to measure that initial reserve and the index of the array is used to get the pointer to the current allocated server (that is stored in `curr_srv`). At timer expiration, when the respective reserve ends, the `curr_car_idx` is incremented (if it reaches the end of the array it re-starts from the beginning), the pointer to the current allocated server is updated, and the timer is started for the new reserve. Then, it triggers the invocation the low-level scheduler so that the processor switches to a task belonging to the newly current server.

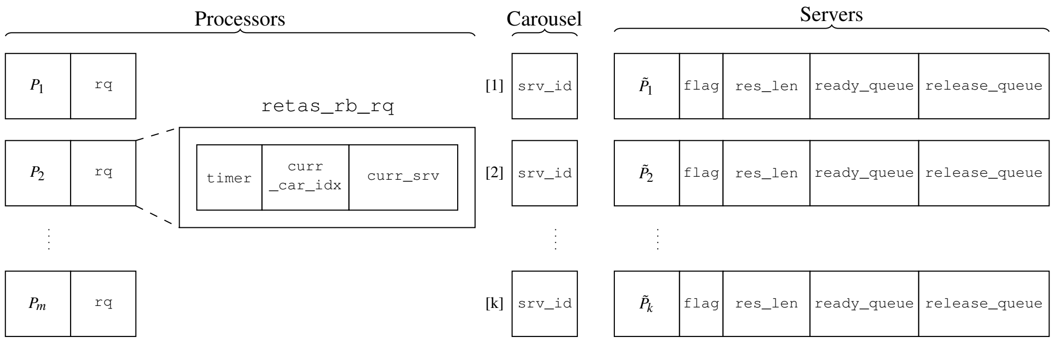


Figure 6.2: Data required to implement the Carousel-EDF scheduling algorithm in Linux kernel.

Consider again Figure 6.1. As it can be seen from that figure, during the first time slot of the processor P_4 there is some time not used by any server, the empty reserve. This continues on processor P_3 on the second time slot, and on processor P_2 on the third time slot. In practice, this unmapped reserve follows the same processor rotation pattern like real servers. Hence, from the point of view of implementation, the best option for handling this unmapped reserve is to allocate it to an *empty server*; that is, a server with a reserve length equal to the empty reserve length and without any task assigned to it. Note that, from a practical point of view, the rotation of such an empty server can be useful for the operating system, because this periodical idle time gives to the processor some available processing capacity that can be used by the operating system to execute some management stuff.

6.3.2 Releasing jobs

As mentioned before, each task generates a potentially infinite number of jobs. This generation is, usually, done either by the means of interrupts or timers. This is known as job release. From the implementation point of view, releasing a job consists in transferring a job from the release-queue to the ready-queue. Our implementation supports the job release either by interrupts or timers, however, we adopt the following release mechanism: each server also has its own release queue (a red-black tree sorted by the arrival time) and whenever a server, $\tilde{P}_q$, is allocated to one processor

```
static enum hrtimer_restart begin_reserve(struct hrtimer *timer)
{
  unsigned long long next_expire;
  struct rq *rq = cpu_rq(smp_processor_id());
  ...
  rq->retas_rb_rq.curr_car_idx = get_next_curr_car_idx(rq->retas_rb_rq.curr_car_idx);
  rq->retas_rb_rq.curr_srv      = get_server_id(rq->retas_rb_rq.curr_car_idx);
    ...
  next_expire = timer->expires + rq->retas_rb_rq.curr_srv->res_len;
  hrtimer_set_expires(timer, ns_to_ktime(next_expire);
  wake_up_jobs(&rq->retas_rb_rq.curr_srv->release_queue, &rq->retas_rb_rq.curr_srv->
      ready_queue);
  resched_task(rq->curr);
  ...
  return HRTIMER_RESTART;
}
```

Listing 6.1: Timer callback function that release all jobs whose $a_{i,j}$ is earlier that the beginning of the current reserve.

(at the beginning of its reserve) its release-queue is polled to transfer all jobs with $a_{i,j}$ smaller than or equal to the beginning of the current reserve. Further, in the case of some pending job releases fall in within the current server $\tilde{P}_q$ reserve, a timer is set.

In practice, this is performed by the top-level scheduler at timer expiration (see Listing 6.1). Thus, when a new reserve starts it updates the `curr_srv` pointer and transfers all pending jobs from the respective release- to ready-queue. Then, it triggers the invocation of the low-level scheduler by flagging the currently executing task to be preempted.

6.4 Overhead-aware schedulability analysis

As mentioned before, the Carousel-EDF comprises two algorithms: the off-line task-to-server assignment and the run-time task-dispatching algorithms. The purpose of the schedulability analysis is two-fold: (i) it checks if the task set is schedulable; and (ii) it configures the system; that is, it computes the parameters used by the run-time task-dispatching algorithm. Therefore, the schedulability analysis is used by the off-line procedure.

The first step in the off-line procedure is the partitioning of the task set into servers. Because at any time a server is allocated at most one processor, the processing demand of all tasks in a server, including overheads, must not exceed the processing capacity that can be supplied by a processor. Thus, we use the same demand-based and overhead-aware schedulability analysis used for mapping tasks to servers developed in Section 5.3.1, which ends with the task-to-server mapping presented in Algorithm 5.1. The second step consists in: (i) sizing the server reserves; (ii) creating the carousel; and finally, (iii) defining the initial phase of each processor. In order to identify the related formulation to this stage, we superscript all functions with “rb” that is an abbreviation of “reserve-based”.

6.4.1 Server inflation

Carousel-EDF guarantees that each server is assigned to a processor for the duration of its reserve every time slot (whose size is S time units); that is, tasks of a server can execute only within a periodic reserve of length $Res^{len}[\tilde{P}_q] = U^{infl}[\tilde{P}_q] \cdot S$. Hence, given a time interval of length t, only a fraction of t is supplied for the execution of a server. Thus, we can express this schedulability test as:

$$\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q,t) \leq \mathrm{sbf}^{\mathrm{rb}}(\tilde{P}_q,t), \forall t > 0 \tag{6.1}$$

Starting the derivation of $\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q,t)$ with $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$(see equation 5.10):

$$\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{rb}}(\tau[\tilde{P}_q],t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t) \tag{6.2}$$

$\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q],t)$ comprises the execution requirement of each task mapped to the server $\tilde{P}_q$, including the context switch (or task switch) as well as the release overheads.

Next, we derive the formulation taking into account the use of reserves. In order to model the execution requirement of the CPMD and interrupts, we follow the same approach used for slot-based non-split servers described in Subsection 5.3.2.1. In fact, both the slot-based non-split and the reserve-based servers use only one reserve during each time slot. Then, the worst-case cost of the CPMD in a time interval of length t is:

$$\mathrm{dbf}^{\mathrm{rb}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) \tag{6.3}$$

On a uniprocessor, each interrupt Int^i can be modelled as a sporadic interfering task with minimal inter-arrival time of T_i^{Int} and an execution time (and deadline D_i^{Int}) equal to C_i^{Int}. However, under periodic reserves, the interrupts only exert overhead when present inside the reserves of the server under consideration. Outside its reserves, an interrupt contributes to the processor-demand of some other server instead. Hence, each interrupt has to be modelled as a bursty periodic task. This was the argument used for the approach followed to model the interrupts for the slot-based task-splitting scheduling algorithm (see Equation A.8). Then, the execution demand of interrupts is given by:

$$\mathrm{dbf}^{\mathrm{rb}}_{\mathrm{IntO}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{IntO}}(\tilde{P}_q,t) \tag{6.4}$$

Hence, the execution requirement of reserve-based server is:

$$\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q,t) = \mathrm{dbf}^{\mathrm{rb}}(\tau[\tilde{P}_q],t) + \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{CpmdO}}(\tilde{P}_q,t) + \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{IntO}}(\tilde{P}_q,t) \tag{6.5}$$

Like the slot-based non-split servers, a reserve-based server $\tilde{P}_q$ is confined to execute within a single reserve of length $Res^{len}[\tilde{P}_q]$ every time slot, S. We model this reduction by amending the $\mathrm{sbf}^{\mathrm{rb}}(\tilde{P}_q,t)$ to:

$$\mathrm{sbf}^{\mathrm{rb}}(\tilde{P}_q,t) = t - \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{Fake}}(\tilde{P}_q,t) \tag{6.6}$$

where $\text{dbf}^{\text{rb}}_{\text{Fake}}(\tilde{P}_q,t)$ is computed as:

$$\text{dbf}^{\text{rb}}_{\text{Fake}}(\tilde{P}_q,t) =$$
$$\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q,t) = \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \tag{6.7}$$

where $C^{fake} = S - Res^{len}[\tilde{P}_q]$, $T^{fake} = S$, and $D^{fake} = C^{fake}$ are the attributes of the fake task that models the unavailability of each reserve. *ResL* is a upper bound for the reserve latency. However, *ResL* of the reserve-based scheduling algorithm could potentially be larger than the *ResL* of the slot-based (see Figure 5.5) due to the release mechanism adopted for the reserve-based scheduling algorithm.

Figure 6.3 illustrates the overheads associated with the switching from one reserve to another. Note that, due to the release approach followed for Carousel-EDF the reserve latency includes the release overhead of the jobs that the arrival time have elapsed. However, this overhead is already accounted for in $\text{dbf}^{\text{part}}(\tilde{P}_q,t)$. In other words, we count this specific overhead twice. The reason for that is: given the sporadic arrival nature of the tasks, we are not able to determine how many jobs are either released inside or outside of the respective server reserves.

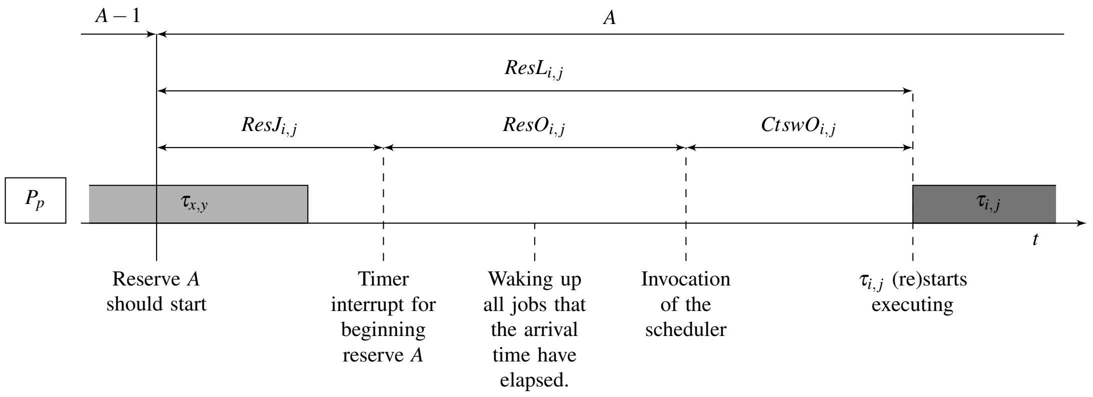

$A-1$
A
$ResL_{i,j}$
$ResJ_{i,j}$
$ResO_{i,j}$
$CtswO_{i,j}$
P_p
$\tau_{x,y}$
$\tau_{i,j}$
t
Reserve A should start
Timer interrupt for beginning reserve A
Waking up all jobs that the arrival time have elapsed.
Invocation of the scheduler
$\tau_{i,j}$ (re)starts executing

Figure 6.3: Illustration of the reserve latency. The *ResL* incorporates the release of jobs that arrival time has elapsed.

Replacing $\text{sbf}^{\text{rb}}(\tilde{P}_q,t)$ by the formulation presented in Equation 6.6 in Inequality 6.1 and moving some terms from the right-hand side to the left-hand side, we obtain the following schedulability test for reserve-based servers:

$$\text{dbf}^{\text{rb}}(\tilde{P}_q,t) + \text{dbf}^{\text{rb}}_{\text{Fake}}(\tilde{P}_q,t) \leq t, \forall t > 0 \tag{6.8}$$

We use the schedulability test in Inequality 6.8, with $\text{dbf}^{\text{rb}}(\tilde{P}_q,t)$ as given by Equation 6.5 and $\text{sbf}^{\text{rb}}(\tilde{P}_q,t)$ as given by Equation 6.6, to determine the server's inflated utilization. For that purpose, we define the `inflate_rb` function. The algorithm of the `inflate_rb` function is similar to the pseudo-code described in Algorithm 5.2. The only difference is the function `dbf_-rb_check`, which implements the schedulability test in Equation 6.1, rather than function `dbf_-sb_non_split_check`. The `dbf_rb_check` function is implemented using the QPA. Hence, the algorithm of that function is similar to the Algorithm 5.4 that shows, in pseudo-code, how the QPA technique can be used with each of the schedulability tests we have defined so far. For the `dbf_rb_check` function, $\mathrm{dbf}^{\mathrm{xxx}}$ is replaced by $\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q, t)$.

6.4.2 Generation of the carousel

The second step in the off-line procedure is the generation of the carousel. It comprises determining the size of the reserves and their sequence and also the initial phasing of the different processors.

Before describing the carousel generator algorithm, let us reason about the server inflation. The server inflation algorithm may lead to an inflated utilization close to 100% or even higher. If the inflated utilization is 100% or higher, it means that the overheads incurred by the use of reserves are higher than the unused processor capacity, and therefore the server should be assigned to a dedicated processor, becoming a single server (a single server is assigned to one dedicated processor, which is only allocated to that server, then, there is no need for reserves). Even if the inflated utilization of a server is lower than 100% it may be worth that it be assigned to a dedicated processor. This is because the size of the reserve of such a server is close to S, the period of the reserves. In this case, the uncertainty in the temporal behaviour of the operating system in general and in the time measurement in particular gives rise to the instantaneous migration problem described in Section 4.5 that may lead to a race condition in which two processors schedule the same task. To address this issue, we suggest the use of rule A2 described in Section 5.4 that states, in other words, that:

A2: if a server's inflated utilization is larger than $1.0 - ResL/S$, then the inflated utilization of the server should be set to 1.0, and the server assigned to a dedicated processor; that is, the server becomes a single server.

The rationale for this rule is to ensure that, when a processor switches to a new server reserve, the previous reserve of that server has terminated. In other words, it does not make sense to employ a reserve mechanism if the potential benefit of that is eliminated by the overheads incurred by such a mechanism. Note that, this amounts to pure partitioning for the respective tasks.

Algorithm 6.1 shows the algorithm that generates the sequence of servers. It has a single loop in which it iterates over the set of servers created by the task set partitioning algorithm (Algorithm 5.1). For each server it inflates its utilization by invoking the `inflate_rb` function, identifies single servers and assigns a sequence number to each server that determines the order of the server in the carousel.

Sequence number zero is assigned to single servers, i.e. servers that are assigned to dedicated processors and hence do not belong to the carousel. The algorithm returns true, if the set of servers is schedulable over m processors, and false otherwise.

Algorithm 6.1: Pseudo-code algorithm of reserve sequence generator.

```
Input: set of $k$ servers $\tilde{P}_q$, with $1 \leq i \leq k$
        $\varepsilon$  {accuracy in server inflation}
        $m$  {number of processors}
Output: boolean {true if schedulable}
        $\Gamma[]$ {server sequence}

$S \leftarrow \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$ {time slot}
$s \leftarrow 1$ {sequence order}
$U \leftarrow 0$ {system utilization}
for $q \leftarrow 1$ to $k$ do
    {inflate servers}
    $t \leftarrow 2 \cdot \mathrm{lcm\_T}(\tilde{P}_q)$
    $U^{infl}[\tilde{P}_q] \leftarrow \mathrm{inflate\_rb}(\tilde{P}_q, \varepsilon, t)$
    if $U^{infl}[\tilde{P}_q] \geq 1.0 - ResL/S$ then
        $U^{infl}[\tilde{P}_q] \leftarrow 1.0$
        $Tp[\tilde{P}_q] \leftarrow$ SINGLE
        $\Gamma[q] \leftarrow 0$ {0 means not in sequence}
    else
        $\Gamma[q] \leftarrow s$
        $s \leftarrow s + 1$
    end if
    $U \leftarrow U + U^{infl}[\tilde{P}_q]$
end for
return $U \leq m$
```

To complete the generation of the carousel, we need to determine the initial phasing of the carousel in each processor, i.e. to determine the first reserve for each processor. Let r be the number of processors used in the carousel. As illustrated in Figure 6.1, the schedule of each processor must be such that processor P_p is S time units ahead of that of processor P_{p-1}, modulo r. Therefore, for each of the r processors, we need to determine the server of its first reserve and its duration. Algorithm 6.2 shows the pseudo-code of an algorithm that computes these parameters. It takes as inputs the carousel, including the server parameters, the number of processors r, whose identifier is assumed to range from 1 to r. The algorithm has one single loop in which it iterates over the carousel (and also the r processors used to run the carousel). It then determines the servers that are active in the first processor at each multiple of the time slot S, and the time left at that point until the end of the reserve, and assigns these values as the parameters of the first reserve of the corresponding processor.

6.4.3 Supporting other job release mechanisms

As mentioned before, the Carousel-EDF scheduler implementation also supports the release of jobs by timer or interrupts. Using the criterion that jobs of a task can be released by any processor used by that task, this means that, under Carousel-EDF any job may be released by any processor. In that case, we have to compute the execution demand of a server taking into account the release interference of the other servers. The release interference $\mathrm{dbf}^{\mathrm{rb}}_{\mathrm{RelI}}(\tilde{P}_q, t)$ in a time interval of length t is computed as:

$$\mathrm{dbf}^{\mathrm{rb}}_{\mathrm{RelI}}(\tilde{P}_q, t) = \sum_{i=1 \wedge i \neq q}^{k} \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_i, t) \tag{6.9}$$

Algorithm 6.2: Pseudo-code algorithm of first reserve generator

```
Input: set of $k$ servers $\tilde{P}_q$, with $1 \leq i \leq k$
       $\Gamma[]$ {server sequence}
       $n$ {number of processors running the carousel}
Output: $\rho[]$ {server of first reserve of each processor}
        $\Phi[]$ {duration of first reserve of each processor}

$p \leftarrow 1$ {processor index}
$S \leftarrow \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$ {time slot}
$t \leftarrow 0$ {accumulated schedule time}
for $q \leftarrow 1$ to $k$ do
  if $\Gamma[q] <> 0$ then
    {only servers that belong to the carousel}
    $t \leftarrow t + U^{infl}[\tilde{P}_q] \cdot S$ {expiration time of this reserve}
    if $t \geq (p-1) \cdot S$ then
      {reserves crosses time slot boundary}
      $\rho[p] \leftarrow q$ {assign first server}
      $\Phi[p] \leftarrow t - (p-1) \cdot S$ {duration of first reserve}
      $p \leftarrow p+1$
    end if
  end if
end for
```

where $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_i, t)$ gives the release interference of a server (see Equation 5.9) and $\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q, t)$ (see Equation 6.5) is amended to:

$$\mathrm{dbf}^{\mathrm{rb}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{rb}}(\tau[\tilde{P}_q], t) + \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{IntO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{rb}}_{\mathrm{RelI}}(\tilde{P}_q, t) \tag{6.10}$$

Further, in this case, all tasks could incur the IPI latency, $IpiL$, as in the case of split tasks under slot-based semi-partitioned scheduling (see Equation 5.30). Then, the $\mathrm{dbf}^{\mathrm{rb}}(\tau[\tilde{P}_q], t)$ has to be modified to incorporate such overhead, as:

$$\begin{aligned}
&\mathrm{dbf}^{\mathrm{rb}}(\tau[\tilde{P}_q], t) = \\
&\quad \mathrm{dbf}^{\mathrm{sb:split}}(\tau[\tilde{P}_q], t) = \\
&\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \\
&\quad \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ + IpiL}{T_i} \right\rfloor + 1\right) \cdot (C_i + RelO + 2 \cdot CtswJ)
\end{aligned} \tag{6.11}$$

6.5 Utilization bound

The utilization bound of an algorithm is a simple performance metric used to compare scheduling algorithms. In this section, we present the utilization bound of the Carousel-EDF scheduling algorithm based on the utilization factor. By design, Carousel-EDF offers, for the same value of corresponding parameter δ, the same utilization bound as the original NPS-F algorithm. This bound is given by the following expression:

$$\frac{2 \cdot \delta + 1}{2 \cdot \delta + 2}$$

The following reasoning establishes this.

Lemma 1. *A set of k servers is schedulable on m processors under Carousel-EDF if and only if*

$$\sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \leq m$$

Proof. This follows from the fact that Carousel-EDF successfully schedules all servers if and only if it provides $U^{infl}[\tilde{P}_q] \cdot S$ time units to each server $\tilde{P}_q$ within every time window of length S (equal to the time slot). This processing time is provided to the k servers from the m processors (by design, in such a manner that each server only receives processing time from at most one processor at any time instant). □

Lemma 2. *After the bin-packing stage of Carousel-EDF, it holds that* $\frac{1}{k}\sum_{q=1}^{k} U[\tilde{P}_q] > \frac{1}{2}$*, for the k utilized bins (servers).*

Proof. This follows from the fact that Carousel-EDF successfully schedules all servers if and only if it provides $U^{infl}[\tilde{P}_q] \cdot S$ time units to each server $\tilde{P}_q$ within every time window of length S (equal to the time slot). This processing time is provided to the k servers from the m processors (by design, in such a manner that each server only receives processing time from at most one processor at any time instant). □

Lemma 3. *: For any server* $\tilde{P}_q$*, it holds that* $U^{infl}[\tilde{P}_q] \leq \text{inflate}(U[\tilde{P}_q])$.

Proof. This holds because Carousel-EDF inflates each server $\tilde{P}_q$ to the minimum $U^{infl}[\tilde{P}_q]$ sufficient for schedulability, according to an exact schedulability test. In contrast, the inflate function used by NPS-F potentially over-provisions. □

Now we can prove the utilisation bound of Carousel-EDF. The following theorem and its proof are analogous to Theorem 3 from [BA09b], which proved the utilisation bound of NPS-F.

Theorem 1. *The utilization bound of Carousel-EDF is* $\frac{2 \cdot \delta + 1}{2 \cdot \delta + 2}$.

Proof. An equivalent claim is that every task set with utilization not greater than $\frac{2 \cdot \delta + 1}{2 \cdot \delta + 2} \cdot m$ is schedulable under Carousel-EDF. We will prove this equivalent claim.

