PAUL D. E. REGNIER

OPTIMAL MULTIPROCESSOR REAL-TIME SCHEDULING VIA REDUCTION TO UNIPROCESSOR

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. George Marconi de Araujo Lima

Salvador
2012

Sistemas de Bibliotecas - UFBA

Regnier, Paul Denis Etienne.
Optimal multiprocessor real-time scheduling via reduction to uniprocessor / Paul Denis Etienne Regnier. - 2012.
143p. : il.

Orientador: Prof. Dr. George Marconi de Araujo Lima.
Tese (doutorado) – Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia em parceria com a Universidade Estadual de Feira de Santana e Universidade Salvador, Salvador, 2012.

1. Processamento eletrônico de dados em tempo real. 2. Multiprocessadores. 3. Algoritmos. 4. Otimização matemática. 5. Cliente/servidor (Computadores). I. Lima, George Marconi de Araujo. II. Universidade Federal da Bahia. Instituto de Matemática. III. Universidade Estadual de Feira de Santana. IV. Universidade Salvador. V. Título.

CDD - 004.33
CDU - 004.415.2.031.43

TERMO DE APROVAÇÃO

PAUL DENIS ETIENNE REGNIER

OPTIMAL MULTIPROCESSOR REAL-TIME SCHEDULING VIA REDUCTION TO UNIPROCESSOR

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 16 de março de 2012

PROFESSOR E ORIENTADOR GEORGE MARCONI LIMA, PH.D.
Universidade Federal da Bahia

PROFESSOR RÔMULO SILVA DE OLIVEIRA, DR.
Universidade Federal de Santa Catarina

PROFESSOR EDUARDO CAMPONOGARA, PH.D.
Universidade Federal de Santa Catarina

PROFESSOR RAIMUNDO JOSÉ DE ARAÚJO MACÊDO, PH.D.
Universidade Federal da Bahia

PROFESSOR FLÁVIO MORAIS DE ASSIS SILVA,DR.-ING.
Universidade Federal da Bahia

To my daughter Ainá, my son Omin and their loving mother,
Vitória

ACKNOWLEDGEMENTS

Thanks to my advisor, George Marconi Lima, for his support, enthusiasm, and patience. During this seven years of Graduate Studies, MSc and finally, PhD, George has been altogether a wonderful adviser as well as a very nice and enthusiast research partner. I have learnt an enormous amount from working with him, and have thoroughly enjoyed doing so. I am also grateful to him for helping arranging financial support for me throughout my stay at UFBa. I would also like to thank Ernesto Massa, PhD student at UFBa, who I worked with very closely. This research would probably not have come up to lightness without their helpful motivation and dedicated participation.

In addition, I would like to thank my committee members Rômulo Silva de Oliveira, Eduardo Camponogara, Raimundo José de Araújo Macêdo and Flávio Morais de Assis Silva. Each committee member contributed to my dissertation in different and valuable ways.

Professor Aline Maria Santos Andrade deserves my sincere acknowledgements for its initial encouragement and confidence in my capacity to become a Computer Science researcher.

Over the years, it has been a pleasure to be a graduate student at the computer science department at UFBa in large part because of the invaluable contributions of the staff. I thank each member of the administrative and technical staff for the countless ways they assisted me while I was a graduate student. I feel privileged to have had so much support.

Also thanks to my french family who gave me support, education and self-confidence to quit my professional European career and begin a new career of Computer Science researcher at Salvador, Bahia.

Finally, I would like to thank the Brazilian people, their culture and hospitality. In particular, thanks to the guardians of Capoeira, Samba and Candomblé, three traditional cultural quilombos, which are partly responsible for my move from France to Brazil. I am also particularly grateful to my friend and debater, Fernando Conceição, professor and radical. It is visiting him in Salvador, 2003, that I met Vitória, who became my life's companion. In 2006, at the beginning of my Master, she gave birth to Omin, our first son and, in 2008, at the beginning of this PhD, to Ainá, our first daughter. Thanks to the three of them for their love and patience during this long journey to doctorate.

ABSTRACT

Over the last decade, improving the performance of uniprocessor computer systems has been achieved mainly by increasing operation frequency. Recently such an approach has faced many physical limitations such as excessive energy consumption, *chip* overheating, and memory size and memory speed access. To overcome such limitations, the use of replicated hardware components has become a necessary and practical solution. However, dealing with the concurrency for resources caused by parallel execution of programs in recent multi-core and/or multiprocessor architectures has brought about new interesting challenges.

In this dissertation, we focus our attention on the problem of scheduling a set of actions, usually called jobs or tasks, on a *multiprocessor* system. Moreover, we consider this problem in the context of real-time systems, whose specification contains constraints in both time and value domains.

From a synthetic point of view, a real-time system is comprised of three main components:

- A real-time workload, which specifies the tasks that must be executed together with their temporal constraints;
- A real-time platform, comprised of a set processors with well-defined properties on which tasks are executing;
- A scheduling algorithm, in charge of scheduling tasks on the processors of the real-time platform.

We are interested here in *optimal* dynamic priority scheduling algorithms which always find a correct schedule whenever one exists, that is we are interested in algorithms able to schedule systems with real-time workloads that require up to 100% utilization of the real-time platform processors.

Although various optimal solutions exist for uniprocessor systems, those solutions can not be simply exported to systems with two or more processors. Indeed, for such multiprocessor systems, the simple fact that a single real-time task can not execute on two processors simultaneously introduce a dramatic amount of complexity in comparison with the uniprocessor scheduling problem.

Hence, optimal multiprocessor real-time scheduling is challenging. Several solutions have recently been presented for some specific task model. For instance, the *proportionate fairness* (Pfair) approach (BARUAH et al., 1993) has been successfully used as building block of many optimal algorithm for the periodic, preemptive and independent task model with implicit deadlines. However, the Pfair approach enforces deadline equality subdividing the workload of each task proportionally to its execution rate and imposing the deadlines of each task on all other tasks (LEVIN et al., 2010). As a consequence, many tasks execute between every two consecutive system deadlines, possibly leading to more preemptions and migrations than necessary.

As the main contribution of this dissertation, we present RUN (Reduction to UNiprocessor), a new optimal scheduling algorithm for periodic task set with implicit deadlines, which is

not based on proportionate fairness and that reduces the multiprocessor problem to a series of uniprocessor problems.

RUN combines two main ideas. First, RUN uses the key concept of idle scheduling. In a nutshell, at some instant t, RUN schedules a task τ using both the knowledge of its remaining execution time as well as its remaining idle time. Since idle and execution time are the two facets of the same task, we call this scheduling approach *duality*. This leads us to the *Dual Scheduling Equivalence* (DSE), as previously introduced in (REGNIER et al., 2011).

Second, RUN is based on the decrease of the number of tasks to be scheduled by their aggregation into supertasks, that we call *servers*, with accumulated rate no greater than one. Each server is responsible for scheduling its set of *client* tasks, according to some scheduling policy.

Combining servers with duality, RUN leads us to the original notion of *partitioned proportionate fairness (PP-Fair)*, which can be viewed as a weak version of proportional fairness. Briefly, under global fairness, each server of a task set $\mathcal{T}$ is guaranteed to execute for a time proportional to the accumulated rate of the tasks in $\mathcal{T}$. As a consequence, the optimality of the scheduling algorithm for a single server, namely Earliest Deadline First (EDF) here, guarantees that each client's job meets its deadline.

In summary, by combining the Dual Scheduling Equivalence and the PP-Fair approach, RUN reduces the problem of scheduling a given task set on m processors to an equivalent problem of scheduling one or more different task sets on uniprocessor systems. Consequently, RUN significantly outperforms existing optimal algorithms in terms of preemptions with an upper bound of $O(\log m)$ average preemptions per job on m processors. Also, RUN possibly reduces to Partitioned-EDF whenever a proper partition of the task set into servers can be found.

Keywords: Real-Time Systems, Multiprocessor, Scheduling, Optimality, Server

RESUMO

Durante a última década, o melhoramento do desempenho de sistemas de computadores monoprocessador foi principalmente alcançado pelo aumento da freqüência de operação. Recentemente, essa abordagem tem enfrentado muitas limitações físicas, como o consumo excessivo de energia, o superaquecimento dos chips, e a quantidade de memória e velocidade de acesso à memória. Para superar tais limitações, o uso de componentes de hardware replicados tornou-se uma solução necessária e prática. No entanto, lidar com a concorrência pelo uso dos recursos causados pela execução paralela de programas em arquiteturas multicores e / ou multiprocessador recentes gerou novos desafios interessantes.

Nesta dissertação, focamos a nossa atenção sobre o problema do escalonamento de um conjunto de ações, geralmente chamadas de jobs ou tarefas, num sistema *multiprocessador*. Além disso, considera-se este problema no contexto de sistemas de tempo real, cuja especificação contém restrições tanto no domínio do tempo quanto no domínio dos valores.

De um ponto de vista sintético, um sistema de tempo real é constituída por três componentes principais:

- A carga de trabalho de tempo real, que especifica as tarefas que devem ser executadas juntamente com as suas restrições temporais;
- Uma plataforma de tempo real, composto de um conjunto de processador com propriedades bem definidas em que as tarefas são executadas;
- Um algoritmo de escalonamento, responsável pelo escalonamento das tarefas sobre os processadores da plataforma de tempo real.

Estamos interessados aqui em algoritmos *ótimos* de escalonamento baseados em prioridade dinâmica, os quais sempre encontram um escalonamento correto quando existe um, ou seja, estamos interessados em algoritmos capazes de escalonar sistemas com cargas de trabalho de tempo real requerendo até 100% de utilização dos processadores da plataforma de tempo real.

Embora existam várias soluções ótimas para um sistema monoprocessador, essas soluções não podem ser simplesmente exportadas para sistemas com dois ou mais processadores. De fato, para esses sistemas multiprocessador, o simples fato de que uma tarefa de tempo real não possa ser executada em dois processadores simultaneamente introduz uma complexidade relevante em comparação com o problema do escalonamento em um sistema monoprocessador.

Por estas razões, o problema do escalonamento ótimo em sistemas de tempo real multiprocessador é um grande desafio. Várias soluções têm sido recentemente apresentadas para alguns modelos específicos de tarefa. Por exemplo, a abordagem *justiça proporcional* (Proportionate Fairness - Pfair) (BARUAH et al., 1993) tem sido utilizada com sucesso como peça chave para o desenvolvimento de algoritmos ótimos para o modelo de tarefas periódicas, preemptivas, independentes e com deadlines implícitos. No entanto, a abordagem Pfair impõe a igualdade dos deadlines, subdividindo a carga de trabalho de cada tarefa proporcionalmente à sua taxa

de execução e impondo os deadlines de cada tarefa para todas as outras tarefas (LEVIN et al., 2010). Como conseqüência, muitas tarefas executam entre cada dois deadlines consecutivos do sistema, levando possivelmente a mais preempções e migrações do que o necessário.

Como principal contribuição desta dissertação, apresentamos RUN (Redução para Uniprocessor), um novo algoritmo de escalonamento ótimo para conjunto de tarefas periódicas com deadlines implícitas, não baseado na abordagem de justiça proporcional, que reduz o problema multiprocessador para uma série de problemas monoprocessador.

RUN combina duas idéias principais. Primeiro, RUN usa o conceito-chave do escalonamento do tempo ócio. Em suma, em algum instante t, RUN agenda uma tarefa usando tanto o conhecimento de seu tempo de execução restante, bem como o seu tempo ócio restante. Chamamos essa abordagem de escalonamento por *dualidade*, pois os tempos ócio e de execução são duas facetas complementares de uma mesma tarefa. Isto nos leva ao princípio de *Equivalência Dual de Escalonamento*, conforme foi previamente introduzido em (REGNIER et al., 2011).

Segundo, RUN baseia-se na diminuição do número de tarefas a ser escalonadas pela sua agregação em supertasks, os quais chamamos de *servidores*, com taxa acumulada não superior a um. Cada servidor é responsável por escalonar o seu conjunto de tarefas *clientes*, de acordo com alguma política de escalonamento.

Combinando servidores com dualidade, RUN nos leva à ideia original de *justiça proporcional particionada (PP-Fair)*, que pode ser visto como uma versão fraca da justiça proporcional. Brevemente, de acordo com a justiça global, cada servidor de um conjunto de tarefas $\mathcal{T}$ é garantido de executar por um tempo proporcional à taxa acumulada das tarefas de $\mathcal{T}$. Conseqüentemente, a otimalidade do algoritmo de escalonamento utilizado por um único servidor, ou seja Earliest Deadline First (EDF) aqui, garante que os jobs de cada cliente cumpre os seus deadlines.

Em suma, combinando o princípio de *Equivalência Dual de Escalonamento* e a abordagem PP-Fair, RUN reduz o problema do escalonamento de um certo conjunto de tarefas em m processadores para o problema equivalente do escalonamento de um ou mais conjuntos de tarefas diferentes em sistemas monoprocessador. Conseqüentemente, RUN supera significativamente os algoritmos ótimos existentes em termos de preempções com um limite superior de $O(\log m)$ preempções média por jobs em m processadores. Além disso, RUN pode se reduzir a EDF-particionado sempre que uma partição adequado das tarefas em servidores pode ser encontrada.

Palavras-chave: Sistemas de Tempo Real, Multiprocessador, Escalonamento, Otimalidade, Servidor

CONTENTS

List of Figures xviii

List of Tables xix

List of Notations xxii

Chapter 1—Introduction 23

1.1 Real-Time Systems . . . 24
1.2 Real-Time Workload . . . 25
1.2.1 Job Model . . . 25
1.2.2 Task Model . . . 26
1.3 Real-Time Platform . . . 27
1.4 Real-Time Scheduling . . . 28
1.4.1 Schedule . . . 28
1.4.2 Scheduling Algorithm . . . 31
1.5 Optimality in Real-Time Systems . . . 32
1.6 Motivation . . . 33
1.7 Contribution . . . 36
1.8 Structure of this Dissertation . . . 39

Chapter 2—Multiprocessor Scheduling Spectrum 41

2.1 Introduction . . . 41
2.2 Multiprocessor Scheduling Spectrum . . . 42
2.3 Simple Algorithms . . . 43
2.3.1 McNaughton Algorithm . . . 43
2.3.2 Global EDF, LLF . . . 45

2.3.3 EDZL 46
2.4 Optimal Multiprocessor Scheduling 49
2.4.1 Proportionate Fairness 49
2.4.2 Pfair derivatives 50
2.5 An Unfair approach 54
2.6 Idle Scheduling 55
2.6.1 Discussion 57
2.7 Conclusion 57

Chapter 3—Tasks and Servers 59

3.1 Introduction 59
3.2 Fixed-Rate Task Model 60
3.3 Fully Utilized System 61
3.4 Servers 63
3.4.1 Server model and notations 63
3.4.2 EDF Server 66
3.5 Partial Knowledge 68
3.6 Partitioned Proportionate Fairness 68
3.7 Conclusion 71

Chapter 4—Virtual Scheduling 73

4.1 Introduction 73
4.2 DUAL Operation 74
4.3 PACK Operation 77
4.4 REDUCE Operation 80
4.5 Conclusion 84

Chapter 5—Reduction to Uniprocessor (RUN) 87

5.1 Introduction 87
5.2 RUN Scheduling 89
5.3 Parallel Execution Requirement 95
5.4 Conclusion 98

Chapter 6—Assessment 99

6.1 Introduction 99
6.2 RUN Implementation 100
6.3 Reduction Complexity 101
6.4 On-line Complexity 102
6.5 Preemption Bound 103
6.6 Simulation 108
6.7 Conclusion 111

Chapter 7—Conclusion 113

Appendix

Appendix A—Idle Serialization 125

A.1 Frame 125
A.2 Mapping 126
A.3 Level 127
A.4 Idle Serialization 128
A.5 On-line scheduling 130

Appendix B—EDF Server Theorem: another proof 133

B.1 Scaling 133
B.2 Direct Proof of the EDF Server Theorem 134

Appendix C—X-RUN: a proposal for sporadic tasks 137

C.1 Task Model 137
C.2 RUN subtree 137
C.3 X-RUN: Switching Approach 139
C.4 X-RUN: Budget Estimation 140
C.4.1 Weighting Approach 140
C.4.2 Horizon Approach 143