From Lemma 1, Carousel-EDF successfully maps all k servers to a feasible online schedule if and only if

$$\sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \leq m$$

Applying Lemma 3 to the above, a sufficient condition for the schedulability of all k servers is

$$\sum_{q=1}^{k} \text{inflate}(U[\tilde{P}_q]) \leq m \tag{6.12}$$

For the function $\text{inflate}(U) = \frac{(\delta+1)\cdot U}{U+\delta}$, it holds that $\frac{d}{dU}\text{inflate}(U) > 0$ and $\frac{d^2}{dU^2}\text{inflate}(U) < 0$ over the interval [0,1]. Therefore, from Jensen's Inequality [Jen06], we have:

$$\sum_{q=1}^{k} \text{inflate}(U[\tilde{P}_q]) \leq k \cdot \text{inflate}(\bar{U})$$

where

$$\bar{U} = \frac{1}{k}\sum_{q=1}^{k} U[\tilde{P}_q])$$

Hence, combining this with the sufficient condition of Inequality 6.12, a new sufficient condition for the successful mapping of the k servers is

$$k \cdot \text{inflate}(\bar{U}) \leq m \tag{6.13}$$

Now, let the function $\alpha(U)$ denote the expression $\text{inflate}(U) - U$. Then, $\alpha(U) = \frac{U\cdot(1+U)}{U+\delta}$ and Inequality 6.13 can be rewritten as

$$k \cdot (\bar{U} + \alpha(\bar{U})) \leq m \tag{6.14}$$

Given that $\alpha(U)$ is a decreasing function of U over $[\frac{1}{2}, 1]$ it holds that

$$\frac{\alpha(U)}{U} < \frac{\alpha(\frac{1}{2})}{\frac{1}{2}} = \frac{1}{2\delta+1}, \; \forall U \in \left(\frac{1}{2}, 1\right] \Rightarrow \alpha(U) < \frac{1}{2\delta+1} \cdot U \; \forall U \in \left(\frac{1}{2}, 1\right]$$

Combining the above with the fact that (from Lemma 2) $\bar{U} > \frac{1}{2}$, we obtain from Inequality 6.13 the following new sufficient condition:

$$k \cdot \left(\bar{U} + \frac{1}{2\delta+1} \cdot \bar{U}\right) \leq m \tag{6.15}$$

In turn, this can be equivalently rewritten as:

$$k\bar{U} \leq \frac{2\delta+1}{2\delta+2} m \tag{6.16}$$

But the left-hand side of the above inequality, corresponds to the cumulative utilization of all tasks in the system. Therefore, if $\sum_{\tau_i \in \tau} u_i \leq \frac{2\delta+1}{2\delta+2} m$, this is a sufficient condition for schedulability under Carousel-EDF. This proves the theorem. □

6.6 Upper bounds on preemptions

Under either of the two approaches to server mapping formulated in [BA09b], for NPS-F, a upper bound $\mathrm{nr}_{\mathrm{pree}}^{\mathrm{NPS-F}}$ on the preemptions (including migrations) related to the reserves within a time interval of length t is given by:

$$\mathrm{nr}_{\mathrm{pree}}^{\mathrm{NPS-F}} < \left\lceil \frac{t}{S} \right\rceil \cdot 3 \cdot m \tag{6.17}$$

This follows from the fact that, within each time slot of length S, other that preemptions due to EDF, there occur: (i) one preemption per-server (hence k in total), when its last (or only) reserve runs out; plus (ii) $m-1$ (at most) migrations, corresponding to the case when some server migrates between a pair of successively indexed processors.

Therefore

$$\mathrm{nr}_{\mathrm{pree}}^{\mathrm{NPS-F}} = \left\lceil \frac{t}{S} \right\rceil \cdot k + \left\lceil \frac{t}{S} \right\rceil \cdot (m-1) \tag{6.18}$$

and the bound of Inequality 6.17 is derived from the fact that $m-1 < m$ and also, as proven by Corollary 1 in [BA09b], $k < 2 \cdot m$.

By comparison, Carousel-EDF generates within each time slot, at most k preemptions/migrations (other than those due to EDF). Each of those corresponds to the ending of the reserve of the corresponding server. Therefore

$$\mathrm{nr}_{\mathrm{pree}}^{\mathrm{Carousel-EDF}} = \left\lceil \frac{t}{S} \right\rceil \cdot k < \left\lceil \frac{t}{S} \right\rceil \cdot 2 \cdot m \tag{6.19}$$

By comparing the respective bounds (Equations 6.17 and 6.19), we observe that the upper bound for preemptions due to the time division under Carousel-EDF is 33.3% smaller than that under NPS-F.

However, the above upper bounds are pessimistic with respect to the value of k (i.e. assume that k is close to $2m$, which is the unfavourable scenario, in terms of preemptions/migrations generated). In fact, the number of servers k may range from $m+1$ to $2m-1$; if it were less, we would use partitioning and it could not be $2m$ or higher from Lemma 2 in Section 6.5. In many cases therefore, k could in fact be close to m. And, by inspection (see Equation 6.18, for $k = m+1$ and $m \to \infty$, the reduction in the upper bound on reserve-related preemptions/migrations when using Carousel-EDF, as compared to NPS-F, would tend towards 50%. Therefore, Carousel-EDF involves 33.3% to 50% fewer reserve-related preemptions and migrations than NPS-F.

6.7 Summary

In this chapter, we proposed a new scheduling algorithm for multiprocessor systems, called Carousel-EDF. It presents some similarities with the slot-based task-splitting scheduling algorithms under consideration in this dissertation. It also requires two algorithms: an off-line task-to-processor algorithm and a run-time task-dispatching algorithm. The schedulability analysis, which guides the off-line algorithm, is also processor demand-based and overhead-aware. Using such schedulability analysis, the off-line algorithm maps the tasks to servers, computes the server reserve lengths, defines the cycling server order, and defines the initial phasing of each processor. The run-time dispatching algorithm divides the time into reserves that are mapped to servers. Such server reserves rotate through all processors, creating a carousel, in such a way that no two processors are allocated the same server simultaneously. The tasks of each server are scheduled according to EDF scheduling policy.

One question arises: how should Carousel-EDF be categorised? Is it global, partitioned, or semi-partitioned? We observe that it does not easily fit in this taxonomy. It is a global algorithm in that it allows the migration of any task, yet it can be implemented with contention-free ready queues, like partitioned algorithms. Furthermore, like partitioned algorithms, it partitions the task set into servers, such that tasks from one server do not directly contend for processing resources with tasks from other servers. On the other hand, in many respects, Carousel-EDF is closer to NPS-F [BA09b], a semi-partitioned algorithm from which it evolved. In particular, the concept of server was inherited from NPS-F, where it is called notional processor. Furthermore, servers, like notional processors, are allocated periodic reserves [AB08], time intervals during which a processor is exclusively allocated for running tasks of the corresponding server. NPS-F has many attractive features: its utilization bounds range from 75% to 100%, by trading off with preemptions. Yet, it has lower upper bounds on preemptions than any known algorithm for the same utilization. Carousel-EDF preserves the utilization bounds of NPS-F, including their trade-off with preemptions, and further reduces the upper bounds on preemptions. This is achieved by never splitting a server. As a result, it is able to reduce the maximum number of preemptions, and therefore context switches, introduced by server reserves to half of those of NPS-F.

An additional benefit of not splitting reserves is that the overhead-aware schedulability analysis of Carousel-EDF presented here is much simpler than that of the slot-based task-splitting scheduling algorithm described in Chapter 5 on which it is based.

Chapter 7

Evaluation

7.1 Introduction

As mentioned before, the scheduling algorithms under consideration in this work comprise two algorithms. An off-line task-to-processor assignment algorithm (in the form of task-to-server and server-to-processor) and a run-time task-dispatching algorithm. In this chapter, we evaluate and compare both algorithms.

7.2 The off-line schedulability analysis

In this section, we apply the new schedulability theory developed so far to three studies. The goal of the first study is to compare the new slot-based task-splitting scheduling algorithm theory with the utilization-based schedulability theory originally proposed. The goal of the second study is to show the practicability of the new schedulability theory when we consider overheads. Finally, the third study compares the slot-based task-splitting against reserve-based scheduling algorithms.

7.2.1 Quantification of overheads

In order to account for the effect of scheduling overheads using the new theory, worst-case estimates for the various overheads themselves are required as input to the analysis. However, upper bounds on the worst-case values of the previously identified overheads cannot be determined via a purely analytical approach because they depend in complex ways on the characteristics of both the hardware and software, including the operating system, that are rarely documented with sufficient detail. Therefore, our approach was to experimentally measure (and log) the overheads of 100 randomly generated task sets, scheduled during 1000 seconds each, first under S-EKG and then under NPS-F. The corresponding maximum recorded values were then treated as safe upper bounds for the respective overheads. Although, arguably, there is always the possibility that worse values might be observed if the experiment ran for more time, we deem this level of accuracy sufficient for our purposes in this study. For a more detailed study, or in practice, the number of

Table 7.1: Experimentally-derived values for the various scheduling overheads (in microseconds).

	RelJ	*RelO*	*ResL*	*CtswO*	*IpiL*
Measured	17.45	8.56	30.24	35.21	19.30
Used	20.00	10.00	40.00	40.00	20.00

measured values will likely vary, and depend on such factors as the variability of the measured parameters or the level of safety required.

The 24-core platform used in our experiments is built from 1.9 GHz AMD Opteron 6168 chips [Dev] running at a frequency of 1.9 GHz. Each Opteron 6168 module has 12 cores and occupies one socket on the motherboard. The operating system was the modified 2.6.31 Linux kernel [SBTA11].

All parameters were determined in a way consistent to their definition in Section 4.7. The context switch overhead is measured from the time instant the scheduler starts executing until the time instant when it calls the assembly routine that switches from the current executing job to the new one. To determine the release jitter, we measured the time interval between the (theoretical) job arrival time and the time instant when the timer actually expires, i.e., when the timer callback is invoked. The release overhead was determined by measuring the time interval between the time instant when the timer callback is invoked and a task removed from the release-queue is inserted into the ready-queue. The reserve latency was estimated by measuring the time interval from the time at which a reserve should (theoretically) start until the time instant when a ready job (if one exists) starts to execute within the reserve. Finally, we measured the IPI latency as the time interval between the generation of the interrupt (by the emitting processor) and the time instant the corresponding handler starts executing (on the other processor).

Table 7.1 presents the values of these parameters determined experimentally and the rounded-up values that were used as input to our experimental evaluation of the demand-based overhead-aware schedulability analysis. Essentially, we took a pessimistic stance and derived the values by rounding up the maximum values measured for each of the parameters.

Other than the various earlier overheads identified, we also collected measurements for the tick interrupt, which occurs on every processor. This is a periodic interrupt used by the operating system kernel for triggering various operations such as the invocation of the scheduler. The worst-case execution time measured for this interrupt was 8.06 microseconds. Although its periodicity (approximately one milliseconds in our setup) can be configured via the Linux kernel macro `HZ`, in practice this interrupt suffers from jitter. We estimated this jitter, by comparing the recorded inter-arrival times with the reference period, as 177 microseconds. These values were obtained with a Linux kernel compiled with both the tickless option (for suppressing the tick interrupts during idle intervals) and the CPU frequency scaling features disabled.

We did not derive estimates for overheads from any interrupts other than the tick interrupt because all other interrupts can be configured to be managed by one specific processor (preferably, the least utilized one). Hence, we deemed that, even if we would have gone through that effort, their inclusion would not meaningfully change the overall picture. However, our analysis still allows the overheads related to any interrupt to be specified as input and factored in.

Determining CPMD is a challenging research problem that exceeds the scope of this work. For the state-of-the-art, see [Bas11, ADM12, LAMD13, DSA$^+$13]. Nevertheless, our new schedulability theory allows the incorporation of such effects. In the study with overheads, which we present below, we have chosen an arbitrary value of 100 microseconds for $CpmdO$.

Although, in a strict sense, the measurement-based estimates characterize only the system in which the measurements were made, we believe that this particular contribution is important for the following reasons. First, it shows the feasibility of the new analysis, which in turn further validates the slot-based task-splitting approach for multiprocessor scheduling as practical and efficient. Second, by documenting how we derived the measurement-based estimates in a manner consistent with the earlier definitions of the respective overheads, it is possible to re-use the same approach in order to derive estimates for the overheads in different systems.

7.2.2 Experiment design

Given the goals of our studies, we have chosen as metric the normalized inflated system utilization, which is defined as follows:

$$U_s^{infl} = \frac{1}{m} \cdot \sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \qquad (7.1)$$

where m is the number of processors in the system, k is the number of servers, and $U^{infl}[\tilde{P}_q]$ is the inflated utilization of each of the servers. A system is considered unschedulable whenever this metric exceeds 1.0.

To compare schedulability analyses, this metric is used indirectly: a schedulability analysis is better than another, if, for a given set of task sets, the former finds that more task sets are schedulable than the latter. Likewise, this metric can be used indirectly in the evaluation of a schedulability algorithm: an algorithm is better than another if it is able to schedule more task sets.

In our studies, we consider different types of task sets. We characterize each task set by its normalized utilization and by the characteristics of its tasks. Because, we use a synthetic load, generated randomly using a unbiased random number generator, rather than specifying a single value for the task set normalized utilization, we use an interval with minimum value $U_{s:min}$ and width inc, $[U_{s:min}, U_{s:min} + inc)$. With respect to the characteristics of the tasks, the period of each task T_i, is uniformly distributed over $[T_{i:min}, T_{i:max})$. All tasks generated are implicit deadline ($D_i = T_i$) in order to allow comparisons with the original analysis. The worst-case execution time of a task, C_i, is derived from T_i and the task's utilization, u_i, which is also uniformly distributed over $[u_{i:min}, u_{i:max})$.

Algorithm 7.1 shows the pseudo-code of the task set generation procedure. It takes as inputs the minimum normalized system utilization, $U_{s:min}$, the granularity of the normalized system utilization of each task set, inc, the number of task sets, n^τ, the minimum and maximum values of the utilization of each task in all task sets, $u_{i:min}$ and $u_{i:max}$, respectively, the minimum and maximum values of the period of each task in all task sets, $T_{i:min}$ and $T_{i:max}$, respectively, and the number of processors in the system, m. The output of this procedure is n^{τ} task sets which are put in array Γ. The normalized system utilization of task set $\Gamma[j]$ (for j between 1 and n^{τ}) is in the range $[U_{s:min} + (j-1) \cdot inc,\ U_{s:min} + j \cdot inc)$, and the parameters of each task in these sets satisfy the values specified in the inputs of the procedure.

Algorithm 7.1: Pseudo-code algorithm of the task set generator.

```
Input: $U_{s:min}$, minimum normalized system utilization
        $inc$, granularity of the normalized system utilization of the task sets
        $n^{\tau}$ number of task sets to generate
        $[u_{i:min}, u_{i:max}]$ range of the utilization of each task to generate
        $[T_{i:min}, T_{i:max}]$ range of the period of each task to generate
        $m$ number of processors/cores
Output: $\Gamma[n^{\tau}]$ set of generated task sets

$U_{s:max} \leftarrow U_{s:min} + inc$
for $j \leftarrow 1$ to $n^{\tau}$ do
    $\Gamma[j] \leftarrow \emptyset$
    $i \leftarrow 0$
    $\tau \leftarrow \emptyset$
    $U_s \leftarrow 0$
    $in\_range \leftarrow$ false
    while $U_s < U_{s:max}$ do
        {generate a task}
        $u_i \leftarrow$ uniform($u_{i:min}, u_{i:max}$)
        $T_i \leftarrow$ uniform($T_{i:min}, T_{i:max}$)
        $C_i \leftarrow T_i \cdot u_i$
        $\tau_i \leftarrow$ create_task($T_i, C_i$)
        {add task to task set}
        add_task_to_taskset($\tau_i, \tau$)
        $U_s \leftarrow U_s + u_i/m$
        $i \leftarrow i + 1$
        if $U_{s:min} \leq U_s < U_{s:max}$ then
            {Done with this task set: it has the target utilization}
            add_taskset_to_set($\tau, \Gamma[j]$)
            $in\_range \leftarrow$ true
            break
        end if
    end while
    if $\neg in\_range$ then
        {Abort current task set generation: utilization is above target}
        $j \leftarrow j - 1$ {Try again}
    else
        {Generated task set successfully. Update target range utilization for next task set}
        $U_{s:min} \leftarrow U_{s:max}$
        $U_{s:max} \leftarrow U_{s:min} + inc$
    end if
end for
```

In all experiments, we used $U_{s:min} = 0.75$, $inc = 0.001$ and $n^{\tau} = 250$, allowing us to evaluate the algorithms for a fairly loaded system, i.e. whose load has normalized utilization between 75% and 100%. Because of the random nature of the load used, we ran each experiment ten times initializing the pseudo-random number generator with different seed values. Thus, we generated 2500 task sets.

Indeed, for systems with a lighter load, we would expect no major differences, as all task sets would most likely be schedulable. To evaluate the effect of different types of tasks, we consider four classes of task sets according to the utilization of their tasks:

- *Heavy*: Tasks whose u_i is in the range [0.65, 0.95).
- *Medium*: Tasks whose u_i is in the range [0.35, 0.65).
- *Light*: Tasks whose u_i is in the range [0.05, 0.35).
- *Mixed*: Tasks whose u_i is in the range [0.05, 0.95).

Independently of their utilization, the periods of all tasks of all task sets are uniformly distributed in the range [5 , 50] milliseconds, with a resolution of one millisecond. In Appendix B, we deal with other periods.

Finally, in all experiments, we set m equal to 24, the number of processors in the system we used to measure the overheads.

For the sake of clarity, in the plots presented in next subsections, task sets were grouped together into utilization classes (each consisting of 100 task sets). For example, all task sets with U_s in the range [0.75, 0.76) are in one class; all task sets with U_s in the range [0.76, 0.77) are in the next class and so on. The values shown in the plots are the average of each class.

Given the goals of this study, we have chosen as metric the normalized inflated system utilization $U_s^{infl} = \frac{1}{m} \cdot \sum_{q=1}^{k} U^{infl}[\tilde{P}_q]$ (see Equation 7.1). A schedulability analysis is more efficient than another, if its normalized system inflated utilization is lower. Moreover, other metrics are used to complement the information given by normalized system inflated utilization metric.

7.2.3 Preliminary evaluation of the merits of slot-based task-splitting over partitioning, when overheads are considered

However, before going through the evaluation study, we believe that is important to assess the benefit of using slot-based task-splitting scheduling algorithms against a simpler approach like Partitioned-EDF (P-EDF).

First, let us note that the algorithms under consideration *dominate* P-EDF, even when overheads are taken into account. This is because they schedule tasks within each server using the EDF scheduling policy. Moreover, they also use partitioning, but they partition tasks to servers, not to processors. But, the overhead-aware task-to-server assignment algorithm employed by slot-based task-splitting scheduling algorithms (defined by Equation 5.6) treats each server as a uniprocessor (i.e. without the overheads related to the reserves migrations and other related issues). This is equivalent to attempting to assign the task set using P-EDF. The difference is that, if more than m bins are needed, P-EDF will declare failure (assuming a system composed by m processors), whereas, with e.g. NPS-F there is still a chance that those task sets for which P-EDF fails will be schedulable. On the other hand, any task set using no more than m bins can still be schedulable with NPS-F using servers. (In the worst case, the overhead-aware server-to-processor mapping algorithm will assign each server on a dedicated processor, i.e. partitioning.)

Hence, it is, by design, impossible for slot-based task-splitting to do worse than P-EDF (with or without overheads). The question then becomes, just by how much the schedulability improves and whether this justifies the added complexity. An answer to this question can be given by the

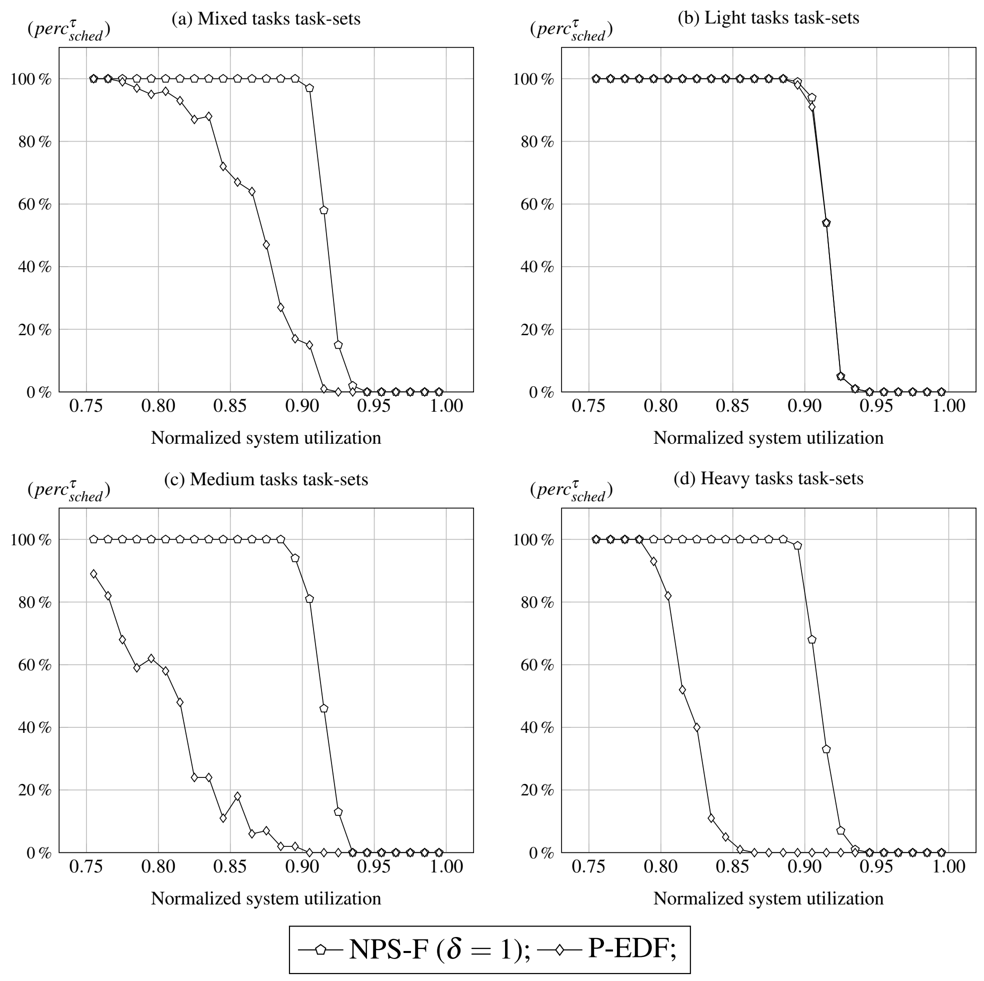


Figure 7.1: Comparison of between NPS-F and P-EDF scheduling algorithms in the presence of overheads.

plots presented by Figure 7.1. Those plots shows schedulability success ratio. This metric gives for each utilization class (recall that each utilization class is composed by 100 task sets in a range of 0.01) the percentage of task sets that are schedulable and is computed as:

$$perc^{\tau}_{sched} = \frac{nr^{\tau}_{sched}}{nr^{\tau}_{sched} + nr^{\tau}_{unsched}} \cdot 100 \tag{7.2}$$

where nr^{τ}_{sched} is the number of task sets schedulable while $nr^{\tau}_{unsched}$ is the number of task sets unschedulable. The sum of $nr^{\tau}_{sched} + nr^{\tau}_{unsched}$ gives the total number of task sets.

In Figure 7.1, we show a schedulability comparison between NPS-F and P-EDF in the presence of overheads. NPS-F considerably outperforms P-EDF for all types of task sets, except for the task sets composed by light tasks where its lead is smaller. This is because P-EDF performs very well for such task sets to start with (since it is easier to partition a task set the smaller the maximum task utilization is).

Hence, these results establish that, even when considering overheads, slot-based task-splitting is a valid option for real-time systems, with better performance than the traditional partitioning.

7.2.4 Evaluation of the new analysis in the absence of overheads

As a first step in the evaluation of the new analysis, we compare it to the original analysis published for both algorithms, so as to evaluate the improvements in processor utilization that stem from having less pessimism in the new analysis. Because the original analysis, utilization-based, assumes no scheduling overheads, the results presented in this subsection were obtained considering all overheads equal to zero.

7.2.4.1 Experiments for the S-EKG scheduling algorithm

In order to apply the new analysis to S-EKG, we employed the task-to-processor mapping algorithm outlined in Section 5.4.2.1. A major difference between this algorithm and the original S-EKG algorithm is that it does not cap the utilization of each processor to the theoretical utilization bound. $\mathrm{UB}_{\mathrm{S-EKG}}$, but rather uses the new schedulability tests presented in Section 4.7.

In our study, we considered the effect of the S-EKG design parameter δ, in addition to the workload itself, because in the original analysis this parameter has a major influence on the system utilization. Thus, for each workload, i.e. task set, we computed the normalized utilization for each of the following δ values: one, two, four, and eight.

Figure 7.2 provides a comparison between the original (utilization-based) and the new (processor demand-based) theory, for task sets generated under the mixed setup ($u_i \in [0.05, 0.95)$ for different values of δ.

As mentioned before, each point in the plots shown in this subsection represents an average of the normalized utilization for 100 randomly generated task sets, satisfying the corresponding parameter values.

As shown in inset (a) of Figure 7.2, many task sets of relatively low utilization are not schedulable according to the original analysis even with higher values for δ. The results are completely different when we apply the new schedulability analysis (see inset (b) of Figure 7.2), with almost all task sets being schedulable even with δ equal to one (the most preemption-light setting). Furthermore, the effect of δ on the schedulability of the task sets is much lower than in the original analysis. Indeed, the original S-EKG schedulability test fails for all task sets with δ equal to one. The explanation is that the original S-EKG task-to-processor assignment algorithm caps the utilization of each processor to the theoretical utilization bound ($\mathrm{UB}_{\mathrm{S-EKG}}$), and for δ equal to one, $\mathrm{UB}_{\mathrm{S-EKG}} \approx 0.65$, which is less than the lowest U_s, 0.75, of any task set used in the experiments.

As a complement to the information given by the average of the inflated utilization metric, $U^{infl}[\tilde{P}_q]$, insets (c) and (d) of Figure 7.2 shows schedulability success ratio. As it can be seen from Figure 7.2 there is a correspondence between the average inflated utilization metric and the percentage of schedulable task sets, $perc^{\tau}_{sched}$; that is, if the average inflated utilization metric indicates that some class of task sets is schedulable, the schedulability success ratio shows that most

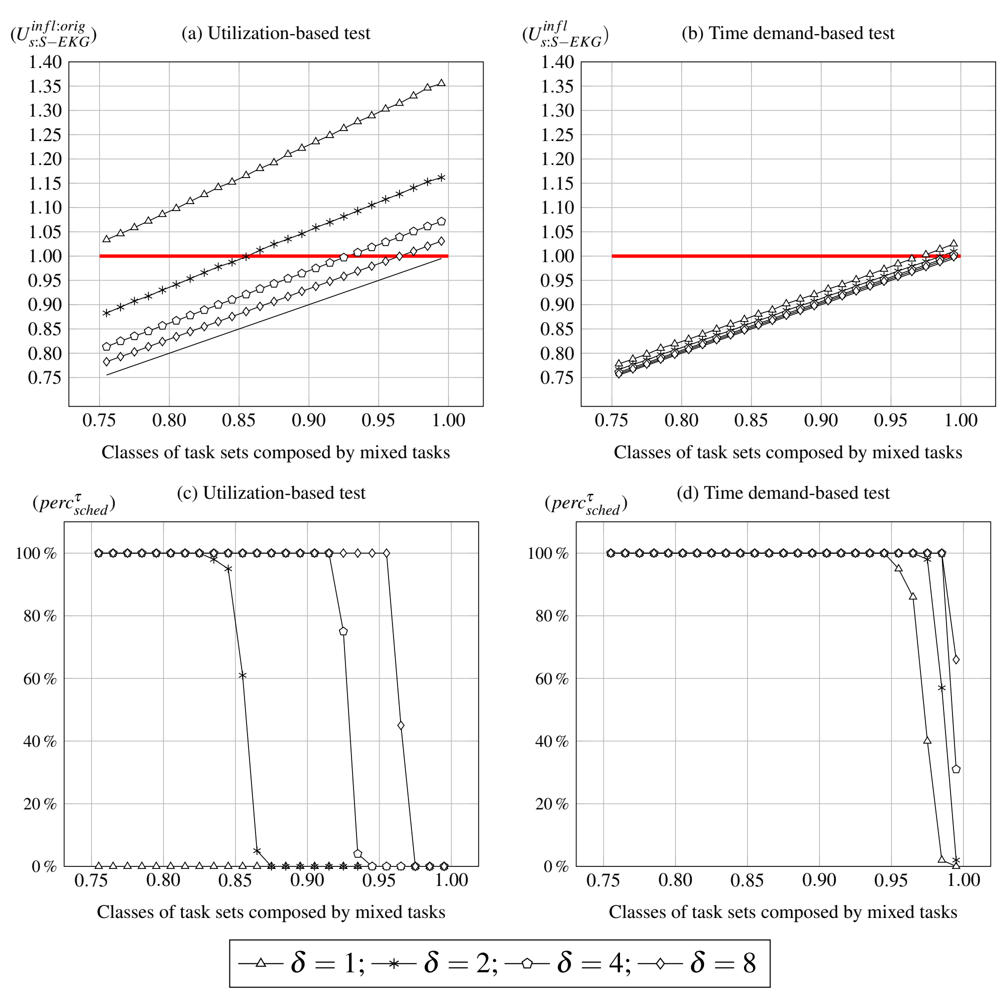


Figure 7.2: Comparison between the original S-EKG and the new schedulability analysis for task sets composed by mixed, $u_i \in [0.05, 0.95)$, tasks.

of those task sets are schedulable, or the contrary. Observing insets (a) and (c) shows that there is a relation between the average inflated utilization and the percentage of task sets schedulable for the results produced by utilization-based schedulability analysis. The same can be observed, insets (b) and (d), for the results produced by the time demand-based schedulability analysis

Figure 7.3 further highlights the benefits of the new schedulability analysis. It compares the results of the new analysis with those of the original analysis for task sets generated according to the light, medium and heavy parameter setup. The same conclusions as before apply. The new analysis clearly improves the inflation efficiency, in all cases. The improvement is so large that the inflated utilization is, at all points in the graph, very close to the uninflated utilization (drawn using a line without any mark) even for δ equal to one. Further, it is also clear that the effect of δ (in other words, the need for a high δ in order to keep inflation low) decreases. In the next subsection, we discuss the results according to the type of tasks together with the results of the NPS-F.

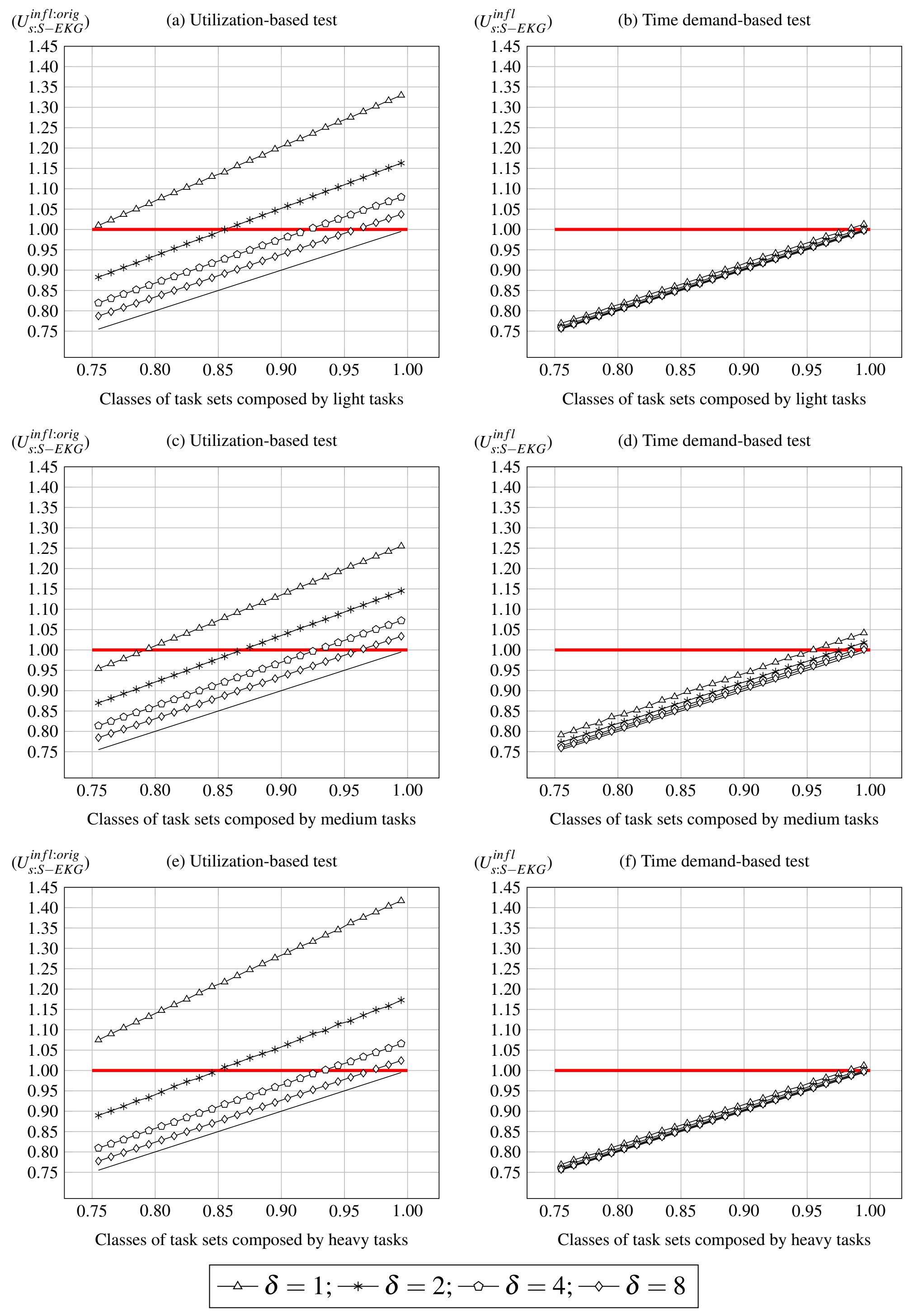


Figure 7.3: Comparison between the original S-EKG and the new schedulability analysis considering task sets composed by light, $u_i \in [0.05, 0.35)$, medium, $u_i \in [0.35, 0.65)$, and heavy tasks, $u_i \in [0.65, 0.95)$.

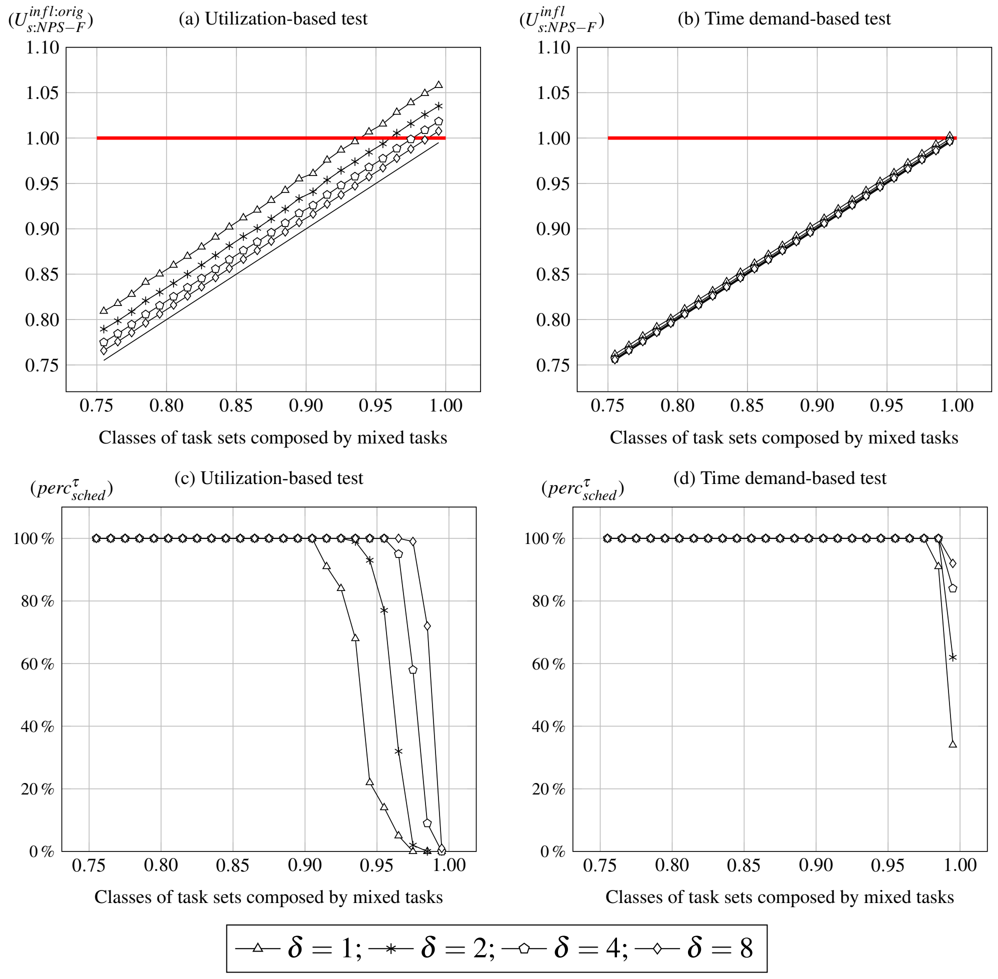


Figure 7.4: Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by mixed, $u_i \in [0.05, 0.95)$, tasks.

7.2.4.2 Experiments for the NPS-F scheduling algorithm

We performed the same set of experiments using NPS-F rather than S-EKG. Figure 7.4 compares the original, inset(a), and the new, inset(b), schedulability analysis for task sets generated under the mixed setup, $u_i \in [0.05, 0.95)$, which demonstrates a considerable improvement in mapping efficiency. In fact, using the new analysis, the points for the inflated and uninflated utilization almost coincide in the graph (even for δ equal to one).

Again, we complement the information given by the average of the inflated utilization metric, $U^{infl}[\tilde{P}_q]$, with information related to the percentage of schedulable task sets, $perc^{\tau}_{sched}$, insets (c) and (d). As it can be seen from Figure 7.4 there is a correspondence between two metrics.

These observations also apply to the experiments with the light, medium and heavy task utilization setup, shown in Figure 7.5.

Focusing on the results from the new schedulability analysis, it is clear that NPS-F produces better results than S-EKG (please note that the y-axis does not have the same scale among all

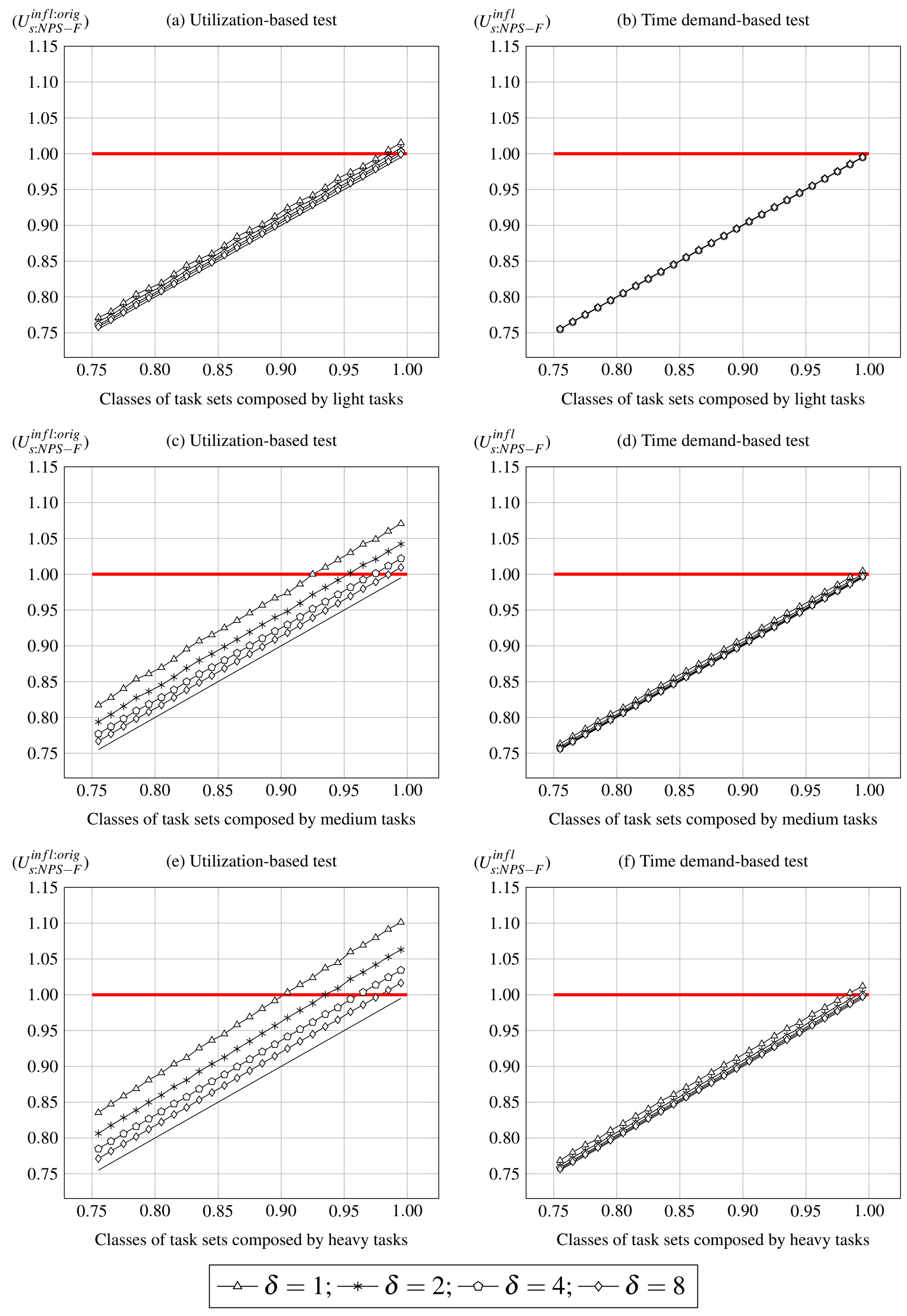


Figure 7.5: Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by light, $u_i \in [0.05, 0.35)$, medium, $u_i \in [0.35, 0.65)$, and heavy tasks, $u_i \in [0.65, 0.95)$.

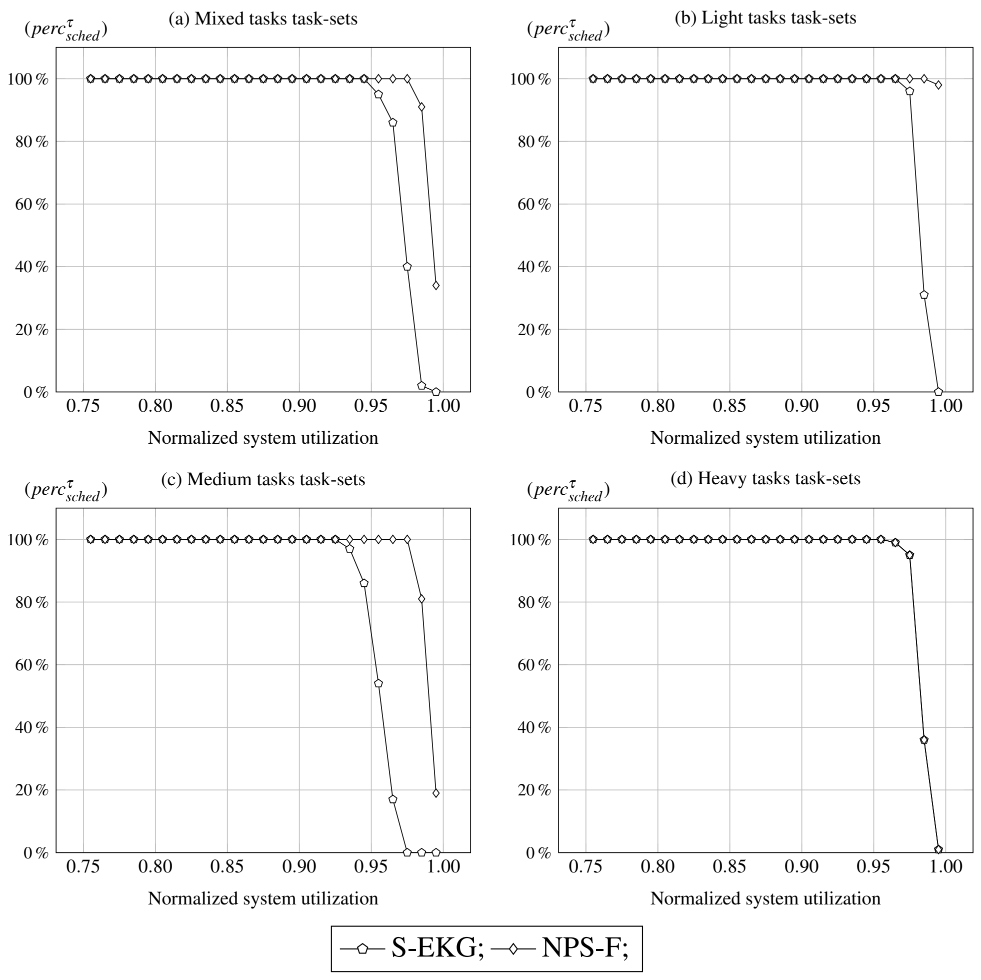


Figure 7.6: Comparison between S-EKG and NPS-F scheduling algorithms according to the new demand-based schedulability analysis using δ equal to one.