LIST OF FIGURES

1.1 Execution of a Job . . . 25
1.2 Periodic task schedule . . . 26
1.3 Global EDF deadline miss . . . 34
1.4 EDZL deadline miss . . . 35
1.5 Valid schedule . . . 35
1.6 Dual Scheduling Equivalence (DSE) . . . 37
1.7 RUN global scheduling approach . . . 39

2.1 McNaughton schedule on 3 processors. . . . 43
2.2 McNaughton proof illustration . . . 44
2.3 McNaughton non-working schedule Example . . . 45
2.4 EDZL deadline miss . . . 47
2.5 EDZL upper bound example . . . 49
2.6 TL-Plane node example . . . 51
2.7 DP-wrap schedule example . . . 52
2.8 EKG schedule example . . . 54
2.9 EDF map examples. . . . 56
2.10 Minimum and Maximum ISM examples . . . 56

3.1 Fixed-rate task schedule . . . 61
3.2 A two-server set. The notation $X^{(\rho)}$ means that $\rho(X) = \rho$. . . . 64
3.3 Valid schedule of a server whose client miss its deadline . . . 65
3.4 Valid schedule of an EDF-server . . . 65
3.5 Budget management and schedule of an EDF-server . . . 66
3.6 External scheduling constraints . . . 69
3.7 Partitioned Proportionate Fairness Approach . . . 70
3.8 Proportionate Fairness Approach . . . 70

4.1 Dual Scheduling Equivalence (DSE) 74
4.2 Packing example of $\Gamma = \{S_1,\ S_2,\ \ldots, S_7\}$ 78
4.3 Packing and PACK operation example of $\Gamma = \{S_1,\ S_2,\ \ldots, S_7\}$ 79
4.4 Packing, PACK operation, and duality example of $\Gamma = \{S_1,\ S_2,\ \ldots, S_7\}$ 81

5.1 RUN tree example 90
5.2 RUN tree example 90
5.3 RUN schedule example 91
5.4 RUN Tree Scheduling rule example 91
5.5 RUN tree example 93
5.6 RUN schedule example 94
5.7 RUN subtree example 96
5.8 Subtree tree example 97

6.1 A dual JRE 105
6.2 Two Preemptions from one job release 106
6.3 Fraction of task sets requiring 1 and 2 reduction levels 112
6.4 Migrations- and preemptions-per-job varying the processor number 112
6.5 Preemptions per job varying utilization 112

A.1 EDF map examples. 127
A.2 History map and maximum ISM 129
A.3 Minimum and maximum ISM comparison 131
A.4 Minimum and Maximum ISM examples 132

B.1 Deadline miss case 1 135
B.2 Deadline miss case 2 136

C.1 RUN subtree example 138
C.2 Switching between WCS and RUN 140
C.3 The Continuity Argument 141

LIST OF TABLES

2.1 Task set $\mathcal{T}$ (with $D_i = P_i$). 48

4.1 Sample Reduction and Proper Subsets 82

4.2 Reduction Example with Different Outcomes. 84

5.1 One Level Reduction Example 89

5.2 Two Levels Reduction Example 92

6.1 Reduction example of a taskset $\mathcal{T}$ comprised of 11 tasks with identical rate $\frac{7}{11}$, and with total utilization $\rho(\mathcal{T}) = 7$. 108

6.2 Reduction example of a 47-taskset $\mathcal{T}$ comprised of 47 tasks with rate $\frac{30}{47}$, and with total utilization $\rho(\mathcal{T}) = 30$. 109

6.3 Reduction example of a 41-taskset $\mathcal{T}$ comprised of 17 tasks with rate $\frac{14}{23}$, 24 tasks with rate $\frac{15}{23}$ and with total utilization $\rho(\mathcal{T}) = 26$. 110

6.4 Reduction example of a 41-taskset $\mathcal{T}$ comprised of 17 tasks with rate $\frac{14}{23}$, 24 tasks with rate $\frac{15}{23}$ and with total utilization $\rho(\mathcal{T}) = 26$ using the worst-fit bin-packing algorithm. 111

LIST OF NOTATIONS

In this list, X refers either to a real-time task or a server as defined in Chapter 3.

J Real-time job 25

$J.r$ Release instant of job J 25

$J.c$ Worst-case execution time (WCET) of job J 25

$J.d$ Deadline of job J 25

$J.f$ Finish instant of job J 25

$J{:}(r,\ c,\ d)$ A job with release instant r, WCET c and deadline d 25

$\mathcal{J}$ A set of real-time jobs 25

W_J Scheduling window of job J 25

τ_i The ith task in a task set 26

s_i Start time of task τ_i 26

T_i Period of task τ_i 26

C_i Periodic worst-case execution time of task τ_i 26

$\rho(X)$ The execution rate of real-time entity X 26

$\tau_i{:}(C_i, T_i)$ Task with start time zero, WCET C_i and period T_i 26

$\mathcal{T}$ A set of periodic and independent real-time tasks 26

D_i Relative deadline of task τ_i 26

Π Platform of identical and uniform processors 27

m Number of processors in Π 27

P_k The kth processor in Π 27

Σ Schedule function 28

$\Sigma(t)$ Set of jobs in $\mathcal{J}$ executing on Π at time t 28

$e(X, t)$ Remaining execution time of job or task X at time t 28

$l(X,t)$ Laxity of job or task X at time t 29

Δ Job-to-processor assignment function 29

$\bar{\Sigma}$ Assigned schedule 29

$\bar{\Sigma}(t)$ Set of tuples (J,P) such that J executes on P at time t 29

τ_i^* Dual task of task τ_i 36

$\mathcal{T}^*$ Dual set of set $\mathcal{T}$ 36

DUAL Operation which transforms a task set in the set of its dual tasks 37

PACK Operation which aggregates real-time entities into servers 38

$\mathrm{R}(X)$ Set of all release instants of X 60

$\rho(\Gamma)$ Accumulated rate of the set of tasks or servers Γ 61

n Number of real-time tasks to be scheduled on Π 61

$\mathrm{ser}(\mathcal{T})$ Server associated to the taskset $\mathcal{T}$ 63

$\mathrm{cli}(S)$ Set of client tasks of server S 63

r_i The ith element in $\mathrm{R}(X)$ 64

J_i^S The ith budget job of server S 64

$e(J_i^S,t)$ The budget of server S at time t 64

$X^{(\rho)}$ X has rate ρ *i.e.*, $\rho(X)=\rho$ 64

$J_{i,j}$ The jth job of τ_i 65

$\eta_\Gamma(t,t')$ Execution demand of task set Γ within a time interval $[t,t')$ 68

φ Bijection which associates a server S with its dual server S^* 76

$f(G)$ Image of subset $G \subset E$ by f *i.e.*, $f(G)=\{f(x),x\in G\}$ 77

A A packing algorithm 77

$\pi_A[\Gamma]$ Packing of the set of server Γ 77

$\mathcal{R}_A$ Equivalence relation on Γ induced by partition $\pi_A[\Gamma]$ 78

$p_A(S)$ The equivalence class of S 79

$\sigma_A(S)$ Server which schedules the servers in $\pi_A[S]$ 79

ψ Composition of the DUAL and PACK operations *i.e.*, $\psi=\varphi\circ\sigma$ 80

ψ^i Iterated ψ operator with $\psi^0=Id$ and $\psi^i=\psi\circ\psi^{i-1}$ 81

$\{\psi^i\}_i$ Reduction sequence 81

$\psi^i(\Gamma)$ Reduction level i of server set Γ 81

Chapter 1

A real-time system is an information processing system which has to respond to externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response.

Alan Burns and Andy Wellings, 2009

INTRODUCTION

Over the last decade, improving the performance of uniprocessor computer systems has been achieved mainly by increasing operation frequency. Recently such an approach has faced many physical limitations such as excessive energy consumption, *chip* overheating, and memory size and memory speed access. To overcome such limitations, the use of replicated hardware components has become a necessary and practical solution. However, the organization of the concurrent use of hardware components by parallel software programs is a challenging task which requires further investigation.

Indeed, dealing with the concurrency for resources caused by parallel execution of programs in recent multi-core and/or multiprocessor architectures has brought about new interesting challenges. For instance, memory sharing must be organized to ensure data consistency between different levels of cache and memory. Also, the organization of communication between the various hardware components must take competition for resources into account without compromising aspects related to timeliness or throughput. In this context, the scheduling of processes or threads must be optimized to ensure correctness and efficient resource usage.

This dissertation focuses on the problem of scheduling a set of actions, usually called jobs or tasks, on a multiprocessor system. More specifically, we consider this problem in the context of real-time systems, whose specification contains constraints in both time and value domains.

Structure of the chapter

We begin by precisely defining a real-time systems in Section 1.1. Then, we define the three main components of a real-time system, *i.e.,* the real-time workload in Section 1.2, the real-time platform in Section 1.3 and the real-time schedule of a set of tasks in Section 1.4. We dedicate Section 1.5 to the clear understanding of the optimality of a scheduling algorithm relatively to a real-time system. This allows us to present the motivation as well as the contributions of this dissertation in Section 1.6 and Section 1.7, respectively. We finish this chapter giving an overview of the structure of this dissertation in Section 1.8.

1.1 REAL-TIME SYSTEMS

According to (BURNS; WELLINGS, 2009), a real-time system is

> "*an information processing system which has to respond to externally generated input stimuli within a finite and specified period: the correctness depends not only on the logical result but also on the time it was delivered; the failure to respond is as bad as the wrong response*".

As a consequence, for real-time systems, all or part of the processing of tasks must be realized within pre-defined deadlines that must be met in order for the system to be correct. For instance, in an Automatic Braking System (ABS), the value of pressure to be applied on each wheel must be computed in a bounded time after the driver step on the brake pedals otherwise an accident may occur. Thus, the tasks which are responsible for sensing, controlling and actuating on the ABS must be properly scheduled in time. Deciding when each of these tasks executes is strongly related to the system correctness.

From a synthetic point of view, a real-time system is comprised of three main components:

- A real-time workload, which specifies the tasks that must be executed together with their temporal constraints;
- A real-time platform, comprised of a set of processors with well-defined properties, on which tasks execute;
- A scheduling algorithm, in charge of scheduling tasks on the processors of the real-time platform.

In the following sections, we formally define each of these components.

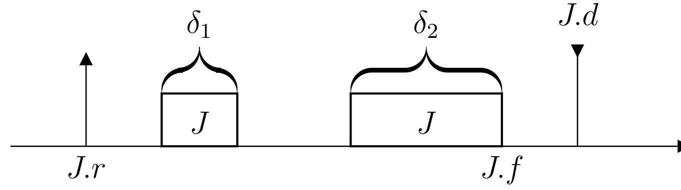


Figure 1.1. Representation of an execution of job J where $\delta_1 + \delta_2$ is the exact amount of execution time needed by J. Note that J does not execute before $J.r$, $\delta_1 + \delta_2 \leqslant J.c$ and $J.f \leqslant J.d$. Therefore, J meets its deadline.

1.2 REAL-TIME WORKLOAD

1.2.1 Job Model

In general, the processing requirement of a set of applications executed by a real-time system is specified by a set of execution quanta, each of which called job.

Definition 1.2.1 (Job)**.** *A real-time* job J*, or simply* job*, is a finite sequence of instructions to be executed on one or more processors with a release instant* $J.r$*, a worst-case execution time (WCET)* $J.c$ *and a deadline* $J.d$*.*

Also, we denote $J.f$ the finish instant of job J, *i.e.,* the time at which J completes its execution.

Given an arbitrary set of jobs $\mathcal{J}$ executing on a real-time system platform, the four parameters (i) release instants; (ii) worst-case execution time; (iii) finish instants; and (iv) deadlines of jobs are related in the following sense. In order for the system to be correct, each job J in $\mathcal{J}$ must execute after its release instant $J.r$ and must meet its deadline $J.d$, *i.e.,* it must finish its execution at some instant before $J.d$ ($J.f \leqslant J.d$). Also, when J completes its execution at time $J.f$, it must have executed for an amount of time δ less than $J.c$ ($\delta \leqslant J.c$) during $[J.r, J.f]$. In a synthetic view, we say that time interval $[J.r, J.d)$ is the *scheduling window* of J and we denote $W_J = [J.r, J.d)$. Whenever needed, we use the more concise notation $J{:}(r,\ c,\ d)$ to specify a particular job with release time r, WCET c and deadline d.

The graphical representation of an execution of a job J which meets its deadline is given by Figure 1.1. In our graphical notation, upside arrows indicate release instants, downside arrows indicate deadlines and framed boxes represent job executions. If not specified, execution can take place on one or more processors.

In this dissertation, jobs are assumed to be *independent*, *i.e.,* there exist neither dependency between the parameter of any two jobs nor synchronizations between their relative execution. Also, the unique shared resources are the processors.

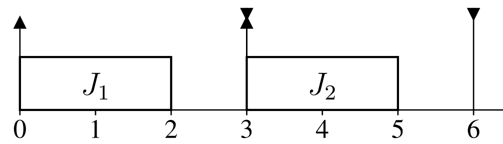


Figure 1.2. Schedule of periodic task τ:$(2, 3)$.

1.2.2 Task Model

Many real-time systems applications, like control systems, have periodic or quasi-periodic execution time requirements. In such systems, the real-time workload can be specified in terms of recurring tasks. Each such a task has a start time, at which it releases the first of an infinite sequence of jobs.

According to the model described in a seminal paper (LIU; LAYLAND, 1973), each task releases its jobs periodically and the deadline of a job is precisely equal to the release instant of the next job.

In other words, according to this task model, referred to as the periodic task model with implicit deadline (PID), a task τ_i is completely characterized by its start time s_i, its period T_i, and its periodic worst-case execution time (WCET) C_i. When all tasks share the same start time, the task system is said *synchronous* and all start times are assumed equal to zero. We simply denote τ_i:(C_i, T_i) a task τ_i with start time zero, WCET C_i and period T_i. Also, we denote $\rho(\tau_i) = C_i/T_i$ the execution rate of τ_i.

For example, Figure 1.2 illustrates the schedule of the first two jobs J_1:$(0,\ 2,\ 3)$ and J_2:$(3,\ 2,\ 6)$ of periodic task τ:$(2, 3)$ on a single processor.

The sporadic task model with implicit deadlines is a generalization of the periodic model. According to this model, hereafter referred as the Liu and Layland (LL) task model, the release instants of two successive jobs of a task are separated by a minimal inter-release time T_i, sometimes called period for historical reasons.

Allowing for explicit deadlines leads to the sporadic task model with explicit deadline, simply referred to as sporadic task model (MOK, 1983). According to this model, each task is still specified by its start time s_i, worst-case execution time C_i and minimal inter-release time T_i. However, each task has a new parameter, its relative deadline D_i which is used to calculate the absolute deadline of a job at runtime. Whenever a job of a task τ_i is released at time $J.r$, its "absolute" deadline $J.d$ is explicitly calculated as $J.d = J.r + D_i$.

Many other task models have been proposed to represent real-time systems with specific characteristics (BARUAH et al., 1999). A comprehensive description can be found elsewhere (FISHER, 2007).

1.3 REAL-TIME PLATFORM

A real-time multiprocessor platform is heterogeneous when different processors may have different execution speed, or even, different hardware. Also, a non-uniform processor may execute different jobs at different speeds while a uniform processor executes all jobs at the same speed. Thus, on a multiprocessor platform comprised of uniform processors, each processor has a speed, at which it executes all jobs, which is possibly different from the speed of another processor.

In a platform comprised of identical processors, it is assumed that all processors are uniform, *i.e.,* all processors have the same speed, usually normalized to one. Hence, all jobs execute at the same speed, independently of the processor on which it is scheduled (FUNK, 2004).

Besides its speed, another important property of a processor is its capability to preempt jobs during their execution. In a *non-preemptive* processor, a scheduled job must execute continuously until completion while in a *preemptive* processor the execution of a job can be interrupted at any time to execute a higher priority job. Note that while preemption of jobs may ease the conception of a scheduling algorithm and allows for an efficient utilization of the processors, they may also result in a significant execution time overhead (BUTTAZZO, 2005).

Finally, jobs and/or task may be allowed to migrate between different processors during their execution. Approaches which do not impose any restriction on task migration are usually called *global* scheduling. Those that do not allow task migration are known as *partition* scheduling since each task is assigned to only one processor. Although partition-based approaches make it possible to apply the results for uniprocessor scheduling straightforwardly, they have two main disadvantages. First, they are not applicable for task sets which cannot be partitioned. Second, the assignment of tasks to processors is a bin-packing problem, which is NP-Hard in the strong sense (GAREY; JOHNSON, 1979).

On the other hand, under *global* scheduling, tasks are enqueued in a single global queue according to some well-defined order. Whenever a processor becomes available, the first job in the queue is picked up to execute. Such approaches can provide effective use of a multiprocessor architecture although with possibly higher implementation overhead (CARPENTER et al., 2004).

In this dissertation, we denote Π a platform comprised of $m \geqslant 2$ identical processors and P_k the kth processor in Π. As a consequence and without loss of generality, the execution speed of each processor is assumed equal to 1 execution quantum per time unit.

Also, we focus on global scheduling and we assume a preemptive job model with migration, *i.e.,* jobs can be preempted at any time and a preempted job may resume its execution on any processor of the platform. However, we make the somehow incorrect but usual assumption

that preemption and migration take zero time. In an actual system, measured preemption and migration overheads can be accommodated by adjusting the execution requirements of tasks.

1.4 REAL-TIME SCHEDULING

1.4.1 Schedule

Given a set of of jobs (or tasks) $\mathcal{J}$ to be executed on platform Π, a *schedule* of $\mathcal{J}$ on Π usually specifies which jobs of $\mathcal{J}$ execute on which processor of Π at all times during the system execution. However, since we assume a multiprocessor platform Π comprised of $m \geqslant 2$ identical processors, we adopt a slightly different definition for a schedule.

In this dissertation, we distinguish two nested steps for a scheduling procedure at some scheduling instant t, namely the *scheduling step* and the *assigning step*.

Scheduling Step

In the scheduling step, which always precedes the assigning step, a subset $\mathcal{J}'$ of jobs in $\mathcal{J}$ is chosen to execute.

Definition 1.4.1 (Schedule). *For any set of jobs $\mathcal{J}$ on a platform of $m \geqslant 1$ identical and uniform processors, a* schedule Σ *is a function from all non-negative times t to the power set of $\mathcal{J}$ such that $\Sigma(t)$ is the subset of jobs in $\mathcal{J}$ executing at time t.*

Within an executing schedule Σ, $e(J,t)$ denotes the maximum work remaining for job J at time t, so that $e(J,t)$ equals $J.c$ minus the amount of time that J has already executed as of time t. Whenever no confusion is introduced doing so, we also denote $e(\tau,t)$ the remaining execution time of task τ at time t. Formally, if $\mathbb{1}_{\Sigma(t)}$ is the indicator function of $\Sigma(t)$ defined by

$$\mathbb{1}_{\Sigma(t)}(J) = \begin{cases} 1 & \text{if } J \in \Sigma(t) \\ 0 & \text{otherwise} \end{cases}$$

then, the execution requirement of a job J at time t can be expressed as

$$e(J,t) = J.c - \int_{J.r}^{t} \mathbb{1}_{\Sigma(u)}(J)du$$

Some assumptions apply to schedules in order for the system to be legal. First, a job can neither execute prior its release instant nor after its finishing instant. Second, there can be no more jobs executing than processors at any time, or, in other words, a processor can not execute more than one job at any time. We summarize those restrictions in the following definition:

Definition 1.4.2 (Legal Schedule)**.** *The schedule* Σ *of a set of jobs* $\mathcal{J}$ *on a platform* Π *of* $m \geqslant 1$ *processors is* legal *if it satisfies the following:*

(i) If a job J *is scheduled at time* t *(*$J \in \Sigma(t)$*), then the release instant of* J *is not after* t *(*$J.r \leqslant t$*) and the remaining execution time of* J *at* t *is greater than zero (*$e(J, t) > 0$*);*

(ii) No more than m *jobs execute at any time,* i.e., $|\Sigma(t)| \leqslant m$ *for all* t.

Note that this definition of a legal schedule also holds when $\mathcal{J}$ is specified as a recurrent task system $\mathcal{T}$ as stated in Definition 1.2.2.

The laxity of job J, denoted $l(J, t)$ is defined as the maximum time that the execution of job can be delayed without compromising its correct completion by its deadline. Formally, $l(J, t) = J.d - t - e(J, t)$. Whenever no confusion is introduced doing so, we also denote $l(\tau, t)$ the laxity of task τ at time t.

Assigning Step

In this step, the jobs chosen to execute at time t are allocated to processors in Π.

Definition 1.4.3 (Assignment)**.** *For any set of jobs* $\mathcal{J}$ *on a platform* Π *of* m *identical processors, an* assignment *function* Δ *assigns a job scheduled at time* t *to a processor in* Π.

We define *an assigned schedule*, denoted $\bar{\Sigma}$, as the composition of the schedule function Σ with an assignment function Δ ($\bar{\Sigma} = \Delta \circ \Sigma$). Formally, at any non-negative time t, $\bar{\Sigma}(t)$ is the set of tuples (J, P) with $J \in \mathcal{J}$ and $P \in \Pi$ such that J executes on P at time t.

Note that an assigned schedule corresponds to the usual definition of schedule. However, we find it convenient to separate both scheduling and assigning steps since this will allow for a more concise description of our original scheduling approach.

Also, since processors are identical and since migration is allowed with no penalty, the job-to-processor assignment function can be considered as an implementation problem which can be solved straightforwardly according to some previously established goal. In Chapter 6, we present an assignment procedure devised to minimize preemptions.

Considering a legal schedule Σ, then the following restriction must be satisfied by the assignment function Δ in order for the system to be legal: *a job can only execute on a single processor at any time*. We state this restriction as follows:

Definition 1.4.4 (Legal Assigned Schedule)**.** *Let* Σ *be a legal schedule of a set of jobs* $\mathcal{J}$ *on a platform* Π *of* $m \geqslant 1$ *processors. Then, the assigned schedule* $\bar{\Sigma}$*, composition of* Σ *with an assignment* Δ*, is* legal *if for any two tuples* (J, P) *and* (J', P') *in* $\bar{\Sigma}(t)$*,* $J = J'$ *if and only if* $P = P'$.

It is important to emphasize that this latter restriction, which states that there must be no parallel execution of the same job on different processors, is the main restriction specific to multiprocessor systems compared to uniprocessor systems. As a matter of fact,

> 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.*

as already stated by (LIU, 1969) as quoted in (BARUAH, 2001).

In this dissertation, we only consider "legal" assignment, according to which, given a legal schedule as input, a legal assigned schedule is produced as output. It is easy to see that there always exists such a legal assignment. Indeed, since a legal schedule chooses no more than m jobs to execute at any time, a simple "legal" assignment is one which allocates a single job per processor at any time in an arbitrary manner. Thus, in the remainder of the dissertation, we will omit the assignment step whenever no confusion is introduced doing so.

Among the legal schedules of a job set, we further distinguish those schedules of interest for real-time systems *i.e.,* schedules in which all jobs meet their deadlines.

Definition 1.4.5 (Valid Schedule)**.** *A legal schedule* Σ *of a job set* $\mathcal{J}$ *is* valid *if all jobs in* $\mathcal{J}$ *meet their deadlines,* i.e., *if for all* J *in* $\mathcal{J}$*,* $J.f \leqslant J.d$*.*

The problem of generating valid schedules of an arbitrary job set on a real-time platform Π raises two different questions. First, given an arbitrary job set $\mathcal{J}$, is it feasible, *i.e.,* is there a valid schedule of $\mathcal{J}$ on Π? This decision problem, referred to as the *feasibility problem*, is known to be NP-complete for arbitrary job sets (GAREY; JOHNSON, 1979). However, a simple feasibility criterion may be found for specific task/job models. For instance, for a set of jobs $\mathcal{J}$ generated by a set of periodic tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ with execution time C_i, period T_i and implicit deadlines, it was shown by (LIU; LAYLAND, 1973) that

$$\sum_{i=1}^{n} \rho(\tau_i) \leqslant 1$$

is a sufficient and necessary feasibility condition of $\mathcal{J}$ on a single processor. This result was later extended to identical multiprocessor platform (HORN, 1974; BARUAH, 2001) meaning that

$$\sum_{i=1}^{n} \rho(\tau_i) \leqslant m$$

is a sufficient and necessary feasibility condition of $\mathcal{J}$ on a platform comprised of m identical processors.

The second question, referred to as the *scheduling problem* can be expressed as follows. Assuming that $\mathcal{J}$ is feasible on Π, is it possible to devise a *scheduling algorithm*, say SA,

that produces a valid schedule of $\mathcal{J}$ on Π? And, having devised SA, is it possible to find a schedulability criterion which allows to decide whether another different job set is schedulable by SA?

In general, the answer to this question is hard and sometimes negative. However, solutions to both problems are known for some specific classes of task sets.

1.4.2 Scheduling Algorithm

Definition 1.4.6. *A* scheduling algorithm *is a procedure which admits a set of jobs $\mathcal{J}$ as input and produces a legal schedule Σ as output.*

A scheduling algorithm is *work-conserving*, or alternatively, non-idling, if it never idles the processor whenever there exist some jobs ready to execute in the system.

Definition 1.4.7 (Schedulability)**.** *A task set $\mathcal{T}$ is* schedulable *by a scheduling algorithm $\mathcal{A}$ if the legal schedule Σ of $\mathcal{T}$ by $\mathcal{A}$ is valid,* i.e., *if all tasks in $\mathcal{T}$ meets their deadlines in Σ.*

Definition 1.4.8 (Feasibility)**.** *A task set $\mathcal{T}$ is* feasible *if $\mathcal{T}$ is schedulable by some scheduling algorithm.*

Depending on the task and system model, *i.e.,* the set of assumptions about jobs, tasks and the relying multiprocessor system adopted, different scheduling approaches can be investigated. For instance, according to the periodic task model, all release time and deadlines are completely specified before the execution of the system. As a consequence, it is possible to find a valid schedule of the system *off-line*, *i.e.,* before its execution. Such a schedule can then be easily implemented at execution time through a table-driven algorithm.

However, such an off-line scheduling approach may be impracticable when part or all of the specification of the system is only known at execution time. This is the case, for instance, when release instants are not known before the execution of the system, as in the sporadic model. Also, the explicit deadline of a job may be only known at its release instant. In those systems partly specified, an *on-line* scheduling procedure is required in order to decide which jobs must execute on which processor at any time.

In general, a scheduling algorithm makes its choices based on the relative value of some parameter, used to define the priority of the jobs. When the priority of each job is calculated (or pre-set) in advance and remains fixed during the whole operation of the system, the scheduling policy is said to have *static* priority. For instance, the rate-monotonic scheduling algorithm (RM) proposed in (LIU; LAYLAND, 1973) is a static priority algorithm which defines the priority of a job as the inverse of the period of its generating task. Thus, jobs of tasks with lower periods turn to have higher priorities. Although such a priority policy has the advantage

of simplicity and allows for an off-line table-driven approach, it fails to produce a valid schedule of some feasible task set.

Another class of scheduling algorithms uses *dynamic* priorities for jobs defined at execution time. For instance, the Deadline Algorithm, also proposed in (LIU; LAYLAND, 1973) and nowadays best known as Earliest Deadline First (EDF) algorithm, is a dynamic priority algorithm according to which the priority of a job is inversely proportional to the value of its absolute deadline. Thus, jobs with earlier deadlines have higher priorities than jobs with later deadlines.

As discussed in (BUTTAZZO, 2005), an off-line fixed-priority algorithm, like rate-monotonic, has the advantages of implementation simplicity and low runtime overhead. On the other hand, dynamic on-line priority algorithms like EDF usually achieve a better utilization of processors.

In this dissertation, we focus our attention on those latter dynamic on-line priority algorithms which achieve a full utilization of the processors. However, even if we assume a fully preemptive and migrating task model, we are interested in algorithms with low preemptions and migrations overheads.

1.5 OPTIMALITY IN REAL-TIME SYSTEMS

As previously discussed, the description of real-time systems is done through a set of assumptions upon the real-time workload and the multiprocessor platform. This set of assumptions defines a model of the system and allows for eventually proving interesting properties for some particular class of scheduling algorithms.

Among those properties, one of the most relevant and considered is the *optimality* of the scheduling algorithm, precisely defined as follows.

Definition 1.5.1. *A scheduling algorithm is said to be* optimal *regarding a real-time system model if it can produce a valid schedule for any feasible real-time job set possibly specified in this model.*

In the realm of uniprocessor systems, many optimal algorithms are know regarding the different task model described in Section 1.2.2. For example, the optimality of the EDF dynamic priority algorithm regarding the periodic, preemptive and synchronous task model with implicit deadlines is proved in (LIU; LAYLAND, 1973), since it achieves full-utilization of the system, as mentioned in Section 1.2.2. The optimality result upon EDF was later extended to the sporadic job model, for both preemptive and not preemptive systems by (DERTOUZOS, 1974; GEORGE et al., 1996).

The least laxity first (LLF) algorithm proposed by (MOK, 1983) is another example of

optimal uniprocessor algorithm for sporadic task model when preemption is allowed. However, the LLF algorithm has the drawbacks to require a possibly infinite number of preemptions under a continuous time model (HOLMAN, 2004).

In a recent work, a characterization of all possible on-line preemptive scheduling algorithm on one processor is given (UTHAISOMBUT, 2008). However, it is still an open problem to determine whether a similar characterization can be found for optimal algorithms on platforms comprised of two or more processors. As a matter of fact, it has been known since the end of the eighties that no optimal on-line algorithm exists when considering a platform comprised of two or more processors for an arbitrary collection of independent jobs when deadlines and release times are not known *a priori* (HONG; LEUNG, 1988; DERTOUZOS; MOK, 1989). This result was recently extended to the sporadic task model (FISHER et al., 2010). However, optimality can be achieved for multiprocessor preemptive systems for more restrictive task model, like the LL model for instance.

Since there exists no on-line optimal algorithm for the sporadic job model, the weaker notion of *suboptimality* was introduced by (CHO et al., 2002).

Definition 1.5.2. *A preemptive algorithm is suboptimal if it successfully schedules any feasible set of ready jobs, where a ready job at time* t *is a job that has been released at or before* t*.*

For instance, the Least Laxity First (LLF) algorithm is suboptimal (DERTOUZOS; MOK, 1989) on any number of processors.

1.6 MOTIVATION

Considering that the multicore / multiprocessor revolution as described in (BERTOGNA, 2007) is an overwhelming reality and since real-time systems are nowadays present in a wide variety of fields, such as control systems, environmental monitoring, avionic and automotive applications, there exists a need to extend well-established solutions to the feasibility and scheduling problems in uniprocessor systems to multiprocessor systems. However, the real-time multiprocessor scheduling problem is commonly acknowledged to be much more complex than the real-time uniprocessor scheduling problem. Indeed, multiprocessor scheduling solutions tend to be computationally more expensive and complicated than those used for uniprocessor scheduling.

A straightforward approach to export uniprocessor scheduling results to multiprocessor scheduling systems consists in *partitioning* the task set by statically assigning each task to a static and single processor. In such an approach, each processor has a fixed set of tasks allocated to it during the execution of the system. As a consequence, no migration of jobs is necessary and the multiprocessor scheduling problem is reduced to m uniprocessor schedul-

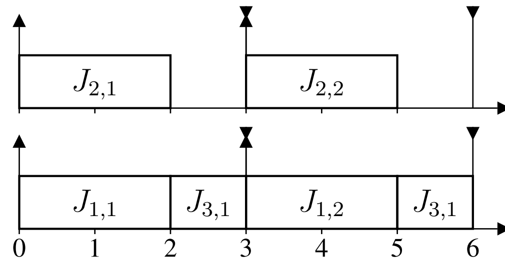


Figure 1.3. Assuming a partitioned approach or global EDF scheduling, the first job $J_{3,1}$ of τ_3 misses its deadline 6.

ing problems. Although elegant and practical, *partitioned* approaches have the drawbacks of achieving a low utilization of the system, guaranteeing only 50% utilization in the worst case (KOREN et al., 1998).