Figures 7.2, 7.3, 7.4, and 7.5). This is due to the FF bin-packing heuristic used by NPS-F that produces fuller (in terms of processing capacity) servers and consequently less inflation is required. Consequently, the number of servers is also smaller, because each server potentially has more tasks assigned to it. Recall that the inflation is used to compensate the time when tasks are in the system and are not allowed to be executed because of the exclusive nature of the server reserves. Hence, if a reserve occupies almost the entire time slot the required inflation is less, because those tasks (assigned to that reserve) are prevented from executing during a small amount of time in each time slot. Conversely, the smaller the reserve, the higher the required inflation (as a fraction of that reserve).

Figure 7.6 presents a comparison between S-EKG and NPS-F based on the schedulability success ratio. For the sake of presentation, we opt to plot the results considering δ equal to one. From the results presented in Figure 7.6, we can conclude that NPS-F is able to schedule more task sets than S-EKG.

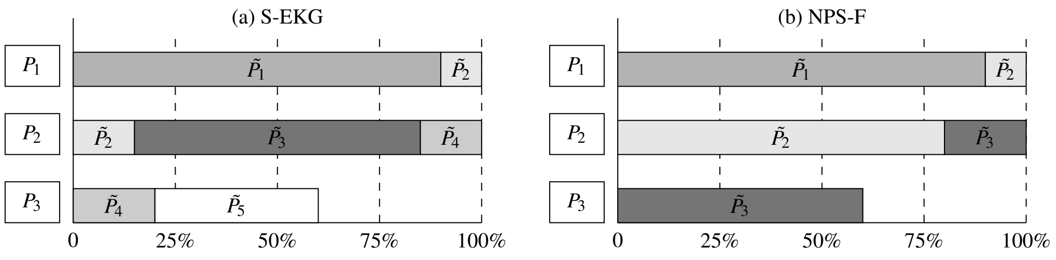


Figure 7.7: Task-to-processor assignment patterns of the S-EKG and NPS-F.

Figure 7.7 illustrates the task-to-processor assignment pattern of S-EKG, inset (a), and of NPS-F, inset (b). As mentioned before, this difference stems from the bin-packing heuristics used by both. S-EKG employs an adapted NF bin-packing heuristic that ensures only one task per-split-server, whereas NPS-F employs the FF bin-packing heuristic that produces fuller servers. As a consequence of such heuristics, generally, the processor time slot composition for NPS-F is, most often, divided into two reserves, whereas for S-EKG is divided into three reserves.

7.2.5 Evaluation of the new analysis in the presence of overheads

In our previous experiments, the new analysis and task assignment were shown to be efficient both under S-EKG and NPS-F. Therefore, in the presence of overheads, we only provide results for scheduling under NPS-F, because under the new schedulability analysis the S-EKG can be viewed as a special case of NPS-F, where the split-servers have assigned only one task.

Contrary to what holds in the absence of overheads, our results (see Figure 7.8) generally show that the schedulability improves as the value of parameter δ decreases. This is in accordance with some of the findings in [BBA11], which states that δ equal to one produces the best schedule. In other words, the most-preemption light setting is, contrary to intuition associated to the parameter δ, also the best for schedulability, once all overheads are factored in. This is because of the overheads associated with the implementation of the reserves themselves.

However, the results plotted in inset (b) of Figure 7.8 are different from the other results plotted in the same figure. There are two reasons for those results. On one hand, such task sets are composed by light tasks, whose u_i is uniformly distributed in the range [0.05, 0.35), and, as a consequence of the task-to-server assigning procedure, the capacity of the servers is almost full. On the other hand, the server-to-processor assignment procedure classifies servers as single, if the inflation of such servers exceeds the threshold defined by rule A2. Note that, single servers execute on dedicated processors. In other words, a single server executes on one processor that has assigned only one server, and in this case, there is no need to create reserves.

From the results presented and discussed in this subsection, it is clear that this kind of scheduling algorithms tend to require, in the presence of overheads, less inflation (i.e. less processing capacity lost) with longer S, which is achievable with δ equal to one. Application of rules A1 and A2 could provide a way for using a longer S without any processing capacity lost. S is computed as the $\frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(D_i, T_i)$ (see Equation 5.41); that is, considering all tasks of the task set. Probably,

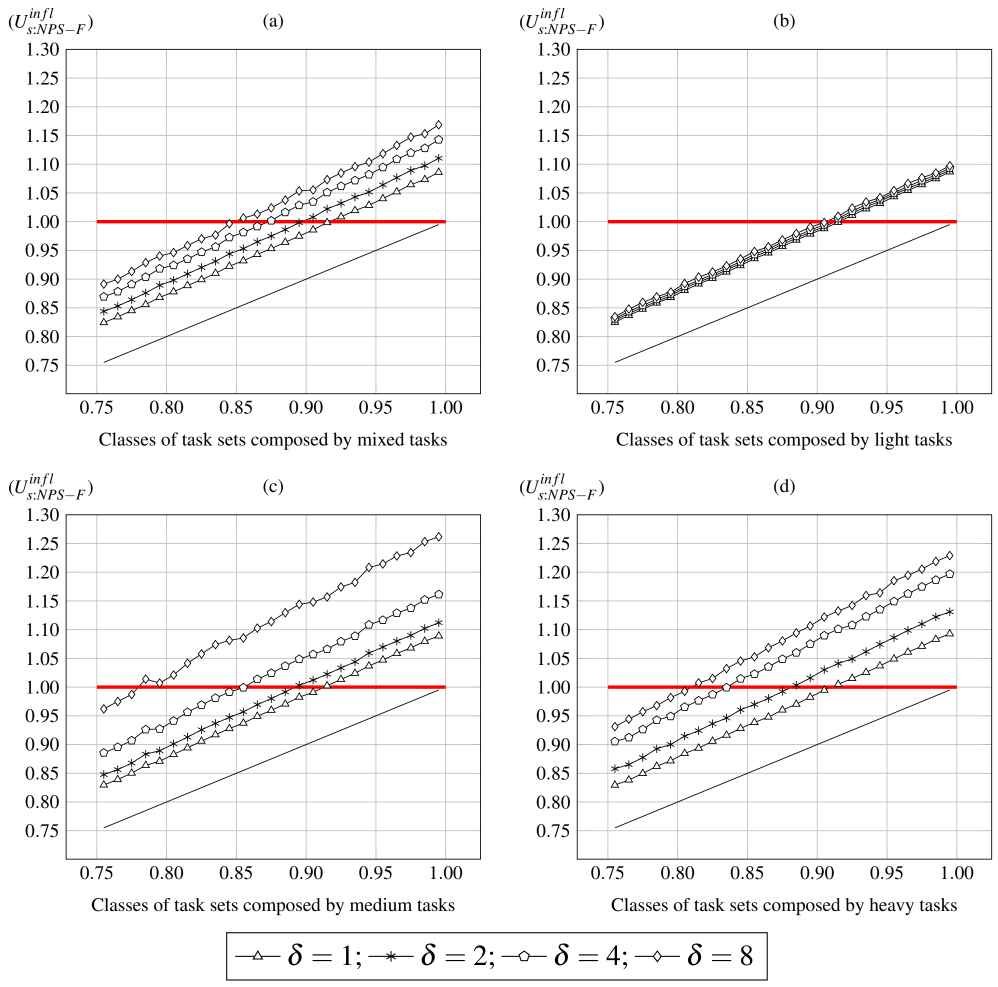


Figure 7.8: Evaluation of the effect of the overheads on the new schedulability analysis.

we could get longer S if the set of tasks, $\tau[\tilde{P}_q]$, mapped to a single server, according to rule A2, were excluded from such computation. Another strategy, could be, whenever rule A1 is applied to compute a new S taking into account the set of tasks mapped to the remaining servers. Furthermore, with this strategy we could get a system with multiple time slot lengths. This is because rule A1 creates an *assignment break* in the system; that is, there is no split server shared between two subsequent processors when rule A1 is applied. Figure 7.9 illustrates this assignment break; there is no shared server between processors P_2 and P_3, so, the time slot lengths could be different in processors $\{P_3, P_4\}$ *vs.* processors $\{P_1, P_2\}$. Recall that, this is achieved without any processing capacity penalty.

7.2.6 Reliability of the schedulability analysis

The lower efficiency of the utilization-based analysis could conceivably (but naively!) be expected to provide a safety margin to compensate for overheads that occur in real systems, instead of

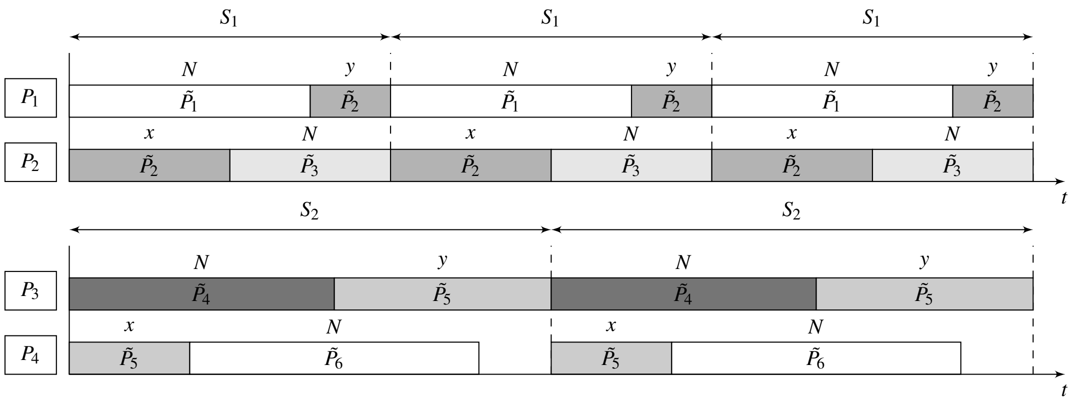


Figure 7.9: Multiple time slot lengths in a system.

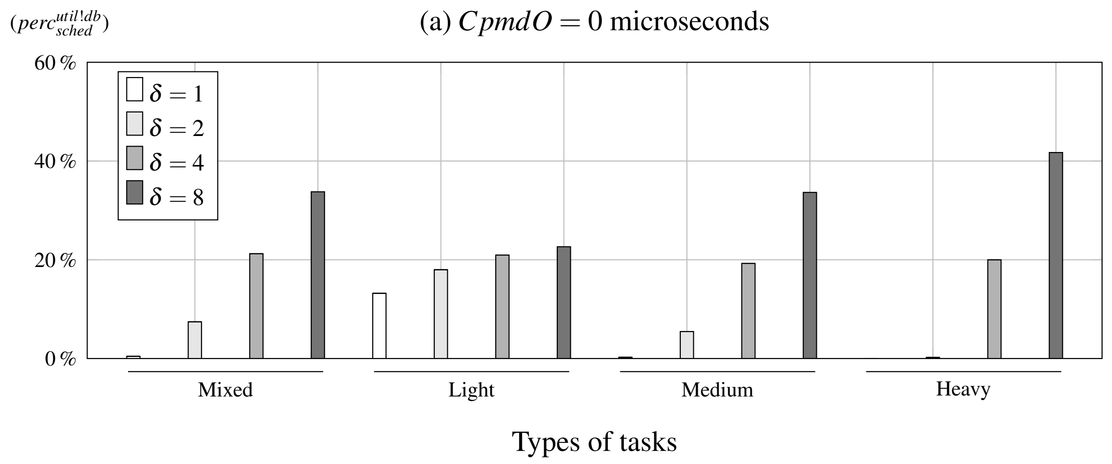
explicitly accounting for them in the analysis. However, there is no guarantee that this over-provisioning is sufficient. It may well be the case that the utilization-based test considers a task set as schedulable, when it really is not because of the overheads incurred in real systems.

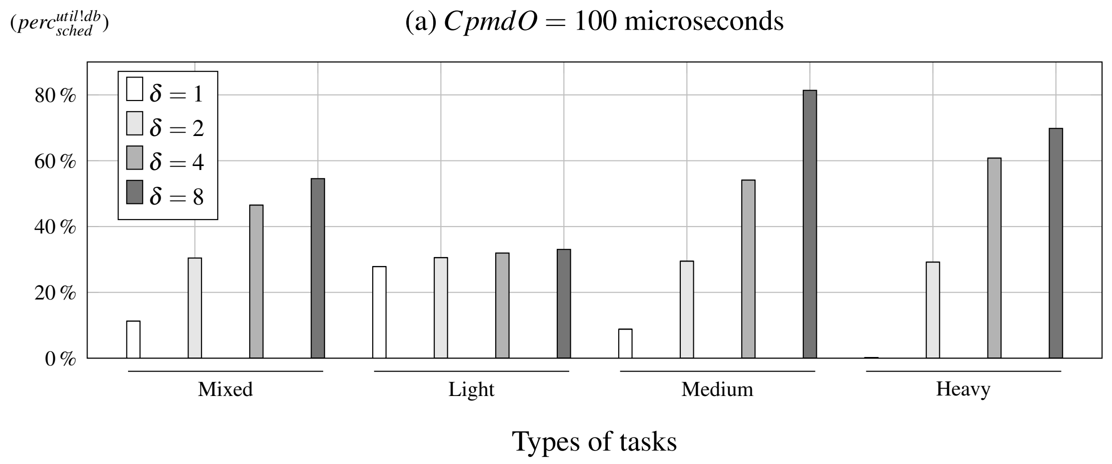
To better evaluate this possibility, we carried out a study in which we assessed whether the task sets deemed schedulable using the utilization-based analysis (which ignored overheads) were unschedulable according to the new demand-based overhead-aware schedulability analysis. Therefore the metric we used in this study was:

$$perc_{sched}^{util!db} = \frac{nr_{sched}^{util} - nr_{sched}^{oa\_db|util}}{nr_{sched}^{util}} \cdot 100 \tag{7.3}$$

where nr_{sched}^{util} is the number of task sets deemed schedulable according to the utilization-based schedulability analysis and $nr_{sched}^{oa\_db|util}$ is the number of these task sets that are also schedulable according to the demand-based overhead-aware schedulability analysis.

In all experiments of this study, we kept all overheads constant using the values presented in Section 7.2.1. For the *CpmdO*, given the dependence of this parameter on the load, we chose to perform a sensitivity analysis and experimented with two more values for this parameter: zero and 500 microseconds. The zero value represents a best case for the utilization-based analysis. The 500 microseconds value corresponds to a rather high value for the CPMD, taking into account that the minimum task period, and consequently the time slot duration, in any task set is not much higher than five milliseconds. For some light tasks, 500 microseconds may be larger than the task computation time (or execution time) itself. The value of 100 microseconds, by comparison should not penalize lighter tasks as much. As in the previous study, for each task set generated according to Algorithm 7.1, we used all the values of the design parameter δ considered (one, two, four, and eight). Furthermore, we ignored all the interrupts except the tick interrupts, which occur on a per-processor basis. (This is tantamount to assuming that interrupt handling is performed by a dedicated processor.) Like in the previous subsection, each point in the plots represents an average of the normalized utilization for 100 randomly generated task sets, satisfying the corresponding parameter values.





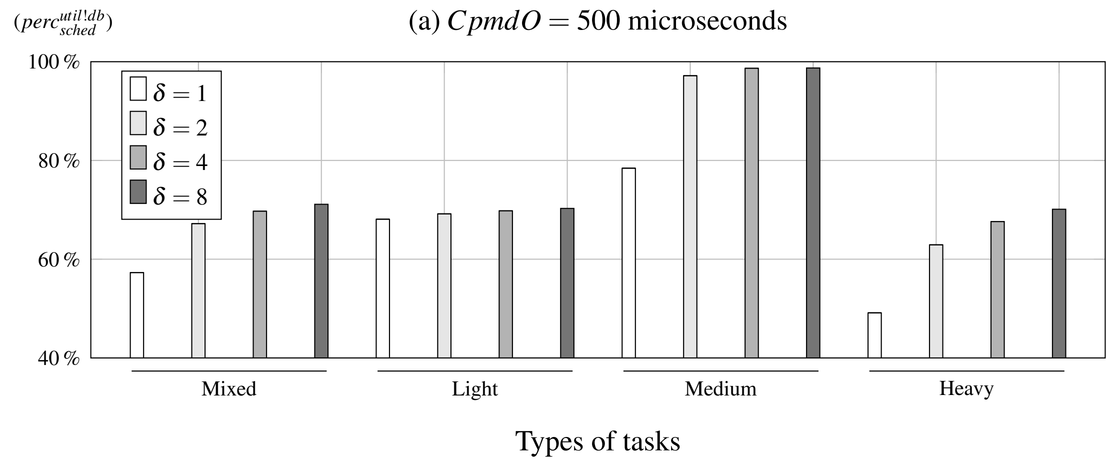


Figure 7.10: Fraction of task sets schedulable according to the utilization-based analysis, but not schedulable according to the new demand-based overhead-aware schedulability analysis, as a function of the type of tasks of the task set and of the design parameter δ.

Figure 7.10 summarizes the results of this study. Each inset shows the results for a different value of the *CpmdO* parameter. The value of this parameter has a major effect, although it may not be that apparent at first sight, because the ranges in the y-axis are different. As expected, the higher the value of *CpmdO*, the higher the fraction of tasks deemed schedulable according to the utilization-based analysis, but not schedulable by the new demand-based overhead-aware schedulability analysis, for the parameter values considered. Also clear is the effect of the design parameter δ. The higher the value of this parameter, the higher the fraction of tasks that are not schedulable.

Figure 7.11 shows the fraction of non schedulable task sets ignoring the utilization of the task set, to make the dependence on the factors considered more clear. Inset (a) of Figure 7.11 shows the dependence on the utilization of the task sets, U_s, for the mixed task sets with *CpmdO* equal to zero. As shown, for utilizations lower than a certain value, which depends on the value of δ, all task sets are schedulable according to both analyses. However, at a given point the fraction of non-schedulable task sets rises sharply to one, and remains around one until a point, which also depends on δ, when it then drops more or less sharply to zero. As shown, the value of δ determines the width of the plateau where the fraction is equal to one: the higher the value of δ the earlier the fraction rises to one, and the later it drops back to zero. For these parameter settings, for δ equal to one, the fraction of unschedulable task sets never reaches one, rather increases up to around 0.60 and then drops back to zero. In any case, the pattern is clear and applies also to other types of task sets and different values of *CpmdO*, and can be easily explained with the help of inset (a) of Figure 7.4 and inset (b) of Figure 7.11, which show the average inflated utilization respectively for mixed tasks task sets for the utilization-based analysis and for the demand-based overhead-aware schedulability analysis with *CpmdO* equal to zero.

Consider a given value of δ, say four. For task sets whose utilization is below 0.91, the overheads are small enough that virtually all task sets are considered schedulable by both analyses. As the task set utilization, U_s, increases from 0.91 to 0.95, the average inflated utilization according to the new analysis increases and becomes higher than one, see inset (a) of Figure 7.11, so that virtually all task sets are deemed unschedulable. On the other hand in that range, for δ equal to four, the inflated utilization according to the utilization-based analysis, see inset (a) of Figure 7.4, is still below about 0.97, and many task sets are still deemed schedulable. Therefore, in that interval (from 0.91 to 0.95) the fraction of non-schedulable task sets rises from zero to one, and remains one until it drops sharply for task sets whose utilization is in the range [0.99, 1.0), which are all deemed unschedulable also by the utilization-based analysis.

Even though the new demand-based overhead-aware schedulability analysis is conservative, i.e. is based on worst-case assumptions, and therefore it may be that a task set it considers non-schedulable is actually schedulable, the parameter values we used are all values we measured in a real system, except the values for the CPMD overheads. For the latter, we assumed several values including zero, and even in this case, which is rather optimistic, the utilization-based analysis may consider a given task set schedulable, when some tasks may miss their deadline. This is unacceptable in safety-critical hard-real time systems, where the consequences of missing a deadline may

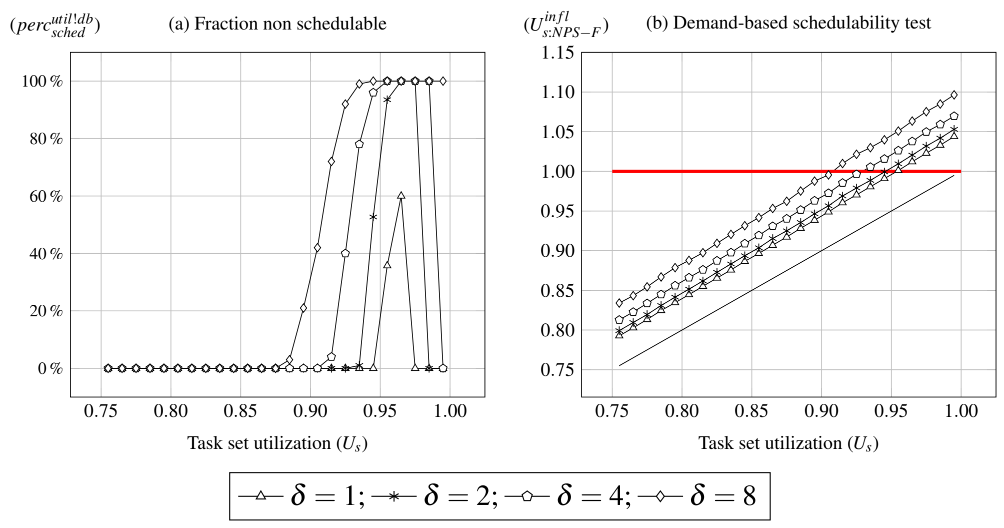


Figure 7.11: Fraction of mixed task sets that are considered schedulable by the original schedulability analysis but are not by the schedulability with overheads ($CpmdO$ equal to zero)

be catastrophic. The demand-based overhead-aware schedulability analysis we developed allows to account for all overheads incurred by an implementation, does so in a conservative way, and therefore ensures that its results are reliable, as long as the values of its parameters are valid.

7.2.7 Slot-based *vs.* reserve-based

In this subsection, we compare the schedulability of NPS-F against Carousel-EDF. We assume the schedulability analysis as respectively formulated in Sections 5.3 and 6.4. However, for a question of fairness, in the case of NPS-F, we eliminate the release interference among servers; that is, $\text{dbf}_{\text{RelI}}^{\text{sb:non-split}}(\tilde{P}_q,t)$ and $\text{dbf}_{\text{RelI}}^{\text{sb:split}}(\tilde{P}_q,t)$ given by Equations 5.21 and 5.33, respectively, are both equal to zero. Further, we assume that $IpiL$ (used in Equation 5.30) is also equal to zero. In doing so, we assume that the release mechanism described in Subsection 6.3.2 can be also employed for the slot-based task-splitting scheduling algorithms.