On the other hand, global scheduling approaches can achieve full utilization by migrating tasks between processors, at the cost of increased runtime overhead. For example, consider a 3-task set $\mathcal{T} = \{\tau_1\!:\!(2,3), \tau_2\!:\!(2,3),\ \tau_3\!:\!(4,6)\}$ to be schedule on a two-processor system. Since

$$\sum_{i=1}^{3} \rho(\tau_i) \ = \ 2$$

$\mathcal{T}$ is feasible on two processors.

However, if the jobs of tasks τ_1 and τ_2 are first scheduled on the two processors and run to completion, then the third task cannot complete on time, as illustrated in Figure 1.3 where $J_{i,k}$ is the kth job of task τ_i. For instance, this would be the case in a partitioned approach or using global EDF. Indeed, global EDF schedules the earliest deadline job sorted from a single global queue on a processor whenever it becomes idle.

If tasks are allowed to migrate, even global EDZL, which raises the priority of a zero-laxity job to the highest priority in the system (CHO et al., 2002), would fail to schedule this simple task set as illustrated in Figure 1.4. Indeed, until time 3, no job reaches zero-laxity. As a consequence, $J_{1,1}$ and $J_{2,1}$ which both have earliest deadline 3 at time 0 are scheduled continuously during interval $[0,2)$. Also, by the non-parallel execution constraint, $J_{3,1}$ can only execute on one of the two processors during $[2,3)$ and an idle slot occurs on one processor during time interval $[2,3)$. When $J_{3,1}$ reaches zero-laxity at time 3, the idle slot already have happened and either $J_{1,2}$ or $J_{2,2}$ misses its deadline at time 6.

However, if tasks may migrate, there exists a valid schedule of $\mathcal{T}$ in which all jobs of these three tasks can meet their deadlines, as illustrated in Figure 1.5.

Note that, if all jobs share the same deadline, *i.e.,* if job $J_{3,1}$ is split into two subjobs,

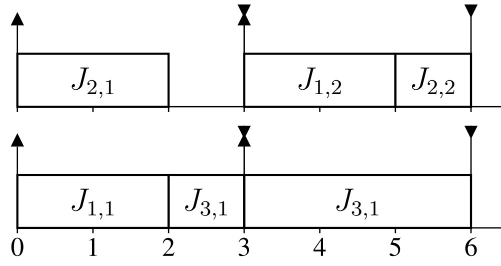


Figure 1.4. Under EDZL, either job $J_{1,2}$ of τ_1 or job $J_{2,2}$ of τ_2 misses its deadline 6.

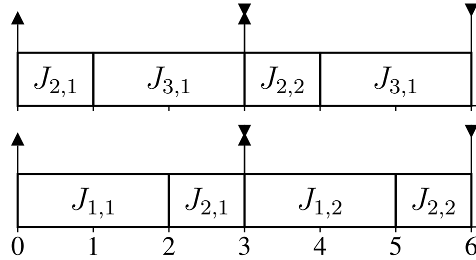


Figure 1.5. A valid schedule produced by a global scheduling approach with migration.

each of which with execution time 2 and deadlines 3 and 6, then the valid schedule shown in Figure 1.5 is a simple example of McNaughton's wrap-around algorithm (MCNAUGHTON, 1959).

Several global scheduling solutions have recently been presented to the optimal multiprocessor real-time scheduling problem, most based on periodic-independent tasks model with implicit deadlines on preemptive, identical and uniform processors. We refer to this model as PPID for short. According to this model, each task is independent of the others, jobs of the same task are released periodically, each job of a task must finish before the release time of its successor job, the system is fully preemptive and migration is allowed between processors.

However, to the best of our knowledge, all optimal algorithms proposed up to date for the PPID model (BARUAH et al., 1996; ZHU et al., 2003; CHO et al., 2006; ANDERSSON; TOVAR, 2006; FUNK, 2010; LEVIN et al., 2010; ZHU et al., 2011) rely on some version of the *proportionate fairness* firstly introduced by (BARUAH et al., 1993). That is, all of them can be considered as approximations of the theoretical fluid model, in which **any** task τ_i executes at the steady rate C_i/T_i in any time interval. They differ essentially by the manner in which the regulation of the executions is realized and by the definition of the time intervals boundaries for calculations of steady rates quanta.

Also, most of these approaches enforce deadline equality by proportionally subdividing workloads and imposing the deadlines of each task on all other tasks (LEVIN et al., 2010). This causes many tasks to execute between every two consecutive system deadlines, leading to excessive context switching and migration overhead.

1.7 CONTRIBUTION

Assumptions

We consider a real-time platform Π comprised of $m \geqslant 2$ identical and uniform processors, each of which executing jobs at a speed of 1 execution quantum per time unit and we focus on global scheduling.

Also, we assume a preemptive and independent job model with free migration, *i.e.,* jobs can be preempted at any time and a preempted job may resume its execution instantaneously on another processor of the platform, with no penalty.

We address a generalization of the PPID model with the goal of finding an optimal on-line and global scheduling algorithm.

Contribution 1

As a first contribution, we introduce the notion of *Dual Scheduling Equivalence* (DSE) in (REGNIER et al., 2011) which is a generalization of (LEVIN et al., 2009). To the best of our knowledge, this work is the first to propose an optimal multiprocessor algorithm based on an efficient use of the DSE approach to ensuring the non-parallel execution of tasks in a multiprocessor real-time system.

As a simple example of DSE, consider the 3-task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$ as introduced in Section 1.6. We show that scheduling this task system on two processors is equivalent to scheduling another 3-task set on one processor. For this purpose, we define the "*dual*" task τ_i^* of a task τ_i as follows: τ_i^* has the same deadline as τ_i and a complementary workload of $3 - 2 = 1$. Hence, the dual τ_i^* of task τ_i represents τ_i's idle time. Hereafter, we refer to τ_i as the *primal* task of the dual task τ_i^*.

In order to produce a valid schedule of the primal set $\mathcal{T}$, we first schedule its dual set $\mathcal{T}^* = \{\tau_1^*, \tau_2^*, \tau_3^*\}$ by EDF on a *virtual* processor, as illustrated in Figure 1.6. Since $\sum_{i=1}^{3} \rho(\tau_i^*) = \sum_{i=1}^{3} 1 - \rho(\tau_i) = 1$, the schedule of $\mathcal{T}^*$ on a single processor by any dynamic priority optimal uniprocessor algorithm is valid.

Then, we apply the following dual scheduling rule to deduce the schedule of $\{\tau_1, \tau_2, \tau_3\}$ by duality. Whenever the dual task executes on the *virtual* processor, its associated original task does not execute on the original system. For instance, when τ_1^* is executing on the *virtual* processor, task τ_1 is not executing on the original system. For this simple 3-task set example, one can easily be convinced that a valid schedule for the primal task set is obtained by blocking τ_i whenever the dual task τ_i^* of τ_i executes in the dual schedule.

In general, we define DUAL as the operation which transforms a task set in the set of its

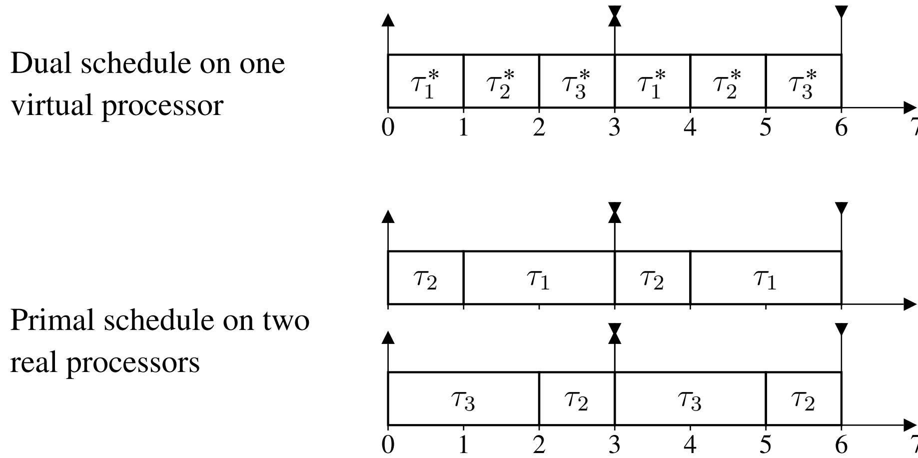


Figure 1.6. Dual Scheduling Equivalence (DSE) of the *primal* task set $\{\tau_1,\ \tau_2,\ \tau_3\}$ on two *real* processors and its *dual* task set $\{\tau_1^*,\ \tau_2^*\ \ \tau_3^*\}$ on one *virtual* processor.

dual tasks, simply called *dual set*. The DUAL operation applied to a single task τ transforms it into the dual task τ^*, whose execution time represents the idle time of τ. More precisely, we assume that $\rho(\tau^*) = 1 - \rho(\tau)$ and that τ and τ^* share the same deadlines. Hence, when primal tasks have rates close to but less than 1, the DUAL operation reduces the accumulated rate of the dual set compared to the accumulated rate of the primal set.

Contribution 2

The simple example of the DSE approach illustrated by Figure 1.6 only requires a single DUAL operation since all tasks in the considered 3-task primal set have relatively high rates compared to one, *i.e.,* more precisely, rates greater than 0.5. However, when the original task set is comprised of many tasks with rates low compared to one, another operation is needed.

For instance, consider a different task set $\mathcal{T} = \{\tau_1{:}(2,3),\ \tau_2{:}(2,3),\ \tau_3{:}(1,6),\ \tau_4{:}(3,6)\}$ to be scheduled on a two-processor system. We can not directly apply the DSE approach to $\mathcal{T}$ since the dual set $\mathcal{T}^*$ would have an accumulated rate of $\sum_{i=1}^{4} \rho(\tau_i^*) = \sum_{i=1}^{4} 1 - \rho(\tau_i) = 2$. Hence, in this case, scheduling the dual set $\mathcal{T}^*$ would be as difficult as scheduling the primal set $\mathcal{T}$.

To overcome this difficulty, we must reduce the number of tasks prior to apply the DUAL operation by aggregating many low rate tasks compared to one into a packet of tasks. In order to schedule such aggregation of tasks, we utilize a server equipped with an ad hoc scheduling policy. For instance, in the above example, a server of rate $\rho(\tau_3)+\rho(\tau_4)$ in charge of scheduling τ_3 and τ_4.

This leads us to the notion of *Partitioned Proportionate Fairness (PP-Fair)*, which is the second contribution of this dissertation. Under PP-Fair scheduling, the original task system is

partitioned into subsets of accumulated utilization no greater than one by a PACK operation. Scheduling of tasks in each packed subset is managed in an isolated manner by a virtual server which globally executes at a steady rate between any two deadlines of its clients, namely those tasks it serves, according to some own scheduling policy. The system is *partitioned proportionate fair* in the sense that each server is guaranteed to execute at a fixed rate which is precisely equal to the sum of the rates of its clients.

However, differently from previous approaches, servers are not required to schedule their clients at a steady rate. In this dissertation, we only consider EDF-servers which schedule their clients by Earliest Deadline First (EDF). As a consequence of the schedule isolation of tasks by servers, a task may essentially cause preemption or migration of another client of the same server it is attended by. The remaining relatively "rare" preemptions/migrations are due to the DSE approach which is used to ensure the non-parallel execution of servers.

Contribution 3

We now enunciate our third and primary contribution as the thesis of this dissertation

> *Optimal on-line algorithm for scheduling periodic and independent real-time tasks with implicit deadlines on a platform of $m \geqslant 2$ preemptive, uniform and identical processors can be built upon Partitioned Proportionate Fairness (PP-Fair) and Dual Scheduling Equivalence (DSE) approaches. An example of such algorithm, called RUN, is exhibited in this dissertation with the following properties.*
>
> - *By performing a sequence of* PACK *and* DUAL *operations, RUN reduces the problem of scheduling a given task set on m processors to an equivalent problem of scheduling one or more different task sets on uniprocessor systems.*
> - *RUN significantly outperforms existing optimal algorithms in terms of preemptions with an upper bound of $O(\log m)$ average preemptions per job on m processors.*
> - *RUN reduces to Partitioned-EDF whenever a proper partitioning is found.*

Figure 1.7 depicts a general view of the RUN scheduling scheme. First, tasks are packed into servers by an off-line PACK operation. Then, servers are scheduled according to the RUN algorithm, which composes DSE and PP-Fair approaches. Finally, no more than m servers chosen to execute are allocated to execute on a processor by the job-to-processor assignment procedure.

It is worth emphasizing here that the core material of this thesis appeared in the 32nd IEEE Real-Time Systems Symposium 2011 (REGNIER et al., 2011), which took place in Vienna, Austria, in December 2011. This paper got the Best Paper Award in this conference. Moreover,

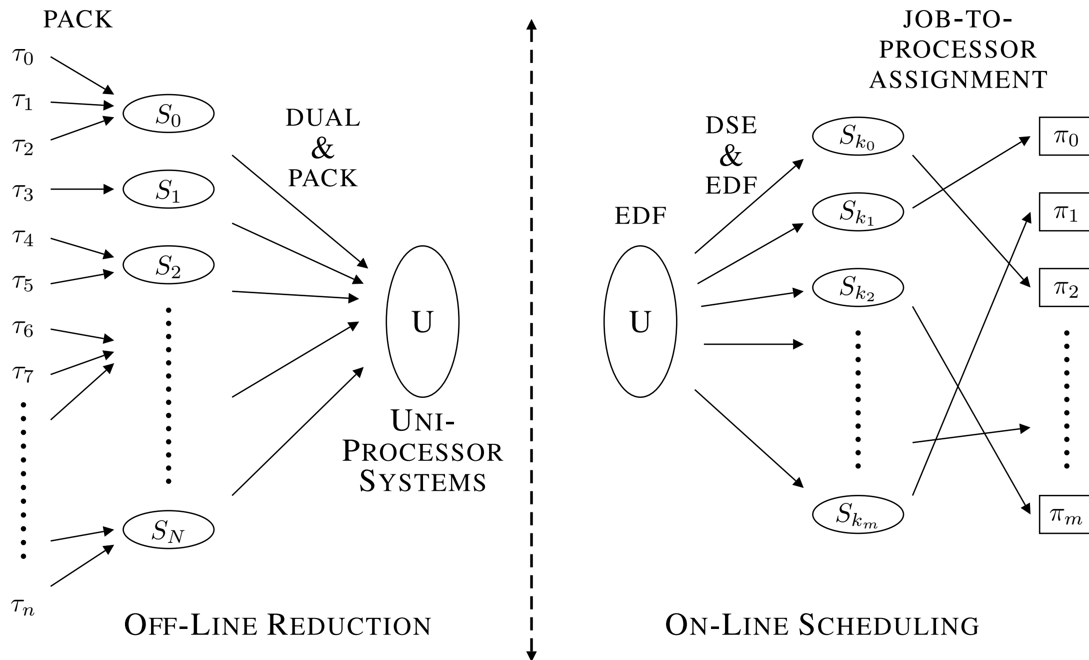


Figure 1.7. RUN: a global scheduling approach using PACK and DUAL operations and job-to-processor assignment.

an extended version of this paper has been invited to be submitted to the Springer Real-Time Systems journal.

1.8 STRUCTURE OF THIS DISSERTATION

Equipped with the theoretical background given in this chapter, we follow with an overview of the state of the art of the multiprocessor real-time scheduling field in Chapter 2, focusing mainly on global and optimal scheduling solutions.

In Chapter 3, we describe the task model adopted in this dissertation and we define the server abstraction, first cornerstone of the RUN algorithm, which is used to aggregate low rate task in order to reduce the total number of tasks to be scheduled.

Chapter 4 depicts the virtual scheduling approach by packing and duality. In particular, the Dual Scheduling Equivalence, second cornerstone of the RUN algorithm, is established. Finally, it is shown how a sequence of reduction by packing and duality transforms a multiprocessor task system into a set of uniprocessor task systems.

Chapter 5 is dedicated to the description of the Reduction to Uniprocessor on-line procedure, the associated on-line scheduling rules and the correctness of the overall RUN algorithm. In particular, the optimality of the RUN algorithm for periodic task set with implicit deadlines is established.

A theoretical upper bound for the average number of preemptions and migrations per job is given in Chapter 6, as well as the results of extended comparisons via simulations of RUN with many other optimal multiprocessor scheduling algorithms.

Chapter 7 concludes this dissertation, introducing some perspectives for future works.

Chapter
2

Most of the complexity of multiprocessor real-time scheduling comes from the impossibility for a task to execute simultaneously on more than one processor. To surround this restriction and achieve optimality for periodic tasks with implicit deadlines, most solutions proposed until now are based on proportionate fairness. However, the idle scheduling idea has shown to be another way toward optimality.

MULTIPROCESSOR SCHEDULING SPECTRUM

2.1 INTRODUCTION

In the realm of uniprocessor, assuming a periodic or sporadic task model with implicit deadline as stated in Definition 1.2.2, the Earliest Deadline First (EDF) and Least Deadline First (LLF) algorithms are optimal scheduling algorithms (LIU; LAYLAND, 1973; DERTOUZOS, 1974; GEORGE et al., 1996). Moreover, a characterization of all possible on-line preemptive scheduling algorithm on one processor is given in (UTHAISOMBUT, 2008). However, it is still an open problem whether a similar characterization can be found for optimal algorithm on platform comprised of two or more processors.

In fact, as previously stated in Section 1.5 of Chapter 1, it is known that no optimal on-line algorithm exists when considering a platform comprised of two or more processors for the sporadic task model (SAHNI, 1979; HONG; LEUNG, 1988; DERTOUZOS; MOK, 1989; FISHER et al., 2010) with constrained deadlines. However, optimality can be achieved for multiprocessor preemptive systems for more restrictive task models, like the LL model for instance.

Structure of the chapter

We begin this Chapter by a brief description of different approaches for multiprocessor scheduling of real-time tasks on identical processors in Section 2.2. Then, we present in Section 2.3, some of the main simple global scheduling algorithms developed to date. In Sections 2.4 and 2.5, we focus our attention on most of the solutions known at the present time

which lead to optimality for periodic real-time tasks with implicit deadlines. Before to briefly conclude this chapter with Section 2.7, we give a glimpse of the idle scheduling approach in Section 2.6, the guiding idea which has led us, ultimately, to the finding of the RUN algorithm.

2.2 MULTIPROCESSOR SCHEDULING SPECTRUM

The spectrum of the real-time multiprocessor scheduling algorithms can be characterized according to the way task migration is controlled. Approaches which prohibit task migration are usually referred to as *partition* scheduling. According to such approaches, tasks are statically allocated to processors off-line, *i.e.,* a single processor has a fixed set of tasks allocated to it during the execution of the system. This allows for the use of uniprocessor scheduling policies, which is a way of avoiding migrations and the consequent complexities of multiprocessor scheduling. However, if it is not possible to partition the considered task set into disjunct subsets of accumulated rate less than or equal to one, this approach cannot be applied. As a matter of fact, it was shown in (KOREN et al., 1998) that, in the worse case, there exist tasks set with accumulated rate greater than but arbitrarily close to 50% of the computing bandwidth that *partitioned* approaches fails to correctly schedule.

On the other side of the spectrum lies approaches which do not control task migration, usually referred as *global* scheduling. According to such approaches, the jobs of tasks are enqueued in a global queue and are scheduled by a scheduling algorithm according to some priority order of the jobs in the queue. This family of solutions usually generates higher implementation overhead and are more complex to be analyzed. However, to the best of our knowledge, global scheduling is the only known way to optimality for recurring task model like the LL task model or the periodic preemptive and independent task model with implicit deadlines (PPID).

Other approaches lie in between global and partition scheduling (ANDERSSON et al., 2008; EASWARAN et al., 2009; KATO et al., 2009; MASSA; LIMA, 2010), which are called semi-partition approaches. The basic idea is to partition some tasks into disjunct subsets. Each subset is allocated to processors off-line, similar to the partition approach. Some tasks are allowed to be allocated to more than one processor and their migration is controlled at run-time. In (BASTONI et al., 2011), it was shown that semi-partitioned approaches are sound. However, they do not always lead to optimal solution for general periodic task sets and one shall be careful on the implementation design to be adopted in order to reduce preemptions and migrations as much as possible.

Since we are interested here in global scheduling algorithms for multiprocessor real-time systems, we briefly describe in the following Section the main global scheduling solutions to date.

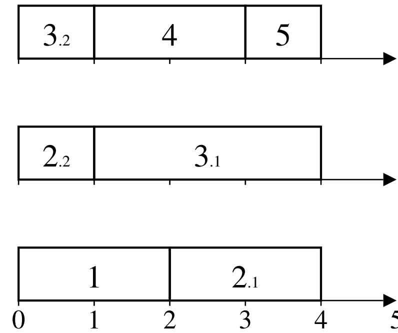


Figure 2.1. McNaughton schedule on 3 processors.

2.3 SIMPLE ALGORITHMS

2.3.1 McNaughton Algorithm

The first optimal solution for scheduling jobs on two or more identical processors is based on the assumption that all jobs share the same deadline. For this restrictive job model, the McNaughton algorithm can be used with a very low implementation cost (MCNAUGHTON, 1959). Since, at some initial stage of our research, we have reinvented this well-known algorithm before discovering that it was more than fifty years old, we give here a proof of its correctness.

Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n preemptive jobs, each of which has an execution time C_i. Moreover, let $\mu = \frac{1}{m}\sum_{i=1}^{n} C_i$ and suppose that $C_i \leqslant \mu$ for all i. The McNaughton algorithm correctly schedules jobs of $\mathcal{J}$ on a system of m identical processors in an arbitrary order, provided that all jobs share the same deadline μ.

Beginning by the first empty processor, jobs are packed from left to right, one after the other. When the first processor is filled, the possible remaining execution time of the last task is packed at the beginning of the next empty processor. This procedure is repeated until all tasks are scheduled. Figure 2.1 shows an example of such schedule for five tasks on three processors, with $C_1 = 2$, $C_2 = 3$, $C_3 = 4$, $C_4 = 2$ and $C_5 = 1$.

Theorem 2.3.1 (McNaughton 1959). *Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n preemptive jobs, each of which has an execution time C_i. If $\mu = \frac{1}{m}\sum_{i=1}^{n} C_i$, $C_i \leqslant \mu$ for all i and if all jobs in $\mathcal{J}$ share the same deadline μ, then $\mathcal{J}$ is feasible on m processors by the McNaughton algorithm in a scheduling window of length μ.*

Proof. Consider an "incorrect" schedule of $\mathcal{J}$ on a single processor in an execution interval of length $m\mu = \sum_{i=1}^{n} C_i$. Without loss of generality, we can suppose that jobs of $\mathcal{J}$ are scheduled according to the increasing order of their indices. Let us divide this execution sequence into

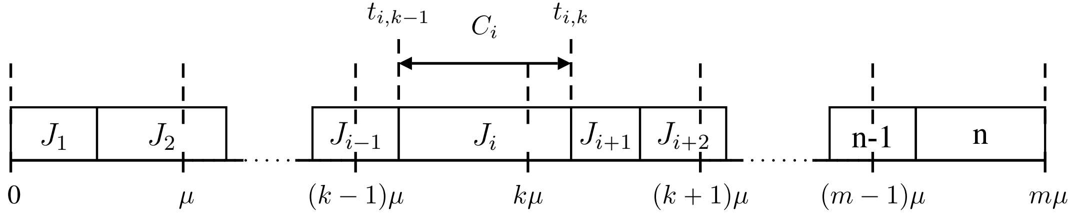


Figure 2.2. Schedule of Γ on a single processor in a window interval of length $m\mu$.

m execution intervals of length μ, called μ-intervals and denoted $I_k = [k\mu, (k+1)\mu)$, for $k = 0, 1, \ldots m-1$. Now, assume that each μ-interval I_k is assigned to a dedicated processor. In such a case, a job J scheduled during μ-interval I_k is guaranteed to complete during I_k, *i.e.,* before its deadline $J.d = \mu$. Thus, if no two μ-intervals contains conflicting execution, the theorem is proved.

First, since $C_i \leqslant \mu$, two non-consecutive μ-intervals can not contain execution intervals of the same job. Next, consider two consecutive μ-intervals I_{k-1} and I_k with $1 \leqslant k \leqslant m-1$ in the schedule of $\mathcal{J}$ on a single processor as shown in Figure 2.2. We suppose that job J_i executes in both I_{k-1} and I_k on processors P_{k-1} and P_k, respectively. Since processors are filled until completion, it must be that the execution intervals of J_i on P_{k-1} and P_k are of the form $J_{i,k-1} = [t_{i,k-1}, k\mu]$ and $J_{i,k} = [k\mu, t_{i,k}]$, respectively.

By construction, $C_i = t_{i,k} - t_{i,k-1}$. Hence,

$$\begin{aligned} t_{i,k} - k\mu &= t_{i,k-1} + C_i - k\mu \\ &= t_{i,k-1} + C_i - ((k-1)\mu + \mu) \\ &= t_{i,k-1} - (k-1)\mu + C_i - \mu \end{aligned}$$

and since $C_i - \mu \leqslant 0$, we deduce that $t_{i,k} - k\mu \leqslant t_{i,k-1} - (k-1)\mu$.

Now, consider that J_i is scheduled on two distinct processors P_{k-1} and P_k. By this transformation, we deduce that the start times $s_{i,k-1}$, $s_{i,k}$ and finish times $f_{i,k-1}$, f_i of J_i on P_{k-1} and P_k, respectively, satisfy $s_{i,k-1} = t_{i,k-1} - (k-1)\mu$, $f_{i,k-1} = \mu$, $s_{i,k} = 0$ and $f_{i,k} = t_{i,k} - k\mu$.

This implies that $f_{i,k} \leqslant s_{i,k-1}$ and, thus, the two execution intervals of J_i on P_{k-1} and P_k can not be concurrent in the produced by the McNaughton algorithm of J_i on two processors. □

It is worth noting that this theorem furnishes an optimal algorithm to schedule a set of tasks with identical periods (EASWARAN et al., 2009). Moreover, if this period equals $\frac{1}{m}\sum_{i=1}^{n} C_i$, then the m processors are fully utilized.

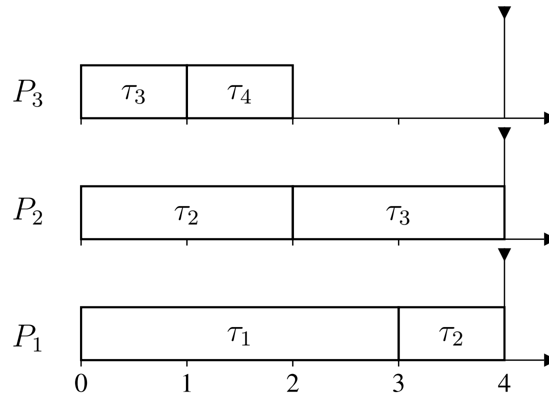


Figure 2.3. Example of non-working schedule produced by the McNaughton algorithm

Note also that the McNaughton algorithm is not work-conserving. Consider for instance the 4-task set J_1:(0, 3, 4), J_2:(0, 3, 4), J_3:(0, 3, 4) and J_4:(0, 1, 4). The McNaughton schedule generated at time 0 is shown in Figure 2.3. Observe that at time 2, processor P_3 remains idle despite the fact that there are three tasks ready to execute. In the context of energy saving and power aware, the idle time produced on P_3 could be used to decrease the speed of the processor. Indeed, the schedule of Figure 2.3 would remain valid if P_3 were to execute tasks twice slower than its normal speed.

However, regarding identical processors and assuming a general task model where periods are arbitrary, the McNaughton algorithm can not be applied as it is. Still, it can be usefully utilized by transforming a general task system in an adequate manner, as will be seen in Section 2.4.

2.3.2 Global EDF, LLF

Before to step into the detailed description of known optimal multiprocessor scheduling algorithms, we recall briefly the rules applied by EDF, LLF and EDZL on a multiprocessor platform, since these three algorithms are commonly referred in the realm of global multiprocessor scheduling of real-time tasks.

On a multiprocessor system, EDF and LLF use a single global queue, denoted $\mathbb{Q}(t)$ in which all ready jobs are stored at time t. To distinguish the fact that the system is now comprised of two or more processors, we denote gEDF and gLLF for global EDF and global LLF, respectively. Like in uniprocessor system, when a processor becomes available, that is, when the execution of a job finishes, both algorithms pick up a job in $\mathbb{Q}(t)$ and schedule it on the processor available. The two algorithms differ in the manner that they choose the job to execute. The former, gEDF, picks up the "most urgent" job in $\mathbb{Q}(t)$, *i.e.,* the one whose deadline is the

earliest. The latter, gLLF, picks up in $\mathbb{Q}(t)$ the job which is more likely to miss its deadline *i.e.,* the one whose laxity, as defined in Section 1.4.1, is the smallest. In both cases, ties are broken arbitrarily.

It is well known that gEDF and gLLF fail to schedule some simple task sets. For example, they do not produce a valid schedule for the simple task set example $\mathcal{T} = \{\tau_1\!:\!(2,3)$, $\tau_2\!:\!(2,3)$, $\tau_3\!:\!(4,6)\}$ given in Section 1.6 on a two-processor system. Indeed, both schedule τ_1 and τ_2 during time interval $[0,2)$, causing a deadline miss at time 6 as shown in Figure 1.3.

As a matter of fact, the new restriction specific to multiprocessor systems, which states that a task can not execute at the same time on two processors, decreases dramatically the number of task sets schedulable by gEDF. A lower bound on accumulated rate for gEDF schedulability can be illustrated with a simple example. Consider an $(n+1)$-task set comprised of n identical tasks with execution time 2ε and period 1, and one different task with execution time 1 and period $1+\varepsilon$; *i.e.,* $\mathcal{T} = \{\tau_1\!:\!(1, 1+\varepsilon), \tau_2\!:\!(2\varepsilon, 1), \ \ldots, \tau_{n+1}\!:\!(2\varepsilon, 1)\}$ with ε positive and very small compared to 1. This task set is not schedulable by gEDF on n processors. Indeed, at time 0, gEDF schedules the n identical tasks on the n processors during the time interval $[0, 2\varepsilon)$. Then, at time 2ε, gEDF schedules τ_1 which misses its deadline at time $1+\varepsilon$. Also, the accumulated rate of this task set tends to 1 when ε tends to 0. This shows that gEDF may fail to schedule a task set which requires more than 1 out of n processors.

Although gLLF is not optimal on two or more processors for the periodic task model with implicit deadlines, it has been shown that gLLF is suboptimal (DERTOUZOS; MOK, 1989) on any number of processors. Moreover, since gLLF generates a high number of preemptions for some task sets, enhanced schemes are needed to make it suitable (HILDEBRANDT et al., 1999). However, such schemes have not lead to known optimal algorithms.

2.3.3 EDZL

Earliest Deadline Zero Laxity (EDZL) is a simple but efficient approach that improves dramatically the behavior of the EDF algorithm on a multiprocessor platform. To do so, the EDZL algorithm adds to the EDF rules a single rule, called Zero Laxity (ZL) rule, which states that any job whose laxity becomes equal to zero has its priority promoted to the highest priority of the system. The simple idea behind this rule is that a job whose laxity reaches zero must be imperatively executed, otherwise, it will miss its deadline. Consider the previous example where $\mathcal{T} = \{\tau_1\!:\!(1, 1+\varepsilon), \tau_2\!:\!(2\varepsilon, 1), \ \ldots, \tau_{n+1}\!:\!(2\varepsilon, 1)\}$. At time ε, the laxity of the first job of τ_1 becomes zero and, by the ZL rule, it preempts one of the other jobs and begins to execute. When one of the jobs not preempted at time ε completes its execution at time 2ε, the preempted job is scheduled on the idled processor and completes by time 3ε.