Figure 7.12 shows the average inflated utilization for each algorithm and the value of the normalized utilization of the task set. The average normalized inflated utilization with Carousel-EDF is smaller than that of NPS-F, independently of the task set type considered; that is, the Carousel-EDF is able to schedule more task sets. This was expected, because NPS-F incurs more overheads than Carousel-EDF. However, the pessimism of the analysis may be larger for NPS-F than for Carousel-EDF. Indeed, based on experimental evidence [BBA10b], the analysis assumes that the worst-case cache-related overheads caused by preemptions are the same whether or not there is a migration. On average, however, the costs incurred by preemptions with migrations are likely to be higher than without, and preemptions with migrations are more likely in Carousel-EDF than in NPS-F.

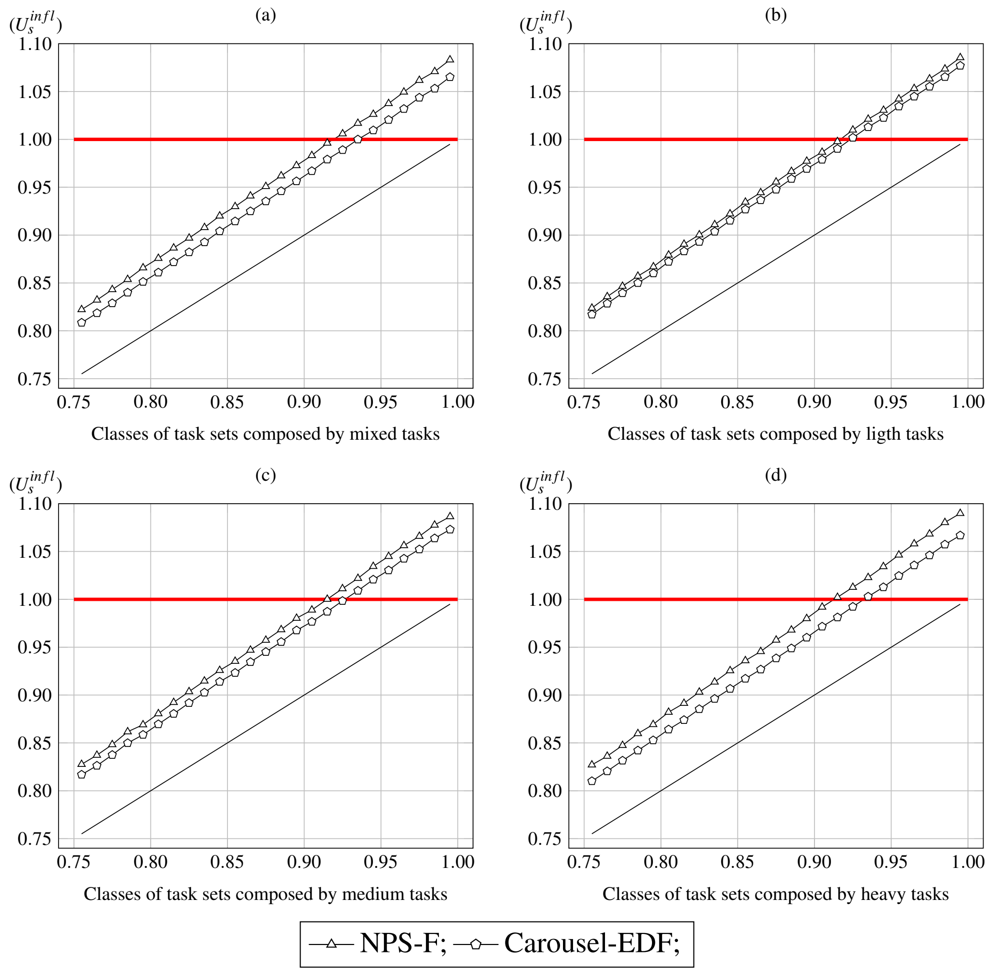


Figure 7.12: Comparison between the NPS-F and Carousel-EDF based on the normalized inflation.

7.3 The run-time task-dispatching algorithm

In this section, we evaluate the run-time task-dispatching algorithm. As mentioned before, the off-line procedure checks the schedulability of a task set; that is, if it is schedulable or not. It also configures the run-time task-dispatching algorithm; In other words, the off-line procedure computes the settings of the task-dispatching algorithm, such as, the set of tasks mapped to each server, the length of each reserve and so on.

We evaluate the efficiency of the new schedulability analysis for slot-based task-splitting scheduling algorithms. For that purpose, we present a comparative study of the NPS-F run-time task-dispatching algorithm. In that study, we compare the run-time task-dispatching algorithm efficiency when the task-dispatching algorithm settings are computed according to the original utilization-based and the new demand-based overhead-aware schedulability analyses. Next, we present an evaluation of NPS-F and Carousel-EDF scheduling algorithms, and those supported by the RT scheduling class of the PREEMPT-RT-patched Linux Kernel.

All of these evaluations are based on the percentage of deadline met, which is computed as:

$$perc^{jobs}_{deadline:met} = \frac{nr^{jobs}_{deadline:met}}{nr^{jobs}_{deadline:missed} + nr^{jobs}_{deadline:met}} \cdot 100 \tag{7.4}$$

where $nr^{jobs}_{deadline:met}$ is the number of jobs of all tasks that met their deadlines. $nr^{jobs}_{deadline:missed}$ is the number of jobs of all tasks that missed their deadlines. $nr^{jobs}_{deadline:met}$ plus $nr^{jobs}_{deadline:missed}$ gives us the total number of all jobs of all tasks in a task set. We consider that one algorithm is better than the other if it meets more deadlines than the other.

Note that, in all experiments presented in this section the run-time task-dispatching (or scheduler) implementation of the slot-based algorithms is as described in Section 4.6 and the implementation of the reserve-based algorithms is as described in Section 6.3.

7.3.1 Experimental setup

The task sets used to evaluate the run-time task-dispatching algorithms were generated using the same method as described in Subsection 7.2.2 and the host platform (a 24-core machine) is the same one described in Subsection 7.2.1. We generated 50 task sets with utilization varying from 50% to 100% with an increment of 1%. We created task sets according to four types of tasks: mixed, light, medium, and heavy. The settings of each type of tasks are as defined in Subsection 7.2.2. Then, we created 200 task sets. The periods of all tasks of all task sets are uniformly distributed in the range [5, 50] milliseconds, with a resolution of one millisecond. In Appendix B, we deal with other ranges of periods. We ran each experiment during 100 seconds.

In order to simulate the execution time of each task, we coded a task using C language, whose snippet code is presented in Listing 7.1. In order to ensure that all tasks are ready for execution at the same time, we put all task to sleep until `time0`. The `clock_nanosleep` function allows the calling task to sleep for an interval specified with nanosecond precision. Then, the algorithm of that task consists of a loop of `nr_jobs` iterations. In each iteration, which we consider a job, it executes a set of empty for-loops to simulate (through the invocation of function `do_work`) the execution time, which is defined by `exec_time` variable, of the task. Then, it computes the release time of the next job, checks if the current job missed its deadline or not (by using the `clock_gettime` function to get the current time) and then it sleeps until the next release. Note that, this task is coded using functionalities provided by the Linux kernel; that is, it does not depend on any special feature added to the Linux kernel.

7.3.2 Why overheads have to be accounted for

In this subsection, we show why the overheads must be considered into the schedulability analysis. To this end, we present the results of several experiments that measure percentage of deadlines met, according to Equation 7.4. Figure 7.13 plots the results of a set of task sets whose run-time task-dispatching parameters were computed using the NPS-F original utilization-based schedulability analysis and the new demand-based overhead-aware schedulability analysis presented in

```c
struct timespec next_release;
struct timespec now;
unsigned long deadline_misses = 0;
unsigned long long release_time;

//task settings
unsigned long long exec_time = atoll(argv[...]);
unsigned long long period = atoll(argv[...]);
unsigned long long time0 = atoll(argv[...])
unsigned long nr_jobs = atol(argv[...]);;
...
release_time = time0;

//Converting nanoseconds to timespec
next_release.tv_sec  = release_time / NSEC_PER_SEC;
next_release.tv_nsec = release_time % NSEC_PER_SEC;

//Sleeps until time zero
clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &next_release,NULL);

for(j=0;j<nr_jobs;j++){

  //Executes its work
  do_work(exec_time);

  //Computes the next release
  release_time = release_time + period;

  //Converting nanoseconds to timespec
  next_release.tv_sec  = release_time / NSEC_PER_SEC;
  next_release.tv_nsec = release_time % NSEC_PER_SEC;

  //Gets the current time
  clock_gettime(CLOCK_MONOTONIC, &now);

  //Checks if it missed the deadline
  //that is, if the current time is higher than the next release
  if( (now.tv_sec * NSEC_PER_SEC + now.tv_nsec) > release_time)
    deadline_misses++;

  //Sleeps until next release
  clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &next_release,NULL);
}
...
```

Listing 7.1: The C language code of the task for experiments.

Section 5.3 (please note that y-axes do not have the same scale among all insets of Figure 7.13). For the new analysis, we consider the overhead values presented in Subsection 7.2.1 and for both analyses the parameter δ was set equal to one.

We set δ equal to one, because according to study pursued in Section 7.2 that value is best when practical issues are considered, like for instance, run-time overheads. In such plots, we only present and compare the results for the task set that are considered schedulable according to the respective schedulability analysis. These results allow us to highlight the importance of incorporating the run-time task-dispatching overheads into the schedulability theory. As can be seen from that figure, all task sets in which run-time task-dispatching input parameters were computed using the new demand-based overhead-aware schedulability analysis met all deadlines, as predicted by the schedulability analysis. By contrast, many of the task sets in which input parameters were computed using the original utilization-based missed deadlines, something not predicted by the respective schedulability analysis. Actually, that schedulability analysis considers those task sets as schedulable.

We would like to highlight that, the only difference between the experiments is the way that task-dispatching settings were computed, everything else is the same; that is, the code of each task, the scheduler implementation, the operating system, and the host platform are the same.

Note that, in Subsection 7.2.6, it is shown that some task sets considered schedulable according to the original utilization-based schedulability analysis are not schedulable according to the new demand-based overhead-aware schedulability analysis. Here, we show that, on one hand, for some task sets considered schedulable according to the utilization-based schedulability analysis are in fact non-schedulable from the results collected from experiments running on a real system. On the other hand, all task sets considered schedulable by the new demand-based overhead-aware schedulability are, in fact, schedulable. From this, we can conclude that the real-world overheads have to be considered into the schedulability to increase its reliability.

7.3.3 Enhancing real-time capabilities of the Linux kernel

The mainline Linux kernel is not designed for real-time systems, because none of its scheduling policies consider one fundamental real-time parameter: the deadline. We believe that the efforts of the PREEMPT-RT community on adding real-time capabilities have one drawback: the absence of any real-time scheduling algorithm. In this subsection, we present a performance comparison between the scheduling algorithms supported by the PREEMPT-RT-patched Linux kernel (RR and FIFO scheduling algorithms) and those added in the context of this work, NPS-F and Carousel-EDF.

As mentioned before, NPS-F and Carousel-EDF scheduling algorithms require the execution of an off-line application, which implements the new schedulability analysis developed in the context of this work. Given that, the release of tasks is performed by using the native Linux `clock_nanosleep` function, which does not follow the task release mechanism defined for Carousel-EDF (described in Subsection 6.3.2), we then use, for Carousel-EDF, the schedulability analysis with the changes introduced in Subsection 6.4.3.

Therefore, for each task set, that application computes the settings of the run-time task-dispatching algorithm, such as servers and respective reserves as well as their tasks and the time slot length. Then, using a specific API the run-time task-dispatching algorithm is configured. Finally, all tasks belonging to the task set are created, configured with their parameters, and then, launched.

The RR and FIFO algorithms have a more simplified setup phase. There is no need to configure the run-time task-dispatching algorithms. Therefore, the only thing that is required is to create all tasks of each task set, set their parameters and launch them.

We ran each task set scheduled according to RR, FIFO, NPS-F, and Carousel-EDF scheduling algorithms during 100 seconds in a 24-core machine. Figure 7.14 plots the percentage of deadlines met for each scheduling policy (please note that y-axes do not have the same scale among all insets

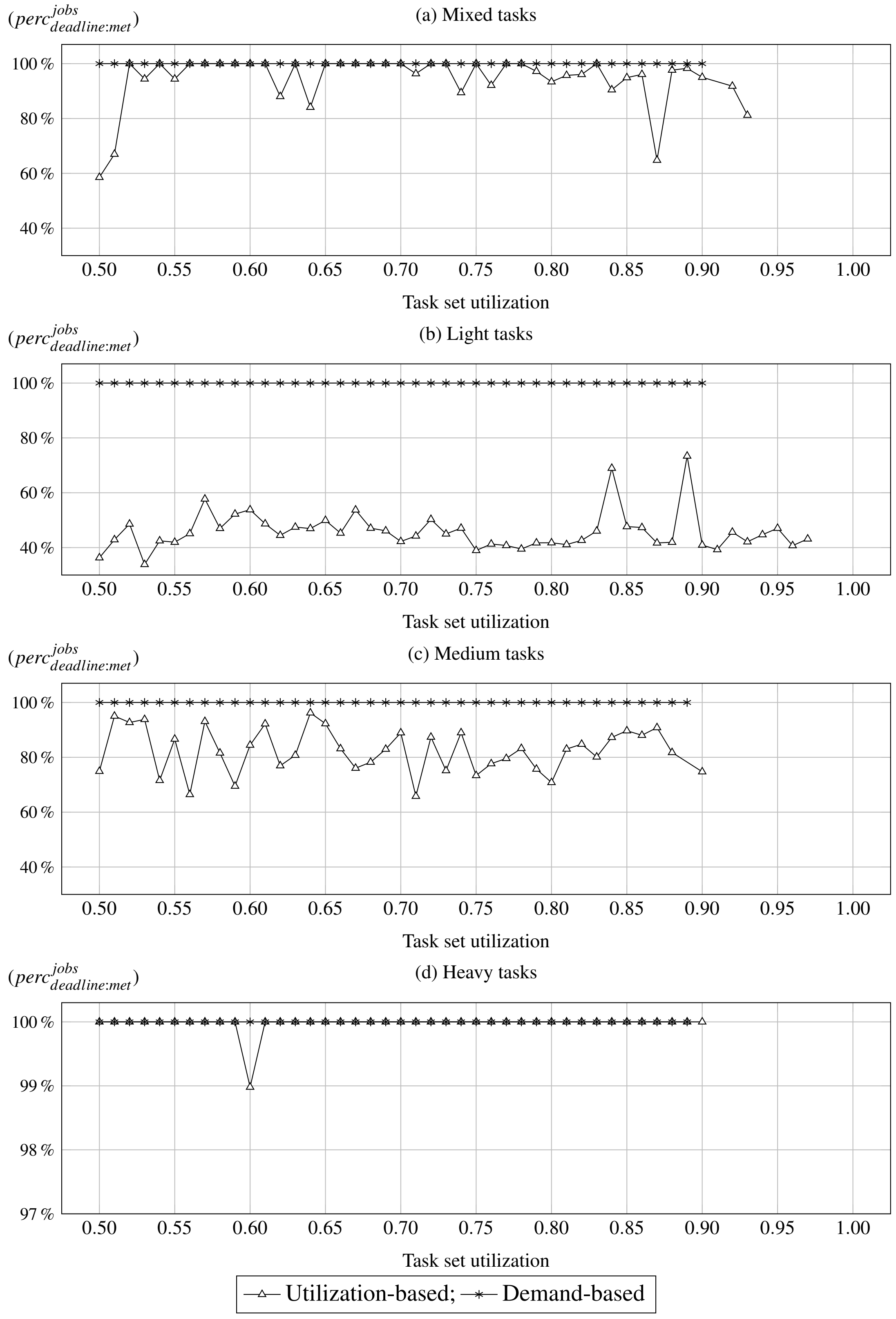


Figure 7.13: A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met.

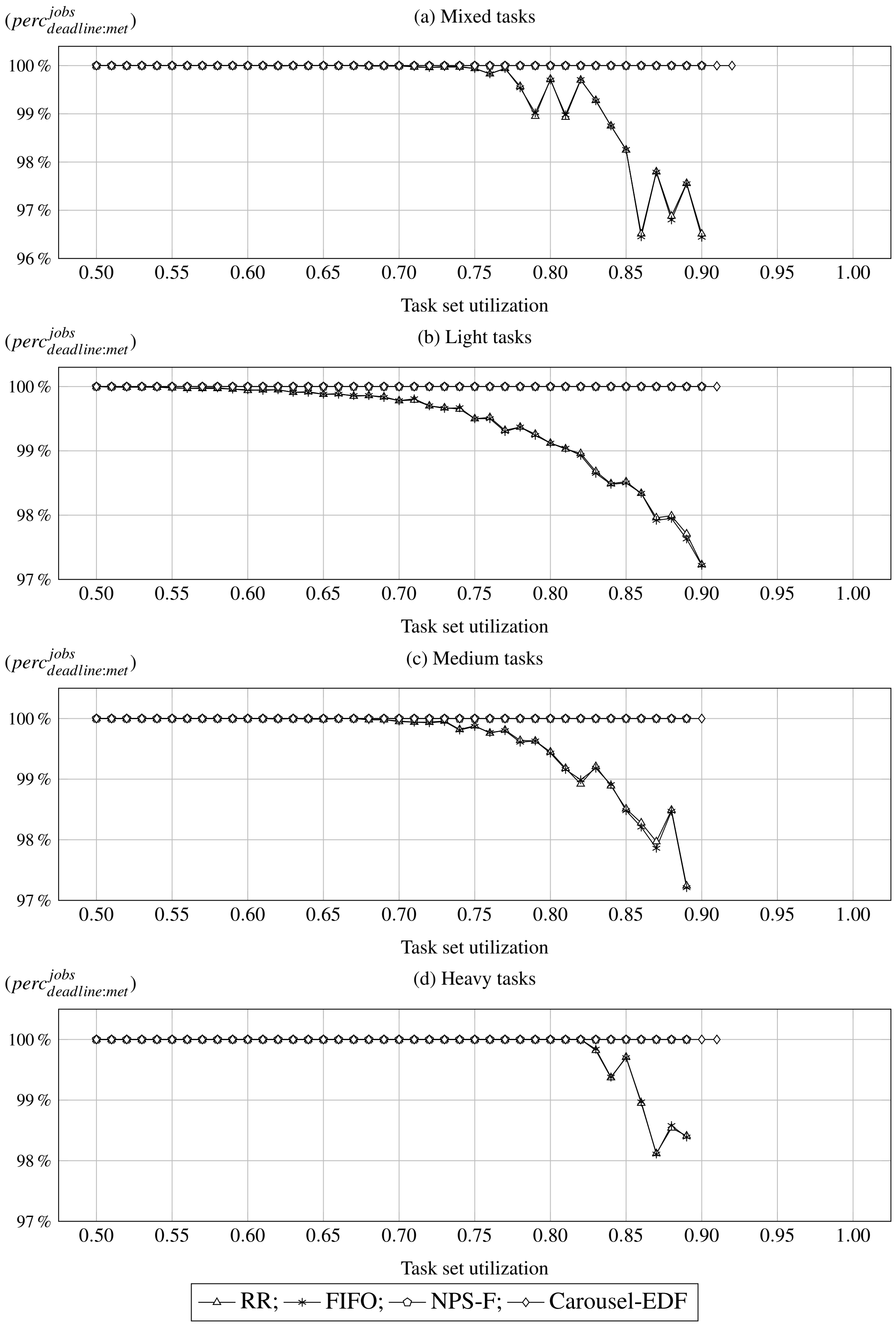


Figure 7.14: A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met.

of Figure 7.14). For the sake of presentation, we only present the RR and FIFO results for task sets that are considered schedulable according to NPS-F. Otherwise, the plots would be misleading, because the percentage of deadlines met decreases abruptly for task sets with U_s higher than 89%.

As can be seen by inspecting Figure 7.14, NPS-F and Carousel-EDF met all deadlines, as predicted by the theory. Note that, only schedulable task sets were considered. The difference between NPS-F and Carousel-EDF is the number of task sets considered schedulable, which is higher for Carousel-EDF than for NPS-F as predicted (and demonstrated in Subsection 7.2.7) by the schedulability analyses. Both RR and FIFO fail to meet all deadlines. Even though this is not visible in the plots, it should be noted that, all task sets with a U_s higher than 55% failed deadlines when scheduled under RR and FIFO.

It should be noted, that the slot-based task-splitting and also the reserve-based scheduling implementations, apart from specific code for managing some specific data structures, require minor modifications to the RT scheduling class. Therefore, in our opinion, adding support for appropriate scheduling algorithms to the Linux kernel (compatible with the PREEMPT-RT patch) is a practical and effective way to make this widely adopted operating system more suitable for real-time systems.

7.4 Summary

In this chapter, we present an evaluation of the scheduling algorithms under consideration in this work. We evaluated the efficiency of the off-line task-to-processor algorithms and also of the run-time task-dispatching algorithm. We have compared the utilization-based and the new demand-based overhead-aware schedulability analyses. We used as comparison metric the inflated utilization factor, in which we consider that one algorithm is better than the other if the former requires less inflated utilization than the latter.

From the study conducted in this chapter, we can conclude that the new demand-based overhead-aware schedulability analysis is better than the original utilization-based schedulability analysis. Using the new demand-based overhead-aware schedulability analysis, S-EKG is able to schedule task sets with utilizations much higher than its theoretical utilization bound, $\mathrm{UB}_{\mathrm{S-EKG}}$. For the NPS-F, the new demand-based overhead-aware schedulability analysis is also more efficient than the original utilization-based schedulability analysis. We also showed that NPS-F is better than S-EKG, because it requires less inflation. This is due to the bin-packing employed, which tends to produce fuller servers for NPS-F, compared to S-EKG. However, S-EKG presents one advantage that is, the number of split servers (and consequently also the number of split tasks) is known at design time, because S-EKG ensures that each split server is assigned only one task.

We also have concluded that, contrary to what holds for the utilization-based schedulability analysis (where the schedulability improves with higher δ), the new demand-based overhead-aware schedulability analysis presents better schedulability as the parameter δ decreases. That is, better schedulability is achieved with δ equal to one, because this creates longer time slots, which in turn implies fewer run-time overheads related to the management of the time division into reserves. With the purpose of getting longer time slots, we defined two strategies that employ rules A1 and A2 to use longer time slots without any penalty.