Recall from Section 1.6 that EDZL also fails to correctly schedule some simple task set.

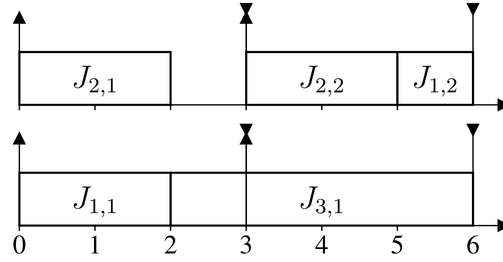


Figure 2.4. EDZL schedule on two processors - $J_{1,2}$ miss its deadline at time 6.

Figure 2.4 shows an example of such a failure, using the simple example $\mathcal{T} = \{\tau_1\!:\!(2,3), \tau_2\!:\!(2,3), \tau_3\!:\!(4,6)\}$ of Section 1.6. Indeed, as at time 0 no jobs have zero laxity, both $J_{1,1}$ and $J_{2,1}$ are scheduled on P_1 and P_2 during time interval $[0,2)$, respectively. Then, at time 2, $J_{3,1}$ reaches zero laxity. It is scheduled continuously until time 6. However, an idle slot occur in time interval $[2,3)$ since both $J_{1,1}$ and $J_{2,1}$ have finished to execute and $J_{3,1}$ can not execute in parallel with itself. This shows that EDZL also fails to avoid the occurrence of an idle slot on one processor, resulting in a deadline miss a time 6.

Nevertheless, it was shown in (PARK et al., 2005) that EDZL strictly dominates EDF in the sense that it correctly schedules any task set schedulable by EDF and there exist task sets feasible by EDZL that EDF does not schedule correctly. Also, it is shown in (CHO et al., 2002) that EDF and EDZL are not suboptimal (see Definition 1.5.2) on two or more processors.

In (PIAO et al., 2006), it has been shown that any task set with total utilization less than $(m+1)/2$ is schedulable by EDZL. We give here an example of periodic task set not schedulable by EDZL on two processors and with total utilization arbitrarily close to $3/2$. Hence, we can deduce that $3/2$ is a tight bound for the accumulated rate of an EDZL schedulable task set on two processors.

Let $\alpha < 1$ and $\beta < 1$ be two positive real numbers and $k > 2$ be an integer. We define the task set $\mathcal{T}$ shown in Table 2.1 whose accumulated rate is given by:

$$\begin{aligned}\rho(\mathcal{T}) &= \frac{1+\alpha}{2} + \frac{1+\alpha}{2} + \frac{k+\beta}{2k} \\ &= \frac{3}{2} + \alpha + \frac{\beta}{2k}\end{aligned}$$

Now, we show that, for some value of α and β, $\mathcal{T}$ can not be scheduled by EDZL on 2 processors without missing a deadline.

Consider time $t_1 = 2(k-1)$. By that time, τ_1 and τ_2 must have executed exactly $k-1$ times. Thus, τ_3 can not have executed for a time x greater than

$$x \leqslant 2(k-1) - (k-1)(1+\alpha)$$

Table 2.1. Task set $\mathcal{T}$ (with $D_i = P_i$).

Task	τ_1	τ_2	τ_3
C_i	$1+\alpha$	$1+\alpha$	$k+\beta$
P_i	2	2	$2k$

However, since EDZL is a work-conserving algorithm, τ_3 must have executed whenever neither τ_1 and τ_2 was executing. Also, since $e(\tau_3, 0) = k + \beta \geqslant (k-1)(1-\alpha)$ for $\beta > 0$, τ_3 can not have finished to execute at time t_1. Hence, it must be that $x = (k-1)(1-\alpha)$ and the remaining execution time $e(\tau_3, t_1)$ of τ_3 at time t_1 satisfies

$$\begin{aligned} e(\tau_3, t_1) &= k + \beta - (k-1)(1-\alpha) \\ &= 1 + \beta + (k-1)\alpha \end{aligned}$$

As a consequence, the laxity $l(\tau_3, t_1) = 2 - e(\tau_3, t_1)$ of τ_3 at t_1 satisfies:

$$l(\tau_3, t_1) = 2 - (1 + \beta + (k-1)\alpha) = 1 - \beta - (k-1)\alpha$$

Now, for a given integer k, we can choose α and β such that $(k-1)\alpha < 1$ and

$$\beta = 1 - (k-1)\alpha > 0 \tag{2.1}$$

For such values of α and β, $l(\tau_3, t) > 0$ for all time t before t_1. Also, $l(\tau_3, t_1) = 0$ and $e(\tau_3, t_1) = 2$.

The schedule of this task set by EDZL is shown by Figure 2.5. As can be seen, at time t_1, the three tasks τ_1, τ_2 and τ_3 have deadline $2k$. Also, the total execution time demand of this three tasks at time t_1 equals $2 + 2(1+\alpha)$, which exceeds the 4 computation units provided by the 2 processors until time $2k$. Thus, a deadline miss must occur at time $2k$.

Now, observe that the accumulated rate of $\mathcal{T}$ is given by:

$$\rho(\mathcal{T}) = \frac{1+\alpha}{2} + \frac{1+\alpha}{2} + \frac{k+\beta}{2k}$$

And thus, by Equation 2.1

$$\begin{aligned} \rho(\mathcal{T}) &= \frac{3}{2} + \alpha + \frac{1-(k-1)\alpha}{2k} \\ &= \frac{3}{2} + \frac{\alpha}{2} + \frac{1+\alpha}{2k} \end{aligned}$$

Thus, choosing k big enough and α satisfying $(k-1)\alpha < 1$, $\rho(\mathcal{T})$ can be made arbitrarily

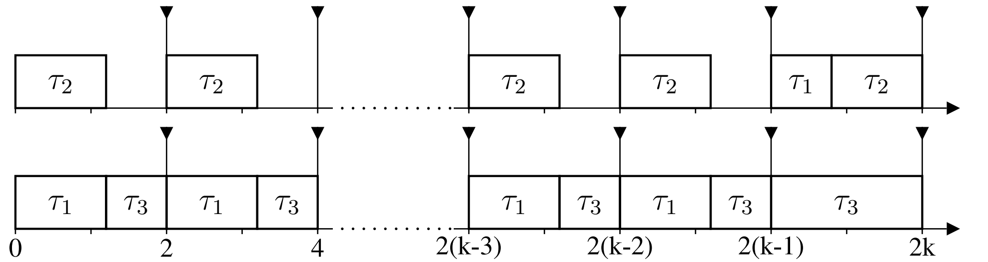


Figure 2.5. EDZL schedule on two processors of $\mathcal{T}$ as defined in Table 2.1. In this schedule, τ_1 misses it deadline at time $2k$.

close to but greater than $\frac{3}{2}$. This shows that $\frac{3}{2}$ is a tight bound for the accumulated rate of a feasible task set by EDZL on 2 processors.

2.4 OPTIMAL MULTIPROCESSOR SCHEDULING

As seen in Section 1.6, a scheduling algorithm is optimal for the periodic and implicit deadline task model on an m-identical multiprocessor system if it produces a valid schedule for any task set $\mathcal{T}$ whenever

$$\sum_{\tau_i \in \mathcal{T}} \frac{C_i}{T_i} \leqslant m$$

For instance, while the Earliest Deadline First (EDF) scheduling algorithm is optimal on a uniprocessor system (LIU; LAYLAND, 1973), we have seen in Section 2.3 that gEDF fails when applied to a multiprocessor system (see Figure 1.3).

Until recently, all optimal scheduling approaches are approximations of the theoretical fluid model, also called proportionate fairness approach (BARUAH et al., 1993), in which all tasks execute at the steady rate $\frac{C_i}{T_i}$ in any time interval. They differ essentially by the manner in which the regulation of the executions is realized.

We present here four of the main algorithms based on the theoretical fluid model: the proportionate fairness approach (Pfair) (BARUAH et al., 1996), the EKG approach (ANDERSSON; TOVAR, 2006), the time and local execution time plane (T-L plane) (CHO et al., 2006) and the deadline partitioning approach (DP-fair) (LEVIN et al., 2010).

2.4.1 Proportionate Fairness

Considering a periodic task model with implicit deadlines, optimality can be achieved by approaches that approximate the theoretical fluid model, according to which all tasks execute in any time interval at the steady rate proportional to their utilization.

According to the proportionate fair (Pfair) approach, as proposed in (BARUAH et al., 1993), tasks are broken into a series of quantum-length Q subtasks, which are fairly distributed on all processor of the system. Given a quantum Q, a Pfair schedule must satisfy the following property. In any time interval of length d, the accumulated execution time c (number of quanta) of a task τ with rate w satisfies

$$c \leqslant wd \leqslant c + Q$$

Since wd would be the accumulated execution time of τ according to the theoretical fluid model, we see that the Pfair approach allows the execution time of a task to be apart from the fluid model by at most one quantum at any time. Hence, each task is guaranteed to execute at an approximately steady rate.

The Pfair approach, recently adapted to sporadic job sets (HOLMAN; ANDERSON, 2005), is elegant and theoretically achieves optimality. However, for some task set, the quantum Q can be arbitrarily small in order to guarantee that all tasks meet their deadlines. As a consequence, the number of preemptions and / or migrations can become arbitrarily large, turning this theoretical approach practically useless for some task sets.

Based upon proportionate fairness, many algorithms, like EPDF (ANDERSON; SRINIVASAN., 2000; ANDERSON; SRINIVASAN., 2004), PD (BARUAH et al., 1995) and PD^2 (ANDERSON; SRINIVASAN., 2004), have been proposed to ensure optimality while making the implementation more suitable for practical systems than the original Pfair algorithm.

2.4.2 Pfair derivatives

In a recent work, (LEVIN et al., 2010) have formalized a minimal restrictive set of scheduling rules, called DP-fair, standing for deadline-partition fair, showing that any algorithm built upon DP-fair rules is optimal for periodic and implicit-deadline task sets. More specifically, it is shown in (LEVIN et al., 2010) that all optimal approaches developed until 2010 (BARUAH et al., 1996; ZHU et al., 2003; CHO et al., 2006; ANDERSSON; TOVAR, 2006; LEVIN et al., 2010; ZHU et al., 2011) share the following characteristics. First, they rely on some version of proportionate fairness, and second, like McNaughton's algorithm, they all rely upon the simplicity of scheduling when deadlines are equal.

T-L plane

Consider the case of the Largest Local Remaining Execution First (LLREF) scheduling algorithm, which is based on the time and local execution-time domain plane (T-L Plane) approach proposed in (CHO et al., 2006; FUNAOKA et al., 2008). LLREF also aims to execute all tasks at a steady rate. However, it differs from the Pfair algorithm by the scheduling instants

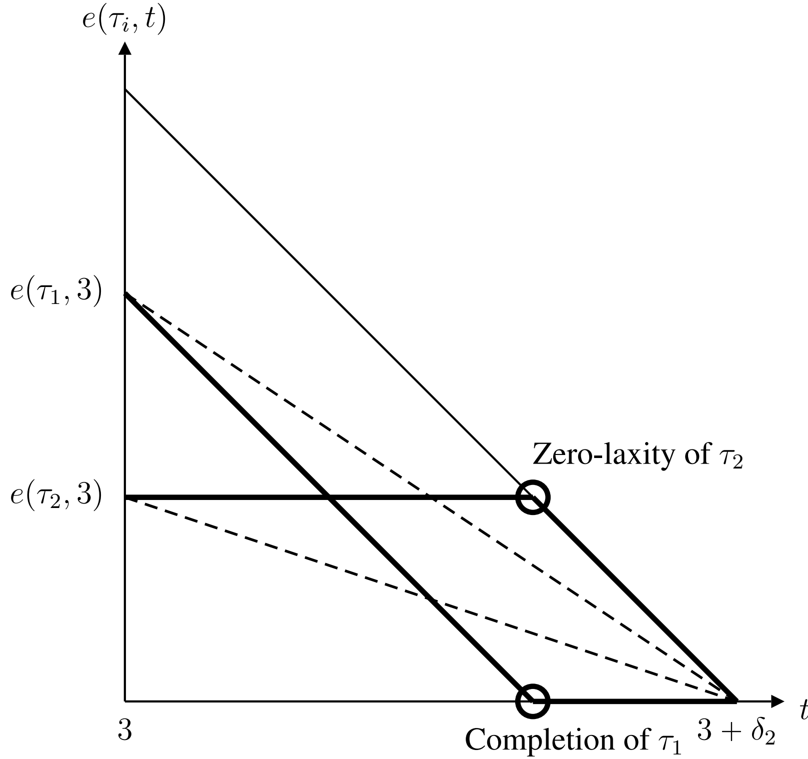


Figure 2.6. Node N_2 of the TL-Plane approach for two tasks $\tau_1:(2,3)$, $\tau_2:(\frac{5}{3},5)$.

that it uses. Instead of breaking all task into fixed size quantum subtasks, this approach defines scheduling windows, called nodes, between any two primary scheduling instants, defined as the task release instants and deadlines. During a node (or slice) N_k of duration δ_k, each active task of the periodic taskset executes for $\delta_k C_i/T_i$. Whenever the laxity of a task reaches zero or a task finishes its local execution time, a secondary scheduling instant is created. In the first case, the "local" zero laxity task is scheduled to execute immediately, until its local deadline, while in the second case, another task is scheduled to execute in place of that one which has "locally" completed.

An illustrative example is given in Figure 2.6, considering two tasks $\tau_1:(2,3)$, $\tau_2:(\frac{5}{3},5)$ to be scheduled on a single processor. Figure 2.6 depicts node N_2, which begins at time 3 and has length $\delta_2 = 2$. At time 3, tasks τ_1 and τ_2 require $e(\tau_1,3) = \rho(\tau_1)\delta_2 = 4/3$ and $e(\tau_2,3) = \rho(\tau_2)\delta_2 = 2/3$ of execution time, respectively. Continuous diagonal lines and horizontal lines represent time intervals during which a task executes and does not execute, respectively. Dashed diagonal lines represent the theoretical fluid model execution.

Since τ_1 has largest local remaining execution time than τ_2, it is scheduled first. Observe that a local completion event and local zero-laxity event occurs at time 6 for τ_1 and τ_2, respectively.

To provide local feasibility for a general task system scheduled on m identical processors, at every scheduling instant, m of the largest local remaining execution time tasks are selected

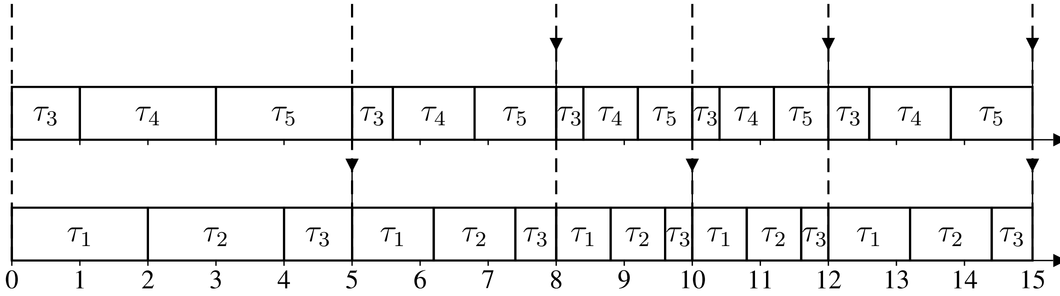


Figure 2.7. DP-wrap schedule of task set $\mathcal{T} = \{\tau_1:(2,5),\ \tau_2:(3.2,8),\ \tau_3:(4,10),\ \tau_4:(4.8,12), \tau_5:(6,15)\}$

first, hence the name LLREF of the scheduling policy.