We measured the number of task sets that are considered schedulable according to the original utilization-based schedulability analysis and not according the new demand-based overhead-aware schedulability analysis. Note that, the former does not take into account the overheads, whereas the latter incorporates the run-time overheads. We observed that in some cases almost 100% of the task sets considered schedulable by the utilization-based schedulability analysis are in fact unschedulable according to the new demand-based overhead-aware schedulability analysis, especially those with higher δ parameter values. Furthermore, we presented a comparative study in a real system running on a 24-core machine where the run-time task-dispatching algorithm settings were computed by the utilization-based schedulability analysis and the new demand-based overhead-aware schedulability analysis. We observed many deadline misses for task sets configured according to the utilization-based schedulability analysis whereas no deadline miss was observed for task sets configured according to the new demand-based overhead-aware schedulability analysis.

We also showed that the Linux kernel does not provide any suitable scheduling algorithm for multiprocessor real-time systems. We presented a comparative study in a real system running on a 24-core, where we compare the scheduling algorithms supported by the PREEMPT-RT-patched Linux kernel, RR and FIFO, against the NPS-F and the Carousel-EDF scheduling algorithms added to the Linux kernel in the context of this work. We observe that those scheduling algorithms supported by the PREEMPT-RT-patched Linux kernel miss deadlines (all task sets with U_s higher than 55% miss deadlines), whereas no deadline misses occurred for task sets scheduled according to NPS-F and the Carousel-EDF scheduling algorithms.

Finally, we observed that the Carousel-EDF scheduling algorithm performs a little better than the NPS-F scheduling algorithm, in the sense that it is able to schedule more task sets (with higher U_s).

Chapter 8

Conclusions and future work

8.1 Introduction

The advent of multi-core chips has renewed the interest of the research community on real-time scheduling upon multiprocessor systems. Real-time scheduling theory for single-cores is considered a mature research field, but real-time scheduling theory for multiprocessors is an emerging research field.

Multiprocessor systems may provide great computing capacity if appropriate scheduling algorithms are used. Unfortunately, many of the research work on scheduling algorithms for multiprocessor systems are based on a set of assumptions that have no correspondence with a real-system; that is, only few works have addressed the scheduling algorithms implementation and the operating system related issues.

Focused on slot-based task-splitting scheduling algorithms, we bridge the gap between the theory and practice for those scheduling algorithms. For that purpose, we implemented those scheduling algorithms on a PREEMPT-RT-patched Linux kernel version. We also developed new scheduling theory for that type of scheduling algorithms that incorporates practical issues to increase the reliability and efficiency.

In this chapter, we review and discuss the results and contributions presented in this dissertation. We also provide some directions for future work.

8.2 Summary of results

As mentioned above, in the course of this work, we narrow the gap between theory and practice for slot-based task-splitting scheduling algorithms for multiprocessor systems. To achieve such goal, we create a unified model that accommodate most of all scheduling algorithms of that class (Section 4.3). We identified one practical issue that could decrease the performance of the scheduling algorithm, which we denoted as the instantaneous migration problem (Section 4.5). The identification of such practical issues guided us towards one implementation rule, which was followed throughout the implementation of the scheduling algorithms under consideration in this work, namely that: *the interactions among processors must be reduced.* Following this rule, we implemented slot-based task-splitting scheduling algorithms in which the scheduling decisions are taken on a per-processor basis; that is, our implementations isolate the scheduling decisions on one processor from activities and decisions on other processors. Further, given the potentially higher number of task migrations imposed by the slot-based task-splitting scheduling algorithms and the associated high overheads for migrating tasks, our implementations employ a lightweight task migration procedure that avoids the locking of multiple ready-queues (Section 4.6). For that purpose, we made available a ready-queue per-server, in which all its ready tasks are stored. An atomic variable is used for stating the queue as *busy* if some ready task of that server is being executed on any processor or *free* otherwise.

The implementation of the scheduler follows a two-level hierarchical approach. Recall, that the processor scheduler executes on per-processor basis; that is, the scheduler executes on the processor in which it is invoked (through the invocation of the `schedule` function) and can be executed simultaneously in multiple processors. For each processor, the top-level scheduler is responsible for the management of the time (through the high resolution timers). It divides the time into slots and then each slot is divided into reserves. Whenever a reserve starts, it sets the corresponding server for that reserve (by updating a pointer) and triggers the invocation of the low-level scheduler. The low-level scheduler picks the highest-priority task, if any, from the assigned server, if the server is stated as free, otherwise it forces its invocation some time later (empirically set to five microseconds). This procedure repeats until the end of the reserve if the server has ready tasks and it is stated as busy, otherwise this mechanism is not triggered.

Being aware of the efforts of the PREEMPT-RT kernel developer community to add real-time capabilities to the Linux kernel and its wide acceptance, we made our implementation completely compatible with the PREEMPT-RT patch. It was implemented with minor modifications to the PREEMPT-RT-patched Linux kernel. Using our implementation, we identified and modelled all the scheduling overheads incurred by slot-based task-splitting scheduling algorithms when running in a real operating system (Section 4.7).

In Chapter 5, we formulated a unified schedulability theory applicable to slot-based task-splitting scheduling algorithms. This new theory is based on exact schedulability tests, thus also overcoming many sources of pessimism in the previous related work. Furthermore, as a response to the fact that many unrealistic assumptions, present in the original theory, tend to undermine the theoretical potential of such scheduling schemes, we identified and modelled into the new analysis all the overheads incurred by the algorithms in consideration. The outcome is a new overhead-aware schedulability analysis that permits increased efficiency and reliability.

In Chapter 6, we presented a reserve-based scheduling algorithm called Carousel-EDF. Carousel-EDF is an offshoot of the slot-based task-splitting scheduling algorithms described in Chapter 4. It also relies on the concept of servers that are instantiated by the means of time reserves. However, contrarily to slot-based task-splitting scheduling algorithms, which migrate tasks at time slot and reserve boundaries, Carousel-EDF only migrates tasks at reserve boundaries. Consequently, the time is no longer constrained to time slots; it is divided into time reserves instead.

Carousel-EDF preserves the utilization bounds of slot-based task-splitting scheduling algorithms, namely that of NPS-F, which is the highest among algorithms not based on a single dispatching queue and that have few preemptions. Furthermore, with respect to slot-based task-splitting scheduling algorithms, Carousel-EDF reduces up to 50% the worst-case number of context switches and the worst-case number of preemptions caused by the time division mechanism. We also formulated an overhead-aware schedulability analysis based on demand-bound functions grounded on an implementation on a PREEMPT-RT-patched Linux kernel version.

In Chapter 7 (complemented with Appendix B), we showed that the new demand-based overhead-aware schedulability analysis for slot-based task-splitting scheduling algorithms is much more efficient and reliable than the original utilization-based one. Assuming that the cost of the overheads is negligible, the new demand-based overhead-aware schedulability analysis is much more efficient; that is, it is able to schedule more task sets with higher utilizations.

When the estimates for the various overheads are factored into the new demand-based overhead-aware schedulability analysis, it appears to perform worse (in terms of the fraction of tasks sets declared schedulable) compared to the original utilization-based (but still, overhead-oblivious) analysis. However, when those task sets are executed in a real systems (on a 24-core machine) it was verified that, in fact, many tasks deemed schedulable by the overhead-oblivious utilization-based analysis missed deadlines, whereas all task sets considered schedulable by the new demand-based overhead-aware schedulability analysis were, in fact, schedulable, and did not miss any deadlines. This shows that incorporating overheads into schedulability analysis of a scheduling algorithm is fundamental from the reliability perspective. As it is intuitive, this imposes the need to accurately measure the overheads in order to provide a correct input to the schedulabilty analysis.

From the research conducted in Chapter 7, slot-based task-splitting scheduling algorithms benefit from longer time slots, in the presence of reserve-related overheads. In that sense, we defined two strategies for using longer time slots without any penalty. As it is intuitive, the benefits of longer time slots come from the decrease in overheads.

We compared a slot-based task-splitting scheduling algorithm, the NPS-F, against reserve-based scheduling algorithm, the Carousel-EDF. In average, Carousel-EDF performs better than NPS-F because it is able to schedule task sets with higher utilization and when executed on a real system no task misses its deadlines.

In our opinion, adding appropriate scheduling algorithms to the Linux kernel (compatible with PREEMPT-RT patch) is an important step towards making this widely adopted operating system more suitable for real-time systems. We presented a comparative study in a real system running on a 24-core machine, where we compare the scheduling algorithms supported by the PREEMPT-RT-patched Linux kernel (RR and FIFO) against the NPS-F and the Carousel-EDF scheduling algorithms added to the Linux kernel in the context of this work. We observe that those scheduling algorithms supported by the PREEMPT-RT-patched Linux kernel miss deadlines (all task sets with a utilization higher than 55% miss deadlines), whereas no deadline miss was observed for task sets scheduled according to NPS-F and Carousel-EDF, which were deemed schedulable by the new analysis.

8.3 Future work

There are several open topics for further research that have been identified: (i) support of different hardware platforms; (ii) addressing multi-criticality systems; (iii) use of different/variations of the scheduling algorithms; (iv) considering resource sharing; (v) implementation slot- and reserve-based scheduling algorithms in co-kernel approaches; (vi) improve the run-time overheads measurements; and (vii) practical performance comparison with other real-time scheduling algorithms. Each of these topics will be briefly developed in the following paragraphs.

Support of different hardware platforms. Supporting other hardware platforms would require the adaptation of the schedulability analysis developed in the context of this work as well as the implementation techniques. Possibly, other platform-related details could require significant changes on both. Our hardware platform is based on x86-64 architecture. It could be interesting to extend the work presented in this dissertation to other hardware platforms. In terms of memory model, our target platform is a UMA platform; that is, it has a shared memory model, which implies a uniform memory access. It could be interesting to investigate distributed memory models, where the memory access times are not uniform (that is, NUMA platforms). NUMA platforms impose a set of challenges, especially for migrating tasks. Note that such platforms scale better (in terms of number of processors) than UMA platforms. Other platform issue to be investigated would be the consideration of platforms equipped with non-identical processors. This could reveal itself as a very challenging work.

Addressing multi-criticality systems. In our work, no distinction of tasks according to the criticality was made: task sets composed by different types (in terms of criticality level) of tasks were not considered. It could be interesting to address task sets composed by hard and soft real-time tasks as well as other types as best-effort tasks [BA07], for instance. The scheduling algorithms under consideration in this work could be a valid option for those task sets. In fact, some works have recently addressed mixed-criticality tasks [Ves07]. We believe that the scheduling algorithms under consideration in this work provide nice features suitable for mixed-criticality, namely, the isolation provided by the time reserves. Additionally, S-EKG could be an interesting approach for systems with very strict requirements such as hard real-time, because most of the tasks are assigned to one processor and a few to two processors. In the course of this work, we did not investigate this feature.

Use of different/variations of the scheduling algorithms. All scheduling algorithms studied in the context of this work are EDF-based. It could be interesting to investigate slot-based task-splitting and reserve-based scheduling algorithms using other type of scheduling algorithms, namely with static-priorities algorithms like RM or Deadline Monotonic.

Some recent work on real-time systems have addressed the possibility to defer scheduling decisions to as late as possible to increase the effectiveness of the systems [BB10, DBM$^+$13]. This approach tends to reduce the impact of the practical issues on the performance of the scheduling algorithms. Given the time division mechanism employed by the scheduling algorithms under consideration in this dissertation, we believe that those algorithms could benefit from the above mentioned approach. Let us assume the following scenario: a task is executing on the respective server reserve and other server's task with higher priority is released in a time instant very close to the end of the respective server reserve. According to the currently defined rules the low priority task must be preempted to give execution to the higher priority task. However, let us assume that the time to the end of the reserve is smaller that the cost of switching tasks. In such case, it is better to keep the low priority task executing until the end of the reserve and when the subsequent server reserve starts the higher priority task will be start being executed.

We did not investigate the cluster variants of the slot-based task-splitting, in which each cluster is formed out of processors on the same chip. The purpose of those variants is to eliminate off chip task migrations, which could be costly due to Cache-related Preemption and Migration Delay (CPMD). As mentioned in Section 7.2, the cost of the CPMD can degrade the performance of slot-based task-splitting and also of reserve-based scheduling algorithms. It could be interesting to evaluate the performance of both types of scheduling algorithms taking into account the CPMD.

In the slot-based task-splitting and in the reserve-based scheduling algorithms the time slot length is computed according to Equation 5.41 ($S = \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$); that is, it depends on the tasks parameters and also on the δ parameter. It could be interesting to investigate the impact on those scheduling algorithms of an arbitrarily chosen time slot length.

Considering resource sharing. Supporting resource sharing is fundamental for the wide acceptance of our scheduling algorithms [BSNN07, BA08, BA10]. Usually, when several tasks share a resource, only one task is allowed to use that resource in turn. This requires the use of some locking protocol to assure mutual exclusion. Moreover, real-time systems require predictability. Therefore, the real-time locking protocols must assure the correct use of the shared resource as well as the predictable real-time behaviour. Some work has been done to address this issue on multiprocessor systems. However, at our best knowledge, there is no published work specific for the scheduling algorithms we proposed in this work. Therefore, adding support for resource sharing to slot- and reserve-based scheduling algorithms is a very relevant and challenging future research effort.

Implementation slot- and reserve-based scheduling algorithms in co-kernel approaches. Our target platform for implementations runs Linux. It could be interesting to extend the work presented in this dissertation to other operating systems, namely those based on co-kernel approaches like Xenomai [Xen12] or RTAI [RTA12], for instance. Note that, the co-kernel approaches are able to offer more predictability than the Linux kernel. Due to that, they are employed in many real-time systems.

Improve the run-time overheads measurements. Measuring the run-time overheads requires intrusive operations into the operating system. Developing non-intrusive or lightweight-intrusive procedures to measure the run-time overheads could be an interesting add-on to the work presented in this dissertation. Note that, probably, it is not possible to get overhead costs via a purely analytical approach. As mentioned before, computational systems are complex and composed by many components of hardware and software, including the operating system, that are rarely documented with sufficient detail allowing the derivation of the required overhead costs.

Practical performance comparison with other real-time scheduling algorithms. In the context of real-time systems, the comparison of scheduling algorithms is, typically, based on the utilization factor and other theoretically-derived parameters. Practical evaluations are much more difficult to perform than theoretical-only evaluations, since the former are subject to many practical issues imposed by the host platform (formed by the operating system and the hardware). Worst, some of them cannot be controlled. Even when it is possible to install into the same host platform the system that we want to evaluate, other issues arise. For instance, it is not possible to execute tasks, which code is presented in Listing 7.1, in the LITMUS^{RT} [LR12] or RESCH [Kat12] frameworks, because specific APIs for implementing the tasks are needed to be used. The same happens in systems with co-kernel approaches. Therefore, some changes are required to the system (tasks or frameworks) in order to be comparable. In any case, benchmarking against other real-time Linux kernel implementations is also a very relevant and challenging future research effort.

References

[AAJ$^+$02] Tarek Abdelzaher, Björn Andersson, Jan Jonsson, Vivek Sharma, and Minh Nguyen. The aperiodic multiprocessor utilization bound for liquid tasks. In *proc. of the 8th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 173–184, San Jose, CA, USA, 2002.

[AAJ03] Björn Andersson, Tarek Abdelzaher, and Jan Jonsson. Global priority-driven aperiodic scheduling on multiprocessors. In *proc. of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 8.1, Nice, France, 2003.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *proc. of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 4–13, Madrid, Spain, 1998.

[AB08] Björn Andersson and Konstantinos Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *proc. of the 20th IEEE Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 243–252, Prague, Czech Republic, 2008.

[ABB08] Björn Andersson, Konstantinos Bletsas, and Sanjoy Baruah. Scheduling arbitrary-deadline sporadic tasks on multiprocessors. In *proc. of the 29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 385–394, Barcelona, Spain, 2008.

[ABD05] James Anderson, Vasile Bud, and UmaMaheswari Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *proc. of the 17th IEEE Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 199–208, Palma de Mallorca, Balearic Islands, Spain, 2005.

[ABJ01] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 193–202, London, UK, 2001.

[ABR$^+$93] Neil Audsley, Alan Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[Ade13] Adeos. Adaptive Domain Environment for Operating Systems, Apr. 2013.

[ADM12] Sebastian Altmeyer, Robert Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time System.*, 48(5):499–526, Sep 2012.

[AJ03] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and Pfair static-priority scheduling on multiprocessors are 50%. In *proc. of the 15th IEEE Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 33–41, Porto, Portugal, 2003.

[AS00] James Anderson and Anand Srinivasan. Early-release fair scheduling. In *proc. of the 12th IEEE Euromicro Conference on Real-Time Systems (ECRTS'00)*, pages 35–43, Stockholm, Sweden, 2000.

[AS04] James Anderson and Anand Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal Computer and System Sciences*, 68(1):157–204, 2004.

[AT06] Björn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemption. In *proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'06)*, pages 322–334, Sydney, Australia, 2006.

[BA07] Björn Brandenburg and James Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *proc. of the 19th IEEE Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 61–70, Pisa, Italy, 2007.

[BA08] Björn Brandenburg and James Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In *proc. of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, pages 185–194, Kaohisung, Taiwan, 2008.

[BA09a] Konstantinos Bletsas and Björn Andersson. Notional processors: an approach for multiprocessor scheduling. In *proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, pages 3–12, San Francisco, CA, USA, 2009.

[BA09b] Konstantinos Bletsas and Björn Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *proc. of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 385–394, Washington, DC, USA, 2009.

[BA09c] Björn Brandenburg and James Anderson. On the implementation of global real-time schedulers. In *proc. of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 214–224, Washington, DC., USA, 2009.

[BA10] Björn Brandenburg and James Anderson. Optimality results for multiprocessor real-time locking. In *proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 49–60, San Diego, CA, USA, 2010.

[BA11] Konstantinos Bletsas and Björn Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *Real-Time Systems*, 47(4):319–355, 2011.

[Bak03] Theodore Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *proc. of the 24th Real-Time Systems Symposium (RTSS'03)*, pages 120–129, Cancun, Mexico, 2003.

[Bak10] Theodore Baker. What to make of multicore processors for reliable real-time systems? In *proc. of the 15th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe'10, pages 1–18, Valencia, Spain, 2010.

[Bas11] Andrea Bastoni. *Towards the Integration of Theory and Practice in Multiprocessor Real-Time Scheduling*. PhD thesis, University of Rome "Tor Vergata", 2011.

[BB07] Theodore Baker and Sanjoy Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[BB10] M. Bertogna and S. Baruah. Limited preemption EDF scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics*, 6(4):579–591, 2010.

[BBA10a] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *proc. of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*, pages 33–44, Brussels, Belgium, 2010.

[BBA10b] Andrea Bastoni, Björn Brandenburg, and James Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*, pages 14–24, San Diego, CA, USA, 2010.

[BBA11] Andrea Bastoni, Björn Brandenburg, and James Anderson. Is semi-partitioned scheduling practical? In *proc. of the 23rd IEEE Euromicro Conference on Real-Time Systems (ECRTS'11)*, pages 125–135, Porto, Portugal, 2011.

[BBC$^+$07] Björn Brandenburg, Aaron Block, John Calandrino, UmaMaheswari Devi, Hennadiy Leontyev, and James Anderson. LITMUS$^{\text{RT}}$: A status reports. In *proc. of the 9th Real-Time Linux Workshop (RTLWS'07)*, pages 107–123, Linz, Austria, 2007.

[BC05] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O Reilly & Associates Inc, 2005.

[BCA08] Björn Brandenburg, John Calandrino, and James Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *proc. of the 29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 157–169, Barcelona, Spain, 2008.

[BCPV93] Sanjoy Baruah, N. Cohen, Greg Plaxton, and Donald Varvel. Proportionate progress: a notion of fairness in resource allocation. In *in proc. of the 25th annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993.

[BCPV96] Sanjoy Baruah, N. Cohen, Greg Plaxton, and Donald Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[BDWZ12] Alan Burns, Robert Davis, P. Wang, and Fengxiang Zhang. Partitioned EDF scheduling for multiprocessors using a $C = D$ task splitting scheme. *Real-Time Syst.*, 48(1):3–33, Jan 2012.

[Ber08] Marko Bertogna. *Real-Time Scheduling Analysis for Multiprocessor Platforms*. PhD thesis, Scuola Superiore Sant'Anna, Pisa, 2008.

[BGP95] Sanjoy Baruah, Johannes Gehrke, and Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *proc. of the 9th IEEE International Symposium on Parallel Processing (IPPS'95)*, pages 280–288, Santa Barbara, CA, USA, 1995.

[BLM$^+$08] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application. *IEEE Transactions on Nuclear Science*, 55(1):435–439, 2008.

[BM10] Jeremy H. Brown and Brad Martin. How fast is fast enough? choosing between Xenomai and Linux for real-time applications. In *proc. of the 12th Real-Time Linux Workshop (RTLWS'12)*, pages 1–17, Nairobi, Kenya, 2010.

[BMR90] Sanjoy Baruah, Aloysius Mok, and Louis Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *proc. of the 11st IEEE Real-Time Systems Symposium (RTSS'90)*, pages 182–190, Lake Buena Vista, Florida, USA, 1990.

[Bra11] Björn Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[BSNN07] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. Sirap: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *proc. of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT '07)*, pages 279–288, Salzburg, Austria, 2007.

[But04] G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[CGJ97] Edward Coffman, Michael Garey, and David Johnson. Approximation algorithms for bin packing: a survey. In Dorit S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1997.

[CLB$^+$06] John Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari Devi, and James Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *proc. of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 111–126, Rio de Janeiro, Brazil, 2006.

[CRJ06] Hyeonjoong Cho, B. Ravindran, and E.D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *proc. of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 101–110, Rio de Janeiro, Brazil, 2006.

[DB11a] R.I. Davis and A. Burns. Fpzl schedulability analysis. In *proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 245–256, Chicago, IL, USA, 2011.

[DB11b] Robert Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[DB11c] Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40, January 2011.

[DBM$^+$13] R.I. Davis, A. Burns, J. Marinho, V. Nelis, S.M. Petters, and M. Bertogna. Global fixed priority scheduling with deferred pre-emption. In *proc. of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages –, Taipei, Taiwan, 2013.

[Der74] M. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[Dev] Advanced Micro Devices. AMD Opteron Processor. http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=645.

[Dha77] S. Dhall. *Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1977.

[DL78] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[DSA$^+$13] Rob Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *proc. of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS'13)*, pages 168–179, Paris, France, 2013.

[EL98] Rudolf Eigenmann and David J. Lilja. Von neumann computers, 1998.

[FGB10] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45:26–71, 2010.

[FKY08] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *proc. of the 20th IEEE Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 13–22, Prague, Czech Republic, 2008.

[Fou13] The Linux Foundation. The Linux foundation, Apr. 2013.

[FTCS09] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino. An EDF scheduling class for the Linux kernel. In *proc. of the 11th Real-Time Linux Workshop (RTLWS'09)*, pages 197–204, Dresden, Germany, 2009.