In (FUNK, 2010), the T-L plane approach is extended to the sporadic task model with unconstrained deadlines.

DP-wrap

As an example of a simple optimal algorithm built upon DP-fair rules, the DP-wrap scheduling algorithm is proposed in (LEVIN et al., 2010). Similarly to T-L plane, time is divided into time slices, *i.e.,* the nodes in the T-L plane approach, of length equal to the distance between two distinct and consecutive deadlines in the system. During a time slice k, each task τ_i executes for its *local execution time* which is proportional to the rate of τ_i and the duration δ_k of slice k. Precisely, if $C_{i,k}$ is the local execution time of task τ_i during slice k, then

$$C_{i,k} = \rho(\tau_i)\delta_k$$

Doing so, the original problem is transformed into an easier problem in each slice since all pieces of jobs in a slice share the same (slice) deadline. Differently from the T-L plane approach, this easier problem is then solved using the McNaughton algorithm (MCNAUGHTON, 1959), previously described in Section 2.3. Since the McNaughton algorithm is optimal when deadlines of tasks are equal, the DP-wrap implementation of the DP-fair rules is optimal.

We illustrate the DP-wrap algorithm with the simple task set $\mathcal{T} = \{\tau_1:(2,5),\ \tau_2:(3.2,8),\ \tau_3:(4,10),\ \tau_4:(4.8,12),\ \tau_5:(6,15)\}$. The corresponding DP-wrap schedule is shown in Figure 2.7.

EKG

The EKG approach, as a short-hand notation for **E**DF with task splitting and **k** processors in a **g**roup, has been proposed in (ANDERSSON; TOVAR, 2006), a couple of years before

DP-wrap. However, it is easier to explain EKG using DP-wrap as cornerstone. Indeed, EKG is a particular case of DP-wrap in which tasks are statically assigned to processors using a bin-packing scheme based on the task rate. Two cases may take place during the process of filling a bin/processor P depending on the rate of task τ. If the accumulated rate of the task set assigned to P, denoted $\rho(P)$ for the sake of simplicity, satisfies

$$\rho(\tau) \leqslant 1 - \rho(P)$$

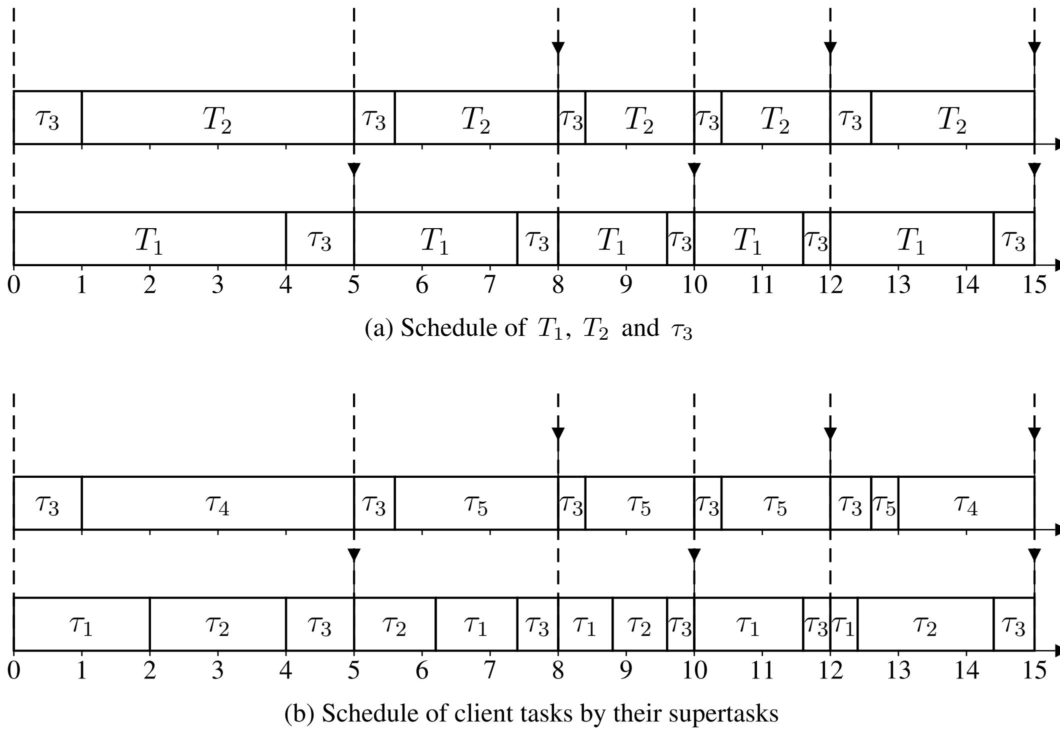
Then, task τ is completely assigned to P. Otherwise, task τ is split into two subtasks. The first of these subtasks, with rate $1 - \rho(P)$, is assigned to P in order to fill that processor. The second, with rate $\rho(\tau) - (1 - \rho(P))$ is assigned to the next empty processor. It is clear that, independently of the bin-packing scheme used, there are at most $m - 1$ split tasks at the end of the task-to-processor assignment step.

Each split task is a migratory task, which may migrate during system execution. Task completely assigned to processor do not migrate. In the EKG scheme, fixed tasks assigned to a given processor P are aggregated into a *supertask* T. Note that the notion of supertask is in line with our definition of server, as introduced in Section 1.7 and precisely defined in Chapter 3. Hence, the rate of a supertask is precisely equal to the accumulated rate of the set of its clients, namely those tasks it aggregates. Also, in each time slice of length δ, a supertask T of rate $\rho(T)$ is guaranteed to execute for exactly $\rho(T)\delta$. However, differently from DP-wrap, the clients of an EKG supertask are scheduled by the EDF scheduling policy. Thus, even if supertasks and migratory tasks follow a DPfair schedule, the proportionate fairness between clients of a single supertask does not need to be guaranteed in each time slice. Still, each client task is guaranteed to meet its deadlines, provided that its supertask meets them. Since EKG uses EDF to schedule non-migratory tasks, it generates fewer preemptions than DP-wrap. However, the number of migration under EKG and DP-Wrap are the same.

We illustrate the EKG algorithm with the same simple task set than previously $\mathcal{T} = \{\tau_1{:}(2,5),\ \tau_2{:}(3.2,8),\ \tau_3{:}(4,10),\ \tau_4{:}(4.8,12),\ \tau_5{:}(6,15)\}$. We assume here that ($\tau_1$, τ_2) and (τ_4, τ_5) are grouped into the same supertask T_1 and T_2, respectively. Hence, the only migratory task is τ_3. Figure 2.8a shows the schedule of T_1, T_2 and τ_3 and Figure 2.8b shows how each supertask schedules its clients.

Discussion

As can be seen, all Pfair-based approaches enforce deadline equality by proportionally subdividing workloads and imposing the deadlines of each task on all other tasks (LEVIN et al., 2010). As a consequence, this may cause many tasks to execute between every two consecutive system deadlines, leading to excessive context switching and migration overhead.



(a) Schedule of T_1, T_2 and τ_3

(b) Schedule of client tasks by their supertasks

Figure 2.8. EKG schedule of task set $\mathcal{T} = \{\tau_1{:}(2,5),\, \tau_2{:}(3.2,8),\, \tau_3{:}(4,10),\, \tau_4{:}(4.8,12), \tau_5{:}(6,15)\}$

RUN is not based upon proportionate fairness but upon partitioned proportionate fairness, as described in Section 1.7. This makes RUN a more general approach capable of generating fewer preemptions than those found by other Pfair-based approaches.

2.5 AN UNFAIR APPROACH

In a recent work (NELISSEN et al., 2011), a new algorithm called U-EDF, which stands for **U**nfair scheduling algorithm based on **EDF**, has been proposed. U-EDF uses a DP-fair algorithm, namely DP-wrap, but relaxes the proportionate fairness assumption in order to decrease the needs for preemptions and migrations.

In a nutshell, the U-Fair algorithm makes reservation for future executions of jobs on all processors using the DP-wrap algorithm. Then, at each scheduling boundary, *i.e.,* at each release instant of a job, U-EDF schedules, for the next time slice, the parts of each job assigned to execute on a processor using an EDF-like algorithm. As a consequence, differently from DP-wrap, a job may execute more than its local execution time during a time slice.

In order to guarantee the non-parallel execution of the different parts of a job, U-EDF uses a variant of EDF on each processor in order to guarantee that two parts of the same job are not scheduled simultaneously. This is achieved by two means. First, the scheduling algorithm always considers the processors in the same off-line defined order. Second, when the first part

of a job is scheduled by EDF on a processor, all the eventual other parts of the same job are removed from the ready queue of all other processors.

Also, the on-line calculation of the reservation for future execution of a job on a processor is carried out at each scheduling event, taking into account the previous reservation already contracted for a job on previous processors. We do not expose a complete schedule of a simple task set, since the U-EDF algorithm requires some non trivial calculation not presented here. However, we invite the interested reader to refer to (NELISSEN et al., 2011) for the complete picture of the U-EDF algorithm.

Still not proven to be optimal, U-EDF has succeeded to correctly schedule more than thousand randomly generated task sets, as described in (NELISSEN et al., 2011). In all those experiments, U-EDF reduces significantly the average number of preemptions and migrations per job when compared to DP-Wrap and EKG.

2.6 IDLE SCHEDULING

During the first two years of this PhD research, we have been actively working on the idea of scheduling both execution and idle times in order to improve the efficiency for generating the schedule. Before to lead us to RUN, the optimal algorithm presented in this dissertation, this idle scheduling idea has led us to develop the idle serialization approach that we briefly present here. Interested readers can refer to Appendix A for a more complete description of this not yet fructuous approach.

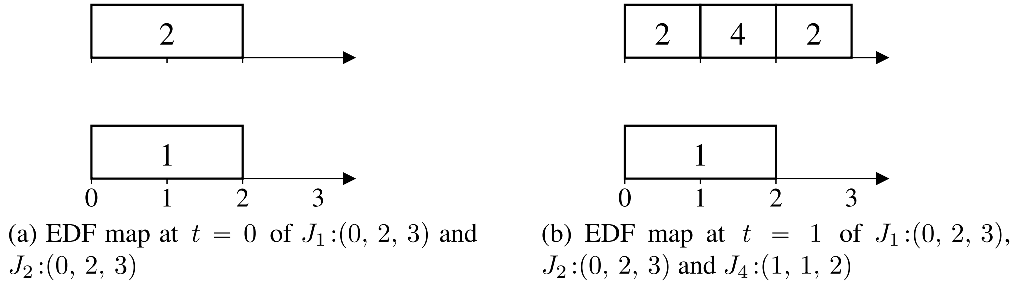
We call *frame*, denoted $[s, f)_k$, the execution time available on a processor P_k during time interval $[s, f)$. An idle frame is one during which no job executes. We denote $[s, f)_{k,i}$ the frame where job J_i executes on P_k.

We say that two frames $F_j = [s, f)_j$ and $F_k = [s', f')_k$ on two distinct processors P_j and P_k are serialized if they do not overlap in time, *i.e.,* $[s, f) \cap [s', f') = \{\}$. Also, serializable frames are those that can be serialized in the same processor.

Upon arrival of a job J at time t, a set of serializable frames, called *mapping* of J, is reserved on the processors for the future executions of J. It is assumed that such reservation do not let idle interstice on the processors. In other words, the reservation of frame is done in a work-conserving manner.

For example, consider the 3-task set $\mathcal{T} = \{\tau_1\!:\!(2,3),\, \tau_2\!:\!(2,3),\, \tau_3\!:\!(4,6)\}$. The mappings assigned to $J_1\!:\!(0,\ 2,\ 3)$ and $J_2\!:\!(0,\ 2,\ 3)$ by EDF are shown in Figure 2.9a.

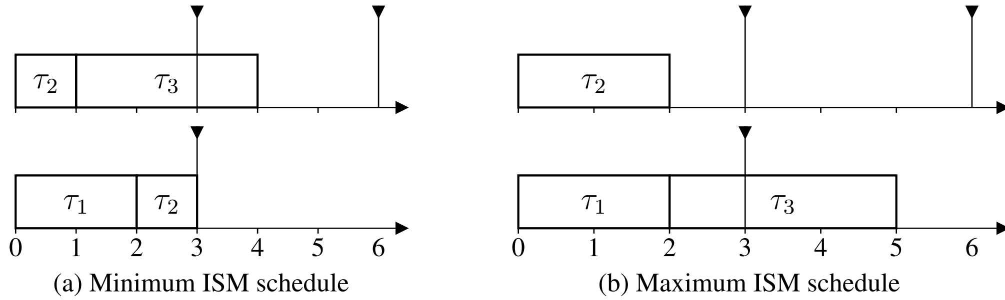
At time t, the set of all mappings already defined on P is called a map and is denoted $\mathbb{M}(t)$. Reserved frames in a map $\mathbb{M}(t)$ can either be free or locked. A free reserved frame can be cancelled at some future scheduling instant, while locked frames are immutable.



(a) EDF map at $t = 0$ of J_1:(0, 2, 3) and J_2:(0, 2, 3)

(b) EDF map at $t = 1$ of J_1:(0, 2, 3), J_2:(0, 2, 3) and J_4:(1, 1, 2)

Figure 2.9. EDF map examples.



(a) Minimum ISM schedule

(b) Maximum ISM schedule

Figure 2.10. The minimum ISM schedule turns J_4:(3, 2, 6) and J_5:(3, 2, 6) feasible and J_4:(2, 4, 6) unfeasible, while the maximum ISM schedule turns J_4:(3, 2, 6) and J_5:(3, 2, 6) unfeasible and J_4:(2, 4, 6) feasible.

For instance, suppose that at time $t = 1$ a job J_4:(1, 1, 2) is added to our 3-task set example. Assuming that $[0, 2)_{2,1}$ is a free frame at $t = 0$, then, the resulting map $\mathbb{M}(1)$ assigned by the EDF scheduling policy, and shown by Figure 2.9b, would be $\mathbb{M}(1) = \{[0, 2)_{1,1}$, $[0, 1)_{2,2}$, $[1, 2)_{2,4}$, $[2, 3)_{2,2}\}$ and $\mathbb{Q}(1) = \{J_3$:(0, 4, 6)$\}$.

Now, we give a glimpse of the idle serialization approach utilized to devise the **I**dle **S**erialization **Ba**sed (ISBa) scheduling algorithm through a simple example.

Consider the set of jobs J_1:(0, 2, 3), J_2:(0, 2, 3) and J_3:(0, 3, 6), ordered by non decreasing deadlines (EDF). At time $t = 0$, once J_1 and J_2 have been mapped to $[0, 2)_{1,1}$ and $[0, 2)_{2,2}$, respectively, there are two possible mappings for J_3. First, the scenario of Figure 2.10(a), called minimum idle serialization map (ISM), can be chosen. Doing so, the schedule of two jobs, yet to be released, J_4:(3, 2, 6) and J_5:(3, 2, 6) becomes feasible.

Second, the schedule of a job J_4:(2, 4, 6) would only be feasible if the scenario of Figure 2.10(b), called maximum ISM, were chosen at time 0. Such impossibility to make the right choice for all scenarios is in strong agreement with the result of Dertouzos (DERTOUZOS; MOK, 1989) which states that no optimal multiprocessor scheduling algorithm exist in the general sporadic job model.

Hence, when a ready job J_i is considered for mapping at time t, ISBa needs to choose between the maximum or minimum idle serialization mapping. However each of these two

choices has consequences. Choosing the maximum ISM scenario may turn feasible a ready job, taking advantage of the full length of the longest idle time. On the other hand, choosing the minimum ISM may turn feasible two jobs with low laxity, yet to be released.

As a consequence, we adopted the following scheduling rules for the ISBa algorithm. While no ready jobs can execute, thanks to the idle serialization, ISBa chooses the minimum idle serialization schedule. Otherwise, ISBa opts for the maximum idle serialization schedule. In other words, ISBa only chooses a maximum ISM schedule when this choice does not cause the idling of a processor. Otherwise, ISBa chooses the minimum ISM schedule.