[GCB13] A. Gujarati, F. Cerqueira, and B.B. Brandenburg. Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities. In *proc. of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS'13)*, pages 69–79, Paris, France, 2013.

[GK06] Pawel Gepner and Michal Kowalik. Multi-core processors: New way to achieve high system performance. In *proc of 5th international symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, Bialystok, Poland, 2006.

[GN06] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the Linux time subsystems. In *proc. of the Linux Symposium (OLS'06)*, pages 333–346, Ottawa, Ontario, Canada, 2006.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and nonpreemptive real-time uniprocessor scheduling. Technical Report 2966, INRIA, France, 1996.

[GSYY10] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with liu & layland's utilization bound. In *proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, pages 165–174, Stockholm, Sweden, 2010.

[HBJK06] H. Hoang, G. Buttazzo, M. Jonsson, and S. Karlsson. Computing the minimum EDF feasible deadline in periodic systems. In *proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, pages 125–134, Sydney, Australia, 2006.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Int13] Intel, Santa Clara, CA, USA. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, March 2013.

[Jen06] J. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica*, 30:175–193, 1906.

[KAS93] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *Transactions on Software Engineering*, 19(9):920–934, 1993.

[Kat12] Shinpei Kato. RESCH: A loadable real-time scheduler suite for Linux, Oct. 2012.

[KRI09] Shinpei Kato, R. Rajkumar, and Y. Ishikawa. A loadable real-time scheduler suite for multicore platforms. Technical report, Technical Report CMU-ECE-TR09-12, 2009.

[KY07] Shinpei Kato and Nobuyuki Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 441–450, Daegu, Korea, 2007.

[KY08a] Shinpei Kato and Nobuyuki Yamasaki. Modular real-time Linux. In *proc. of the 10th Real-Time Linux Workshop (RTLWS'10)*, pages 97–102, Guadalajara, Mexico, 2008.

[KY08b] Shinpei Kato and Nobuyuki Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *proc. of the 8th ACM/IEEE International Conference on Embedded Software (EMSOFT'08)*, pages 139–148, Atlanta, GA, USA, 2008.

[KY09] Shinpei Kato and Nobuyuki Yamasaki. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *proc. of the 21st IEEE Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 239–248, Dublin, Ireland, 2009.

[KY11] Shinpei Kato and Nobuyuki Yamasaki. Global EDF-based scheduling with laxity-driven priority promotion. *J. Syst. Archit.*, 57(5):498–517, May 2011.

[LAMD13] W. Lunniss, S. Altmeyer, C. Maiza, and R.I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *proc. of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, pages –, Philadelphia, PA, USA, 2013.

[Lee94] S. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology (TENCON'94)*, pages 607–611, 1994.

[Leh90] John Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *proc. of the 11st IEEE Real-Time Systems Symposium (RTSS'90)*, pages 201–213, Lake Buena Vista, FL , USA, 1990.

[LFS$^+$10] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. DP-Fair: A simple model for understanding optimal multiprocessor scheduling. In *proc. of the 22nd IEEE Euromicro Conference on Real-Time Systems (ECRTS'10)*, pages 3–13, Brussels, Belgium, 2010.

[LGDG00] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *proc. of the 12th IEEE Euromicro Conference on Real-Time Systems (ECRTS'00)*, pages 25–33, Stockholm, Sweden, 2000.

[Liu69] C. L. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. *JPL Space Programs Summary*, II(1):37–60, 1969.

[LL73] 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.

[LL03] Lars Lundberg and H. Lennerstad. Global multiprocessor scheduling of aperiodic tasks using time-independent priorities. *proc. of the 9th Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, 0:170–180, 2003.

[LLFC11] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *proc. of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11)*, pages 6–15, Porto, Portugal, 2011.

[Low06] G. Lowney. Why intel is designing multi-core processors. In *proc. of the 25th internation Symposium on Parallelism in Algorithms and Architectures (SPAA'06) – Invited Talk*, pages 113–113, Montreal, Canada, 2006.

[LR12] LITMUS-RT. Linux Testbed for MUltiprocessor Scheduling in real-time systems (LITMUS$^{\mathrm{RT}}$), Oct 2012.

[LRL09] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 239–248, Dublin, Ireland, 2009.

[Lun02] Lars Lundberg. Analyzing fixed-priority global multiprocessor scheduling. In *proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 145–153, San Jose, CA, USA, 2002.

[Mau08] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[MBH$^+$02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[McK05] Paul McKenney. A realtime preemption overview, August 2005.

[Mol04a] Ingo Molnar. CONFIG_PREEMPT_REALTIME, fully preemptible kernel, October 2004.

[Mol04b] Ingo Molnar. remove the BKL (big kernel lock), this time for real, September 2004.

[Pai00] Vijay S. Pai. *Exploiting Instruction-Level Parallelism for Memory System Performance*. PhD thesis, Rice University, 2000.

[PR12] PREEMPT-RT. Real-time Linux wiki, Sep 2012.

[RCM96] I. Ripoll, A. Crespo, and A.K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[ReT12] ReTAS. Real-time TAsk-Splitting scheduling algorithms framework., Oct 2012.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[RH07] Steven Rostedt and Darren V. Hart. Internals of the RT patch. In *proc. of the Linux Symposium (OLS'07)*, pages 161–172, Ottawa, Ontario, Canada, 2007.

[RLM$^+$11] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *proc. of the 32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 104–115, Vienna, Austria, 2011.

[RTA12] RTAI. Real Time Application Interface for Linux, Sep 2012.

[SA02] Anand Srinivasan and James Anderson. Optimal rate-based scheduling on multiprocessors. In *proc. of the 34th ACM Symposium on Theory of Computing*, pages 189–198, Montreal, Quebec, Canada, 2002.

[SAT10a] Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Challenges and design principles for implementing slot-based task-splitting multiprocessor scheduling. In *proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10) – Work-in-Progress Session*, San Diego, CA, USA, 2010.

[SAT10b] Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Implementing slot-based task-splitting multiprocessor scheduling. Technical report HURRAY-TR-100504, CISTER, Polytechnic Institute of Porto (ISEP-IPP), 2010.

[SAT11] Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Implementing slot-based task-splitting multiprocessor scheduling. In *proc. of the 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11)*, pages 256–265, Vasteras, Sweden, 2011.

[SB02] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.

[SBAT11] Paulo Baltarejo Sousa, Konstantinos Bletsas, Björn Andersson, and Eduardo Tovar. Practical aspects of slot-based task-splitting dispatching in its schedulability analysis. In *proc. of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, pages 224–230, Toyama, Japan, 2011.

[SBT$^+$13] Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Akesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. Technical report CISTER-TR-130201, CISTER, Polytechnic Institute of Porto (ISEP-IPP), 2013.

[SBTA11] Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, and Björn Andersson. On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems. In *proc. of the 13th Real-Time Linux Workshop (RTLWS'13)*, pages 207–218, Prague, Czech Republic, 2011.

[SCH12] SCHED_DEADLINE. An implementation of the popular earliest deadline first (EDF) scheduling algorithm or the linux kernel., Oct. 2012.

[SLB13] José Augusto Santos-Jr., George Lima, and Konstantinos Bletsas. On the processor utilisation bound of the C=D scheduling algorithm. In *proc of Real-time systems: the past, the present, and the future (Alanfest 2013)*`http://www.cs.unc.edu/~baruah/AlanFest/Procs.pdf`, pages 119–132, 2013.

[SMD$^+$10] Angela C. Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. Parallelism via multithreaded and multicore CPUs. *Computer*, 43:24–32, 2010.

[SPT12] Paulo Baltarejo Sousa, Nuno Pereira, and Eduardo Tovar. Enhancing the real-time capabilities of the Linux kernel. *SIGBED Rev.*, 9(4):45–48, November 2012.

[Spu96] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical report, INRIA, 1996.

[SRL90] Lui Sha, R. Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SSTB13] Paulo Baltarejo Sousa, Pedro Souto, Eduardo Tovar, and Konstantinos Bletsas. The Carousel-EDF scheduling algorithm for multiprocessor systems. In *proc. of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages –, Taipei, Taiwan, 2013.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[TIL12] TILERA. Manycore without boundaries. delivering the world's first 100-core general purpose processor., Sep 2012.

[TW08] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Pearson, Upper Saddle River, NJ, 3 edition, 2008.

[Ves07] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *proc. of the 28th IEEE Real-Time Systems Symposium (RTSS'07)*, pages 239–243, Tucson, Arizona, USA, 2007.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per

Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[Xen12] Xenomai. Real-time framework for Linux, Sep 2012.

[Yod99] Victor Yodaiken. The RTLinux manifesto. In *proc. of the 5th Linux Expo*, 1999.

[ZB08] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. Technical report YCS-2008-426, University of York, Department of Computer Science, 2008.

[ZB09] Fengxiang Zhang and Alan Burns. Improvement to quick processor-demand analysis for EDF-scheduled real-time systems. In *proc. of the 21st IEEE Euromicro Conference on Real-Time Systems (ECRTS' 09)*, pages 76–86, Dublin, Ireland, 2009.

[ZMM03] Dakai Zhu, Daniel Mossé, and Rami G. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *proc. of the 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, pages 142–151, Cancun, Mexico, 2003.

Appendix A

Interrupts

Interrupts in an operating system are raised by any hardware or software component when it wants the processor's attention. Basically, when a processor receives an interrupt, it stops the execution of the current task to execute the interrupt service routine (ISR) associated with the received interrupt. We model each sporadic interrupt Int_i as a sporadic interfering task with minimum inter-arrival time of T_i^{Int} and an execution time equal to C_i^{Int} which runs at a higher priority than normal tasks. Some periodic interrupts (for example, the periodic tick) are also characterised by an arrival jitter J_i^{Int}. We assume that C_i^{Int} is much smaller than S ($C_i^{Int} \ll S$) and the number of distinct types of interrupts is limited to n^{Int}. Modelling interrupts in this manner allows safely upper-bounding the cumulative execution demand by interrupts using conventional analysis for sporadic tasks (which we next formulate in detail). However, specifically for interrupts with $T_i^{Int} < S$, modelling such interrupts instead as bursty periodic tasks[1] is sometimes less pessimistic. Intuitively, this is because under slot-based task-splitting scheduling algorithms, interrupts only exert overhead when present inside the reserve(s) of the server under consideration. Outside its reserve(s), an interrupt contributes to the processor-demand of some other server instead. Therefore, for each interrupt with $T_i^{Int} < S$, we consider both models and pick the least pessimistic value.

Next, we present in detail how the processor demand of interrupts is bounded under our analysis. Note that depending on the server type (split or non-split), we model the execution demand of interrupts in a slightly different way. First, let us consider non-split servers:

A non-split server executes in a single reserve of length $Res^{len}[\tilde{P}_q]$ (see Equation 5.22). The cumulative execution demand by all interrupts on the server can be upper-bounded as

$$\mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{sb:non-split}}(\tilde{P}_q, t) = \sum_{i=1}^{n^{Int}} \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t) \tag{A.1}$$

where $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ is the respective upper bound on the processor demand by interrupt Int_i.

[1]The bursty periodic arrival model was introduced in [ABR$^+$93]

For every interrupt Int_i (irrespective of whether $T_i^{Int} < S$ or $T_i^{Int} \geq S$), a upper bound for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ can be (pessimistically) computed as:

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, t) = \left\lceil \frac{t + J_i^{int}}{T_i^{Int}} \right\rceil \cdot C_i^{Int} \tag{A.2}$$

The pessimism in this derivation lies in that even interrupts raised outside the reserves of the server in consideration are treated as interfering. However, for interrupts with $T_i^{Int} < S$, an alternative derivation of an upper bound for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ is possible, using the bursty periodic model, as explained earlier. Namely:

$$\begin{aligned} &\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(non-split:bursty)}}(Int_i, \tilde{P}_q, t) = \\ &\qquad \mathrm{nr}^{\mathrm{S}}(t) \cdot \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{N}}(Int_i, \tilde{P}_q) + \mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{tail:N}}(Int_i, \tilde{P}_q) \; \forall i, T_i^{Int} < S \end{aligned} \tag{A.3}$$

where $\mathrm{nr}^{\mathrm{S}}(t)$ is a upper bound on the number of time slots fully contained in the time interval under consideration (of length t) and $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{N}}(Int_i, \tilde{P}_q)$ is a upper bound on the demand by Int_i inside the reserve (of length $Res^{len}[\tilde{P}_q]$) of server $\tilde{P}_q$ in a single time slot (of length S). Similarly for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:N}}(Int_i, \tilde{P}_q)$, but over the remaining time interval (i.e. the "tail") of length t^{tail}. These terms, in turn, are derived as:

$$\mathrm{nr}^{\mathrm{S}}(t) = \left\lfloor \frac{t + ResJ}{S} \right\rfloor \tag{A.4}$$

$$t^{tail} = t - \mathrm{nr}^{\mathrm{S}}(t) \cdot S \tag{A.5}$$

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{N}}(Int_i, \tilde{P}_q) = \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, Res^{len}[\tilde{P}_q]) \tag{A.6}$$

and

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:N}}(Int_i, \tilde{P}_q) = \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, \min(t^{tail}, Res^{len}[\tilde{P}_q])) \tag{A.7}$$

Often, though not always, $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(non-split:bursty)}}(Int_i, \tilde{P}_q, t)$ provides a less pessimistic estimate than $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, t)$, for interrupts with $T_i^{Int} < S$. Hence in the general case $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, t)$ is computed as:

$$\begin{aligned} &\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t) = \\ &\quad \begin{cases} \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, t) & \text{if } T_i^{Int} \geq S \\ \\ \min\left(\mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{(continuous)}}(Int_i, t),\ \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(non-split:bursty)}}(Int_i, \tilde{P}_q, t)\right) & \text{if } T_i^{Int} < S \end{cases} \end{aligned} \tag{A.8}$$

Next, we deal with split servers. For convenience we define:

$$
\begin{aligned}
Res_x^{len}[\tilde{P}_q] &= x[P_{p+1}] = U_x^{infl}[\tilde{P}_q] \cdot S \\
Res_y^{len}[\tilde{P}_q] &= y[P_p] = U_y^{infl}[\tilde{P}_q] \cdot S
\end{aligned}
\tag{A.9}
$$

As mentioned before, a split server $\tilde{P}_q$ executes on two reserves (of length $Res_x^{len}[\tilde{P}_q]$ and $Res_y^{len}[\tilde{P}_q]$) separated by Ω time units. That is, it is idle during Ω time units, next it executes during $x[P_{p+1}]$ on processor P_{p+1}, then it is idle again during Ω time units, and finally it executes during $y[P_p]$ on processor P_p (see Figure 4.6).

The cumulative execution demand by all interrupts on the server reserves can be upper-bounded as

$$
\mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{sb:split}}(\tilde{P}_q,t) = \sum_{i=1}^{n^{Int}} \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:split}}(Int_i,\tilde{P}_q,t) \tag{A.10}
$$

where $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:split}}(Int_i,\tilde{P}_q,t)$ is the respective upper bound on the processor demand by interrupt Int_i.

For every interrupt Int_i (irrespective of whether $T_i^{Int} < S$ or $T_i^{Int} \geq S$), a upper bound for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:split}}(Int_i,\tilde{P}_q,t)$ can be (pessimistically) computed by $\mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{continuous})}(Int_i,t)$. As for non-split servers, the pessimism in this derivation lies in that even interrupts raised outside the reserves of the server in consideration are treated as interfering. Then, for interrupts with $T_i^{Int} < S$ it is possible to employ the bursty periodic model that may reduce the pessimism:

$$
\begin{aligned}
&\mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{split:bursty})}(Int_i,\tilde{P}_q,t) = \\
&\qquad \mathrm{nr}^{\mathrm{S}}(t) \cdot (\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{X}}(Int_i,\tilde{P}_q) + \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{Y}}(Int_i,\tilde{P}_q)) \\
&\quad + \ \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:X}}(Int_i,\tilde{P}_q) + \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:Y}}(Int_i,\tilde{P}_q) \ \forall i, T_i^{Int} < S
\end{aligned}
\tag{A.11}
$$

where $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{X}}(Int_i,\tilde{P}_q)$ and $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{Y}}(Int_i,\tilde{P}_q)$ are upper bounds on the demand by Int_i inside the reserves (of length $Res_x^{len}[\tilde{P}_q]$ and $Res_y^{len}[\tilde{P}_q]$) of server $\tilde{P}_q$ in a single time slot (of length S). Similarly for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:X}}(Int_i,\tilde{P}_q)$ and $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:Y}}(Int_i,\tilde{P}_q)$, but over the remaining time interval (i.e. the "tail") of length t^{tail}. These terms, in turn, are derived as:

$$
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{X}}(Int_i,\tilde{P}_q) = \mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{continuous})}(Int_i,Res_x^{len}[\tilde{P}_q]) \tag{A.12}
$$

$$
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{Y}}(Int_i,\tilde{P}_q) = \mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{continuous})}(Int_i,Res_y^{len}[\tilde{P}_q]) \tag{A.13}
$$

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:X}}(Int_i,\tilde{P}_q) =
\begin{cases}
0 & \text{if } t^{tail} \leq \Omega \\
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, t^{tail} - \Omega) & \text{if } \Omega < t^{tail} \leq \Omega + Res_x^{len}[\tilde{P}_q] \\
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, Res_x^{len}[\tilde{P}_q]) & \text{if } \Omega + Res_x^{len}[\tilde{P}_q] < t^{tail} \leq S
\end{cases} \tag{A.14}$$

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{tail:Y}}(Int_i,\tilde{P}_q) =
\begin{cases}
0 & \text{if } t^{tail} \leq 2\cdot\Omega + Res_x^{len}[\tilde{P}_q] \\
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i, t^{tail} - 2\cdot\Omega + Res_x^{len}[\tilde{P}_q]) & \text{if } 2\cdot\Omega + Res_x^{len}[\tilde{P}_q] < t^{tail} \leq S
\end{cases} \tag{A.15}$$

As for non-split servers, often, though not always, $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(split:bursty)}}(Int_i,\tilde{P}_q,t)$ provides a less pessimistic estimate than $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i,t)$, for interrupts with $T_i^{Int} < S$. Hence in the general case $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:split}}(Int_i,t)$ is computed as:

$$\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:split}}(Int_i,\tilde{P}_q,t) =
\begin{cases}
\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i,t) & \text{if } T_i^{Int} \geq S \\
\min\left(\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(continuous)}}(Int_i,t),\ \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{(split:bursty)}}(Int_i,\tilde{P}_q,t)\right) & \text{if } T_i^{Int} < S
\end{cases} \tag{A.16}$$

Appendix B

Other results

This appendix complements the evaluation presented in Chapter 7. The results presented in Chapter 7 are all from task sets with *short-period*, in the range 5 to 50 milliseconds. Here, we also provide results considering for task sets with *long-period*, in the range 50 to 250 milliseconds, and with *mixed-period*, in the range 5 to 250 milliseconds. As the name implies, the utilization-based schedulability analysis relies on the ratio between C_i and T_i. The task set generation, whose pseudo-code is presented in Algorithm 7.1, first, randomly generates the utilization, next, randomly generates the period, and then, computes the execution time of each task τ_i as $C_i = u_i \cdot T_i$. The system utilization (U_s) is used as a criterion for task set generation, which means that for the same utilization-related input parameters the task set generated is the same. Therefore, the results for mixed- and long-period task sets are equals to the results for short period task sets presented in in Chapter 7. In this appendix, we focus on the NPS-F and also in the Carousel-EDF scheduling algorithms.

B.1 Off-line task-to-processor algorithm

Figure B.1 plots the results of the new overhead-aware schedulability analysis with δ equal to one, $CpmdO$ equal to 100 microseconds, and the remaining parameters set according to the Table 7.1.

Recall that, to compare schedulability analyses, this metric is used indirectly: a schedulability analysis is better than another, if, for a given set of task sets, the former finds that more task sets are schedulable than the latter.

Mixed- and long-period task sets present better results than the short-period task sets. These results are due to: (i) larger time slots created by mixed-period and, especially, long-period task sets and (ii) larger task periods. Note that, overhead costs do not change according to either the length of the time slots or the periodicity of tasks. Then, with larger time slots the associated overhead costs decrease. Figure B.2 shows the mean and the standard deviation of the time slot length for each type of task sets (according to both the period ranges and task utilizations). Further, less frequently arriving tasks also decrease the costs of the task release and also the task switch overheads. However, CPMD overheads could change according to the working set size of each

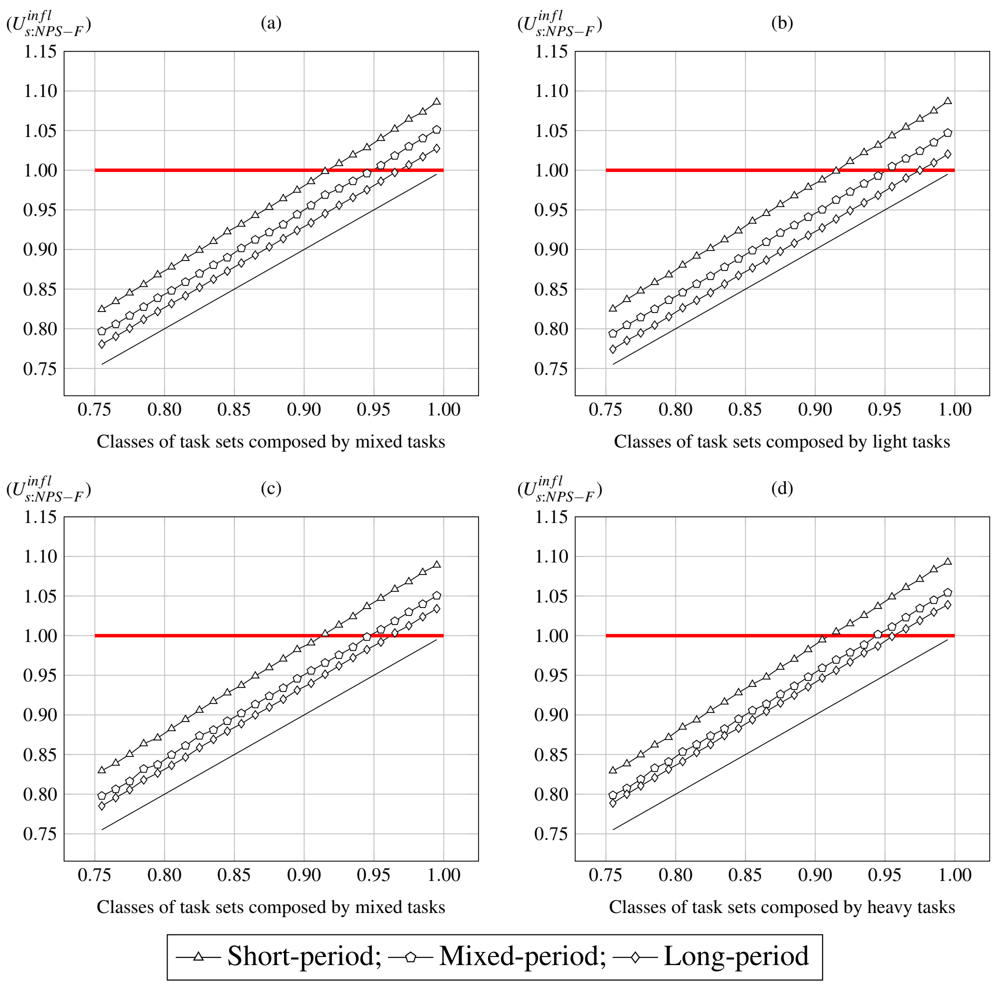


Figure B.1: Comparison of the new NPS-F demand-based overhead-aware schedulability analysis for short-, mixed-, and long-period task sets.

task. It is possible that tasks with larger execution time could require more memory operations, which could imply larger CPMD overheads. As mentioned before, computing the CPMD values is out of the scope of this work.

B.2 Run-time task-dispatching algorithm

Figure B.3 and Figure B.4 plots the percentage of deadlines met for mixed- and long-period task sets scheduled under the NPS-F scheduling algorithm (please note that y-axes do not have the same scale among all insets of Figure B.3 and Figure B.4), respectively. In those figures, we present for each type of tasks (mixed, light, medium, and heavy) a reliability comparison of the NPS-F scheduling algorithm based on the percentage of deadlines met when the run-time task-dispatching settings are computed according to the original, utilization-based, and the new, demand-based

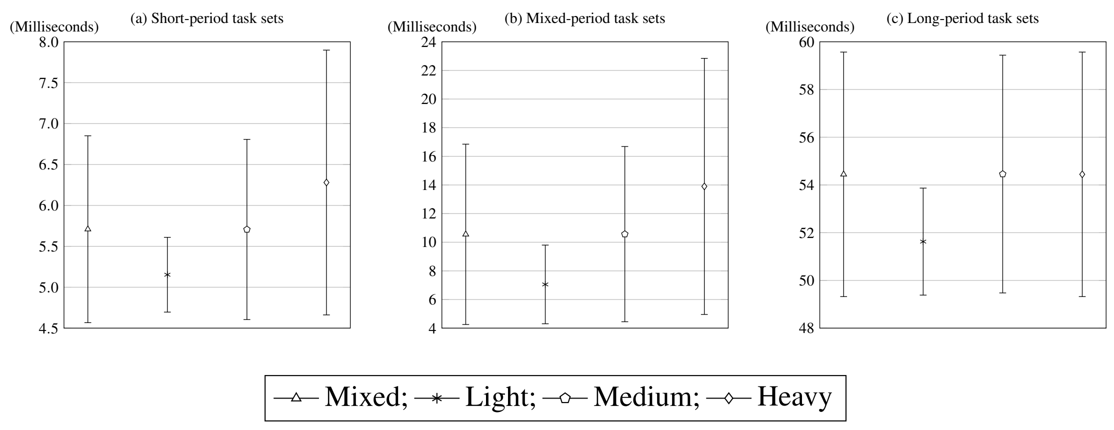


Figure B.2: The mean and the standard deviation of the time slot length for short-, mixed-, and long-period task sets according to the type of tasks.

overhead-aware, schedulability analyses. Despite being considered schedulable by the utilization-based analysis many task sets miss deadlines. We would like to highlight that all task sets composed by light tasks miss deadlines. In our opinion, this is due to the fact that these task sets are composed by more tasks than the other task sets. Thus, a higher number of tasks implies more task-related overheads such as task releases and task switches. And, the pessimism of the utilization factor of the utilization-based schedulability analysis is not sufficient to accommodate such overheads. By comparison almost all task sets composed by heavy tasks, meet their deadlines. These task sets are composed by a small number of tasks and therefore incur fewer task-related overheads. The pessimism associated to the utilization-based schedulability analysis is enough to absorb the overheads. From these results, we can observe that the type of tasks has impact on the performance of the task-dispatching algorithm. In any case, no task set deemed schedulable by the new analysis missed any deadline.

This comparative study shows that the overheads have to be considered into the schedulabilty analysis of the slot-based task-splitting scheduling algorithms to increase their reliability.

Figure B.5 and Figure B.6 plot the percentage of deadlines met for mixed- and long-period task sets scheduled under the RR, FIFO, NPS-F, and Carousel-EDF scheduling algorithms (please note that y-axes do not have the same scale among all insets of Figure B.5 and Figure B.6), respectively. For the sake of presentation, we, in the plots, limit the results to 90%; that is, we do not present the results for task sets that have a percentage of deadlines met below to 90%. As can be observed, NPS-F and Carousel-EDF meet all deadlines, whereas RR and FIFO miss deadlines (and, in fact, even though this is not visible in the plots, this even happens for some task sets with with U_s equal to 50%).

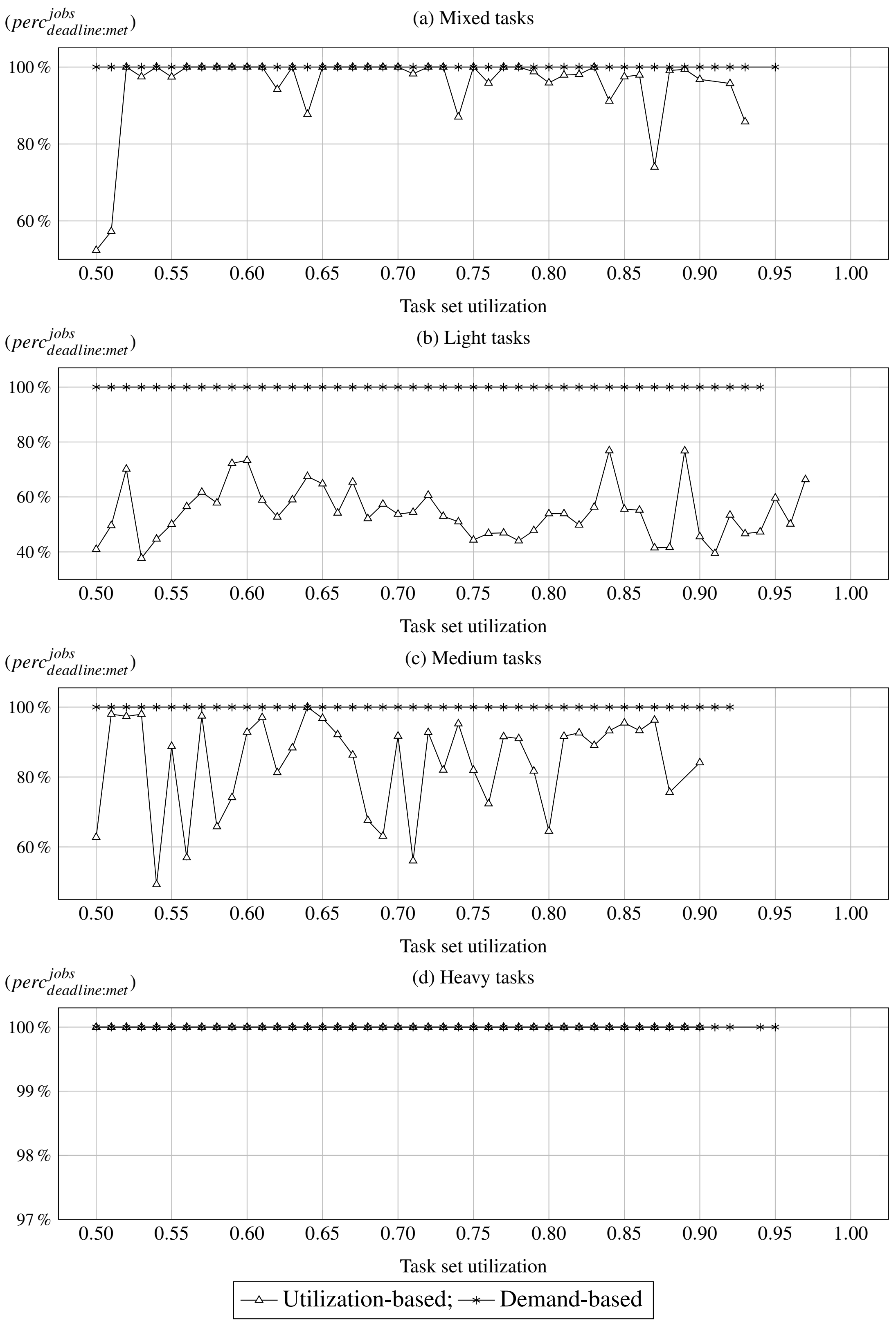


Figure B.3: A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met for mixed-period, in the range [5, 250] milliseconds.

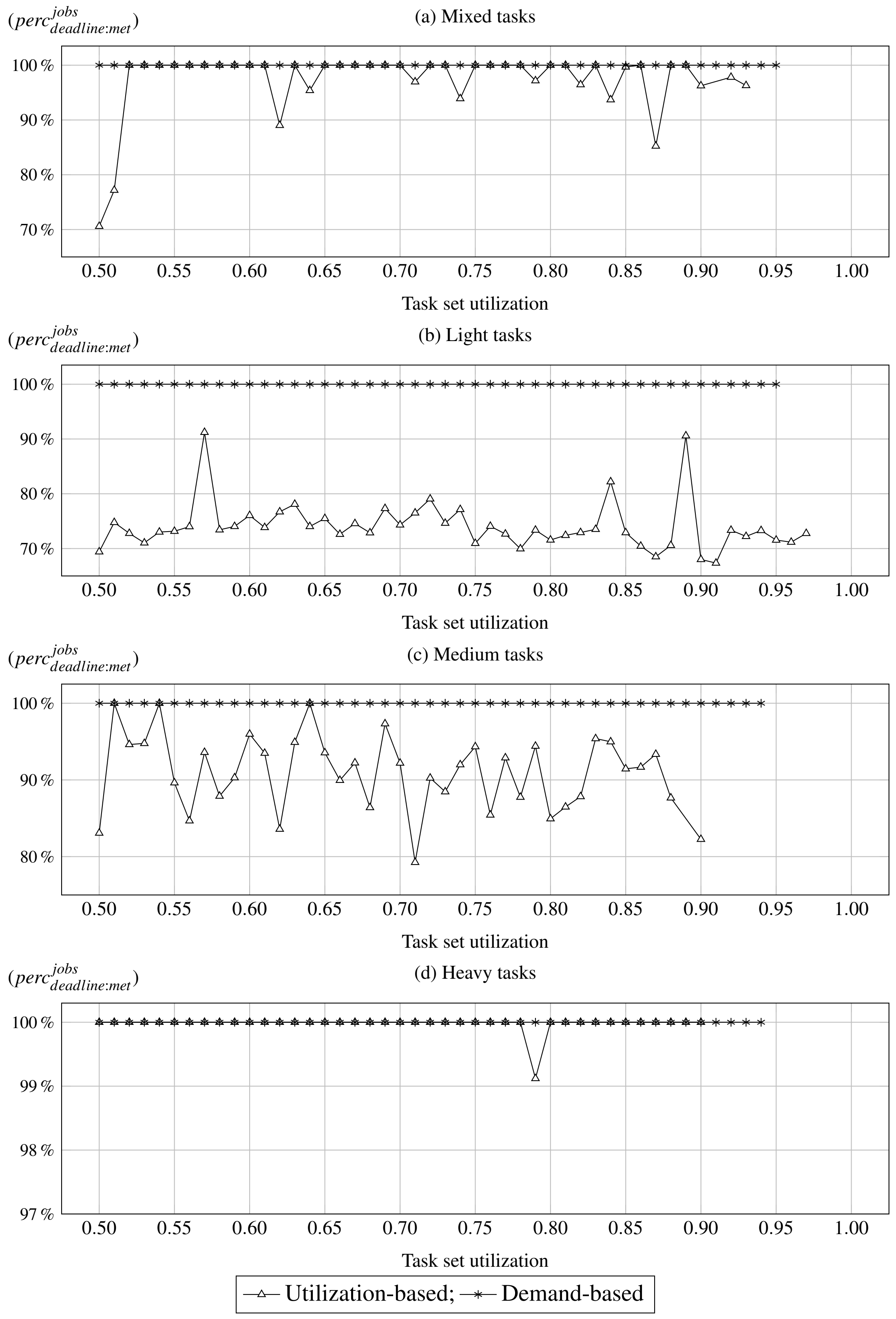


Figure B.4: A comparison between the utilization-based and the demand-based schedulability analyses based on percentage of deadlines met for long-period, in the range [50, 250] milliseconds.

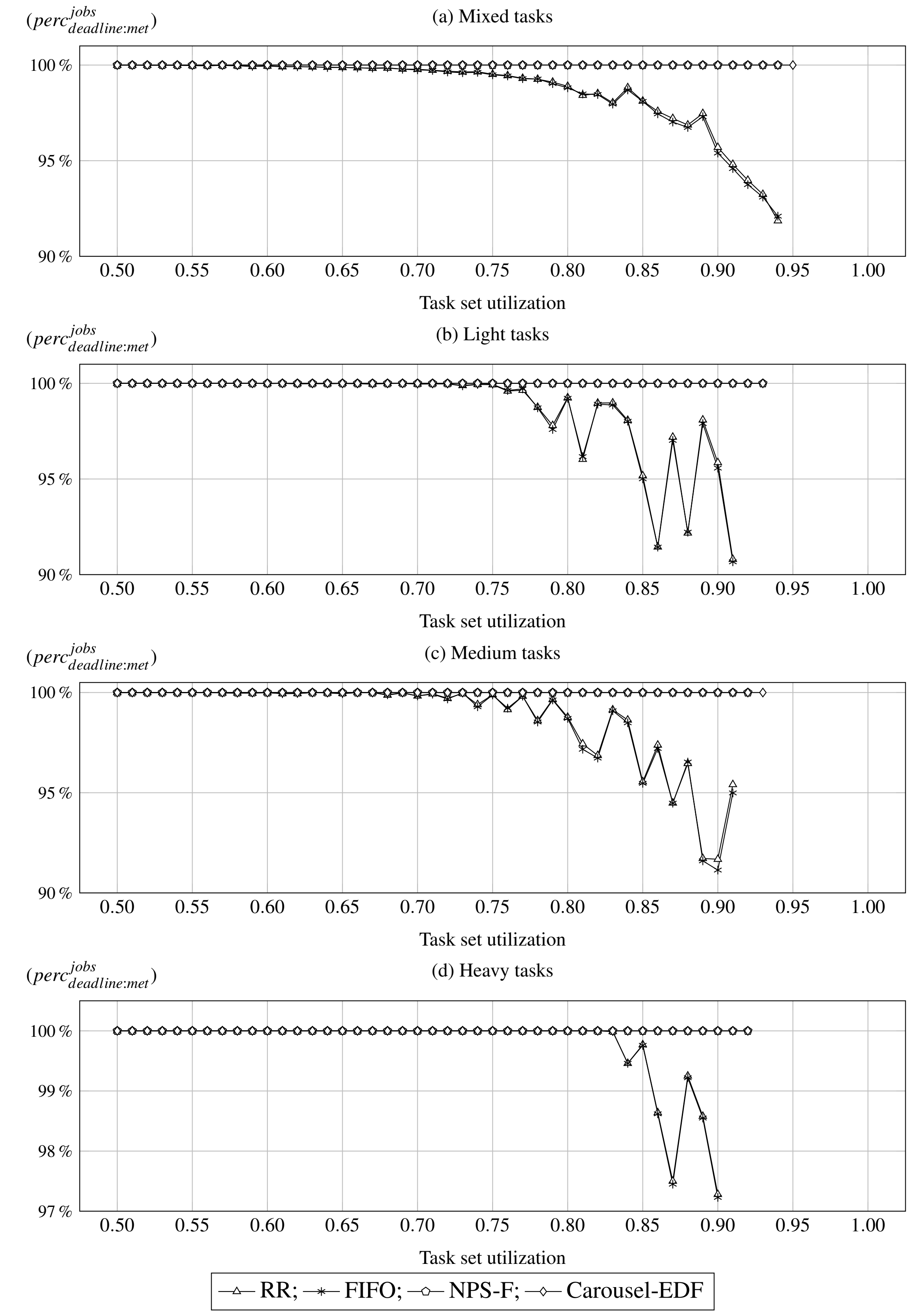


Figure B.5: A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met for mixed-period (in the range [5, 250] milliseconds) task sets.

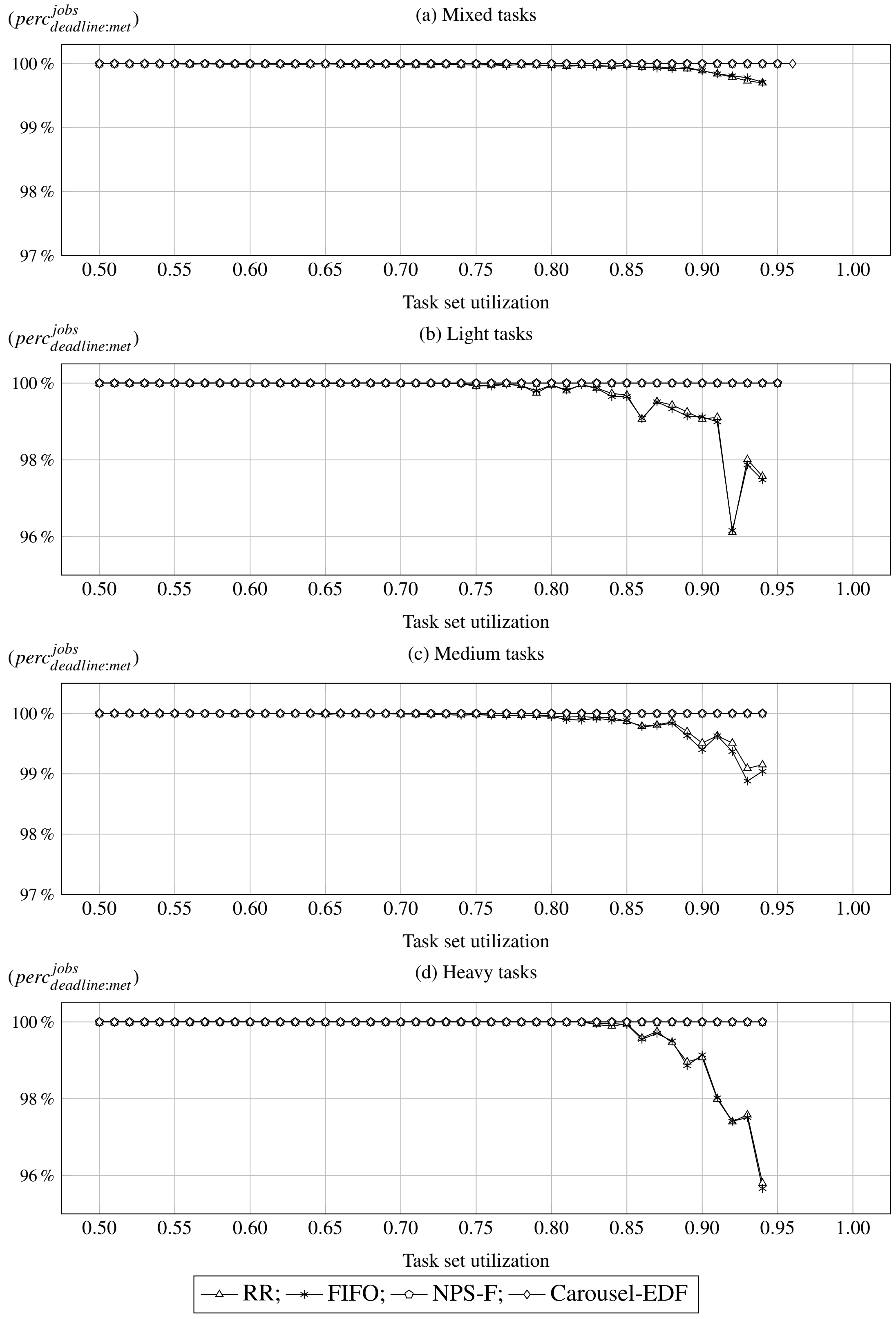


Figure B.6: A performance comparison among RR, FIFO, NPS-F, and Carousel-EDF based on percentage of deadlines met for long-period (in the range [50, 250] milliseconds) task sets.

Appendix C

Papers and materials

C.1 List of papers by the author

This is a list of publications that reflects the results achieved during the development of the research work presented in this dissertation. Most of the results are included in this dissertation.

1. Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Challenges and design principles for implementing slot-based task-splitting multiprocessor scheduling. In *proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10) – Work-in-Progress Session*, San Diego, CA, USA, 2010.

2. Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Implementing slot-based task-splitting multiprocessor scheduling. Technical report HURRAY-TR-100504, CISTER, Polytechnic Institute of Porto (ISEP-IPP), 2010.

3. Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar. Implementing slot-based task-splitting multiprocessor scheduling. In *proc. of 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11)*, pages 256–265, Vasteras, Sweden, 2011.

4. Paulo Baltarejo Sousa, Konstantinos Bletsas, Björn Andersson, and Eduardo Tovar. Practical aspects of slot-based task-splitting dispatching in its schedulability analysis. In *proc. of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, pages 224–230, Toyama, Japan, 2011.

5. Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, and Björn Andersson. On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems. In *proc. of the 13th Real-Time Linux Workshop (RTLWS'13)*, pages 207–218, Prague, Czech Republic, 2011.

6. Paulo Baltarejo Sousa, Nuno Pereira, and Eduardo Tovar. Enhancing the real-time capabilities of the Linux kernel.

- In *proc. of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12) – Work-in-Progress Session*, Pisa, Italy, 2012.
- *SIGBED Rev.*, 9(4):45–48, November 2012.

7. Paulo Baltarejo Sousa, Pedro Souto, Eduardo Tovar, and Konstantinos Bletsas. The Carousel-EDF scheduling algorithm for multiprocessor systems. In *proc. of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages –, Taipei, Taiwan, 2013.

8. Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Akesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. Technical report CISTER-TR-130201, CISTER, Polytechnic Institute of Porto (ISEP-IPP), 2013. (**Under submission to Journal of Real-time Systems**).

C.2 Materials

In the course of this work, we developed the ReTAS (that stands for Real-time TAsk-Splitting scheduling algorithms) framework [ReT12]. The ReTAS framework implements, in a unified way, slot-based task-splitting scheduling algorithms in the Linux kernel. Further, it also implements one reserve-based scheduling algorithm, called Carousel-EDF. All Linux kernel patches and the related tools, which implement the schedulability analyses for slot-based task-splitting and reserve-based scheduling algorithms, can be freely downloaded from `http://webpages.cister.isep.ipp.pt/~pbsousa/retas/`.