2.6.1 Discussion

We have successfully implemented the ISBa algorithm. However, after more than a year of intensive work, we were disappointed because ISBa was only capable to schedule about the same number of fully-utilization task sets as EDZL when using random task set generated by the open-source random task generator developed by Emberson (EMBERSON et al., 2010; EMBERSON et al., 2011). Since the ISBa implementation was much more complicated than that of EDZL, we concluded that, in general, the idle serialization approach was not worth it from an implementation viewpoint.

Nevertheless, the idea of scheduling the idle time of a task instead of its execution time remained as a promising cornerstone from this unsuccessful serialization approach. A couple of months latter, this idea gave birth to our reduction to uniprocessor algorithm, which is partly based upon duality, *i.e.,* idle scheduling.

Other related work may be found on the topics of duality. For instance, a recent and not yet published work (LEVIN et al., 2009) establishes that if a set $\mathcal{T}$ of $m+1$ tasks have their total utilization exactly equal to m, then a dual-based algorithm produces a feasible schedule of these tasks on m processors. This result can be seen as a special case of the approach being proposed here.

2.7 CONCLUSION

Up to date, optimality in multiprocessor scheduling has mainly been achieved through different variations of the proportionate fairness (Pfair) idea as proposed in (BARUAH et al., 1993). It is only recently that a new approach U-EDF, based on a DP-fair algorithm, but relaxing the fairness constraint has been proposed in (NELISSEN et al., 2011). Although not proved to achieve optimality for periodic task systems with implicit deadlines, the relaxation of the fairness constraint allows U-EDF for achieving much lower preemption overhead than previous Pfair-based algorithms.

As will be exposed in the remaining chapters of this dissertation, the RUN algorithm, first published in (REGNIER et al., 2011), also achieves a low preemption overhead by relaxing the fairness constraint through the clever use of servers to aggregate low rate tasks. Also, the combination of such servers with the idle scheduling idea leads to the original reduction to uniprocessor approach exposed in this dissertation, the first to our knowledge, not based on Pfair and proven to be optimal for periodic task systems with implicit deadlines.

Chapter 3

The reduction of the number of task of a general task system is obtained by aggregating many low rate tasks compared to one into single servers of accumulated rate less than or equal to one. Since a server can schedule its clients on a single processor, we establish the properties of servers simply considering a uniprocessor system.

TASKS AND SERVERS

3.1 INTRODUCTION

As briefly mentioned in Section 1.7 of Chapter 1, the partitioned proportionate fairness (PP-Fair) approach relies on the aggregation of low rate tasks, the *clients*, into virtual scheduling entities, the *servers*, such that each server has high, but less than one, accumulated rate.

For instance, in the first off-line step of the Reduction to Uniprocessor (RUN) scheduling procedure, a set of servers is defined such that each primal task is associated to a unique primal server. Regarding this primal system of servers, RUN is *partitioned proportionate fair* in the sense that each server is guaranteed to execute at a fixed rate, equal to the sum of its clients' rates, between any two of its clients' deadlines.

Since tasks and servers play a central role in the RUN algorithm, we dedicated this chapter to their precise definition and to the description of their properties.

Note that the concept of task servers has been extensively used to provide a mechanism to schedule soft real-time tasks (LIU, 2000), for which timing attributes like period or execution time are not known *a priori*. There are server mechanisms for uniprocessor systems which share some similarities with the one presented here (DENG et al., 1997; SPURI; BUTTAZZO, 1996). Other server mechanisms have been designed for multiprocessor systems, *e.g.,* (MOIR; RAMAMURTHY, 1999; ANDERSSON; TOVAR, 2006; ANDERSSON et al., 2008). However, unlike such approaches, the mechanism described here works as if each server were a uniprocessor system providing a useful scheduling framework which hides some complexities related to the multiprocessor scheduling problem.

Structure of the chapter

In Section 3.2, we introduce a slightly more general specification for a real-time task than the usual periodic-preemptive and independent with implicit deadlines (PPID) task model. Then, we present the fully-utilization system assumption adopted in this dissertation in Section 3.3, before to step into the full description of a server and its properties in Section 3.4. The Chapter finishes with a discussion of partial knowledge in Section 3.5 and partitioned proportionate fairness in Section 3.6.

3.2 FIXED-RATE TASK MODEL

Recall from Section 1.2.1 that a real-time job J is a finite sequence of instructions to be executed on one or more processors with a release instant $J.r$, a worst-case execution time $J.c$ and a deadline $J.d$.

In order to represent possibly non-periodic execution requirements, like those of servers in particular, we introduce a general real-time object, called *fixed-rate task*, whose execution requirement is specified in terms of processor utilization within a given interval. Since a task shall be able to execute on a single processor, its utilization cannot be greater than one. Although the definition of a fixed-rate task is slightly different from the usual definition of real-time task as given in Section 1.2.2, we somehow abusively will simply call task a fixed-rate task when no confusion is introduced doing so in the remaining of this dissertation.

Definition 3.2.1 (Fixed-Rate Task)**.** *Let $\rho \leqslant 1$ be a positive real number and K a countable and unbounded set of positive real numbers, possibly including zero. The* fixed-rate task *τ with rate ρ and release instants K, denoted $\tau{:}[\rho, K]$, releases an infinite sequence of jobs satisfying the following properties:*

i) A job J of τ is released at time t if and only if t is in K;

ii) The deadline $J.d$ of job J released at time $J.r$ equals $\min_t\{t \in K, t > J.r\}$;

iii) The execution time $J.c$ of job J released at time $J.r$ equals $\rho(J.d - J.r)$.

As can be seen from point (ii) of this definition, we assume an implicit deadline model, *i.e.,* the deadline of τ's job is precisely equal to the release instant of the next job of τ. As a consequence, $K \backslash \{\min(K)\}$ is also the set of all deadlines of jobs of task τ.

Given a fixed-rate task τ, we denote $\rho(\tau)$ its rate and $\mathrm{R}(\tau)$ the set of the release instants of its jobs.

As a simple example of task, consider a periodic task τ characterized by three attributes: (i) its start time s; (ii) its period T; and (iii) its execution requirement C. Task τ generates

an infinite collection of jobs each of which released at time $s + kT$ and with deadline at time $s + (k + 1)T$, for $k \in \mathbb{N}$. Hence, τ can be seen as a fixed-rate task with start time s, rate $\rho(\tau) = C/T$ and set of release instants $\mathrm{R}(\tau) = \{(s + kT), k \in \mathbb{N}\}$, which requires exactly $\rho(\tau)T$ of processor during each of its scheduling windows $[s + kT, s + (k + 1)T)$, for $k \in \mathbb{N}$.

Figure 3.1 illustrates a generic example of schedule of jobs J_{k-1}, J_k and J_{k+1} of a fixed-rate task τ with rate $\rho(\tau) = 1/2$.

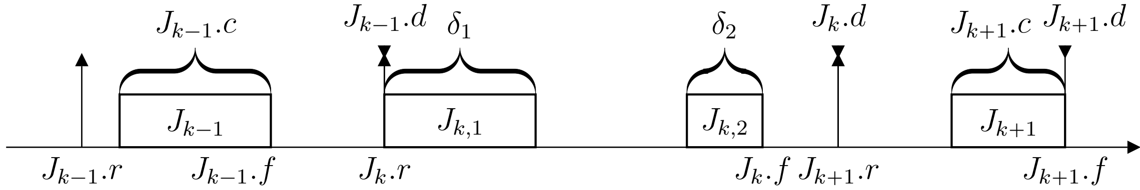


Figure 3.1. Schedule example of jobs J_{k-1}, J_k and J_{k+1} of a fixed-rate task τ where $\rho(\tau) = 1/2$ and $\delta_1 + \delta_2 = J_k.c$.

Definition 3.2.2 (Accumulated Rate)**.** *Let $\mathcal{T}$ be a set of fixed-rate tasks. We say that $\mathcal{T}$ has an* accumulated rate *equal to the sum of the rates of the tasks in $\mathcal{T}$, and denote this by $\rho(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} \rho(\tau)$.*

We use the more general model of a fixed-rate task because it can also represent groups of tasks, with rate equal to the accumulated rate of the group of tasks and deadlines equal to the union of the tasks' group deadlines.

3.3 FULLY UTILIZED SYSTEM

In the remaining of this dissertation, we consider a real-time system comprised of n fixed-rate and independent tasks to be scheduled by a global scheduling algorithm on a platform Π comprised of m identical, uniform and preemptive processors. Tasks may migrate freely and instantaneously between processors with no penalty.

Although one of our goals in this dissertation is to minimize preemptions and migrations, our calculations make the standard assumption that each of these two events take zero time. Albeit this assumption may seem "incorrect", it is acceptable since, in an real system, measured preemption and migration overheads can be accommodated by adjusting the task execution times.

Definition 3.3.1 (Fully-Utilized System)**.** *Let $\mathcal{T}$ be a set of fixed-rate tasks to be scheduled on a multiprocessor platform Π. We say that Π is* fully utilized *by $\mathcal{T}$ if the accumulated rate of $\mathcal{T}$ exactly equals m, the number of processors in Π.*

Hereafter and when no mention of the contrary is done, we only consider task / processor systems for which the *full utilization* assumption holds *i.e.,* the set of n fixed-rate tasks fully utilizes all the processors in the system.

It is important to emphasize that the full utilization assumption does not restrict the applicability of the proposed approach.

Consider, for instance, that a job J of a task is supposed to require $J.c$ time units of processor but that it completes consuming only $c' < J.c$ processor units. In such a case, the system can easily simulate $J.c - c'$ of J's execution by blocking a processor accordingly. That is, if a job does not require its full worst-case execution time estimate, we may fill in the difference with forced idle time.

Another situation occurs when start times of tasks are known but different from zero. Suppose, for example, that task τ has its first initial job release at some time $s > 0$ and that s is known at time 0. In such a case, we may add a dummy job J_0 with release time 0, deadline s and execution time $J_0.c = \rho(\tau)s$.

Finally, if the accumulated rate of the task set to be scheduled is less than the number of processors, idle tasks may be inserted as needed to fill in slack in order to comply with the full utilization assumption. As a matter of fact, the careful use of any existing slack may significantly improve the performance of the system by allowing some interesting aggregation of tasks into servers. For instance, when there exists some slack in the task system, it is more likely that the resulting set of servers produced by the DUAL operation can be efficiently scheduled by local EDF, as will be shown in Chapter 6.

Hence, without loss of generality, we consider hereafter that the full utilization assumption holds and so each job J of a task τ executes exactly for $\rho(\tau)(J.d - J.r)$ time units within its scheduling window $[J.r, J.d)$.

Lemma 3.3.1. *Let $\mathcal{T}$ be a task set which fully-utilizes m identical processors. If Σ is a valid schedule of $\mathcal{T}$ as defined by Definition 1.4.5, then exactly m jobs must be executing at all times in Σ* i.e., *$|\Sigma(t)| = m$ for all $t \geqslant 0$.*

Proof. Suppose that there exists a time interval I during which less than m jobs execute. Then, at least one processor must be idle during I. Since $\mathcal{T}$ fully-utilizes m processors, there must exist an interval $[t, t')$ after I by which the total workload generated by $\mathcal{T}$ is greater than $m(t' - t)$. Hence a deadline miss must occur during $[t, t')$ and this contradicts the hypothesis that Σ is valid. □

3.4 SERVERS

As mentioned in Section 1.7, the derivation of a schedule for a multiprocessor system will be done by generating a schedule for a series of equivalent uniprocessor systems using the Dual Scheduling Equivalence approach. But, prior to this, one may need to aggregate tasks into servers via a PACK operation. However, since an aggregated task set must be feasible on a single processor, we require that the rate of a server is not greater than one.

Hence, in this section we will not deal with the multiprocessor scheduling problem. The focus here is on precisely defining the concept of server (Section 3.4.1) and showing that a server correctly schedules the tasks associated to it (Section 3.4.2). In other words, one can assume in this section that there is a single processor in the system. Later on we will show how multiple servers are scheduled on a multiprocessor system by means of the Dual Scheduling Equivalence approach.

3.4.1 Server model and notations

We treat servers as fixed-rate tasks with a sequence of jobs, but they are not actual tasks in the system. In brief, each server can be seen as a proxy for a collection of *client* tasks that it schedules according to an internal scheduling policy. Somehow abusively, we shall say that a server is executing on a processor when the processor time is consumed by one of its clients.

We now give a precise definition of the server for a set of tasks.

Definition 3.4.1 (Server/Client)**.** *Let $\mathcal{T}$ be a set of fixed-rate tasks with total rate given by $\rho(\mathcal{T}) = \sum_{\tau\in\mathcal{T}} \rho(\tau) \leqslant 1$. A* server *$S$ for $\mathcal{T}$, denoted $\mathrm{ser}(\mathcal{T})$, is a virtual task with rate $\rho(\mathcal{T})$, release instant set $\mathrm{R}(S) = \bigcup_{\tau\in\mathcal{T}} \mathrm{R}(\tau)$ and equipped with a work-conserving scheduling policy to schedule the tasks in $\mathcal{T}$. A task in $\mathcal{T}$ is called an S's client and $\mathcal{T}$ is the set of S's* clients, *also denoted $\mathrm{cli}(S)$.*

We refer to a job of any client of S as a *client job* of S. If S is a server and Γ a set of servers, then $\mathrm{ser}(\mathrm{cli}(S)) = S$ and $\mathrm{cli}(\mathrm{ser}(\Gamma)) = \Gamma$. Also, if S_1 and S_2 are two clients of the same server S, then we say that S_1 and S_2 are siblings.

By Definition 3.4.1, the execution requirement of a server S in any interval $[r_i, r_{i+1})$ equals $\rho(S)(r_{i+1} - r_i)$, where r_i and r_{i+1} are consecutive release instants in $\mathrm{R}(S)$. Then the workload for job J of server S with $J.r = r_i$ and $J.d = r_{i+1}$ equals $J.c = e(J, J.r) = \rho(S)(J.d - J.r)$, just as with a "real" job.

However, just as a server S is a proxy for its clients, so too are the "jobs" of server S, which represent the budget allocated to S so that its clients' jobs may execute. Hence, we refer to a job J_i^S of server S as a *budget job* with the following interpretation. At each time

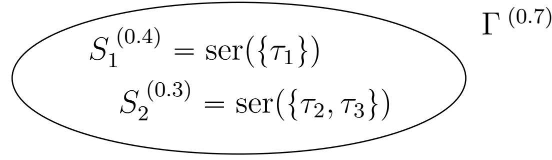


Figure 3.2. A two-server set. The notation $X^{(\rho)}$ means that $\rho(X) = \rho$.

r_i in $\mathrm{R}(S)$, server S replenishes its budget for the interval $[r_i, r_{i+1})$ with $r_{i+1} = \min_t\{t \in \mathrm{R}(S), t > r_i\}$ by releasing a budget job J_i^S with $J_i^S.r = r_i$ and $J_i^S.d = r_{i+1}$. As a consequence, at any given time t, the budget of S just equals $e(J_i^S, t)$, where J_i^S is the current budget job of S at time t.

As will become clearer in Chapter 4, the PACK operation is an off-line procedure which statically allocates tasks to servers. As a consequence, the client/server relationships are invariant during an on-line schedule of the task system. This allows us to consistently define the rate $\rho(S)$ of server S to be $\rho(\mathrm{cli}(S))$.

As an example, consider Figure 3.2, where Γ is a set comprised of the two servers $S_1 = \mathrm{ser}(\{\tau_1\})$ and $S_2 = \mathrm{ser}(\{\tau_2, \tau_3\})$ for the tasks τ_1, and τ_2 and τ_3, respectively. If $\rho(\tau_1) = 0.4$, $\rho(\tau_2) = 0.2$ and $\rho(\tau_3) = 0.1$, then $\rho(S_1) = 0.4$ and $\rho(S_2) = 0.3$. Also, if $S = \mathrm{ser}(\Gamma)$ is the server in charge of scheduling S_1 and S_2, then $\Gamma = \mathrm{cli}(S) = \{S_1, S_2\}$ and $\rho(S) = 0.7$.

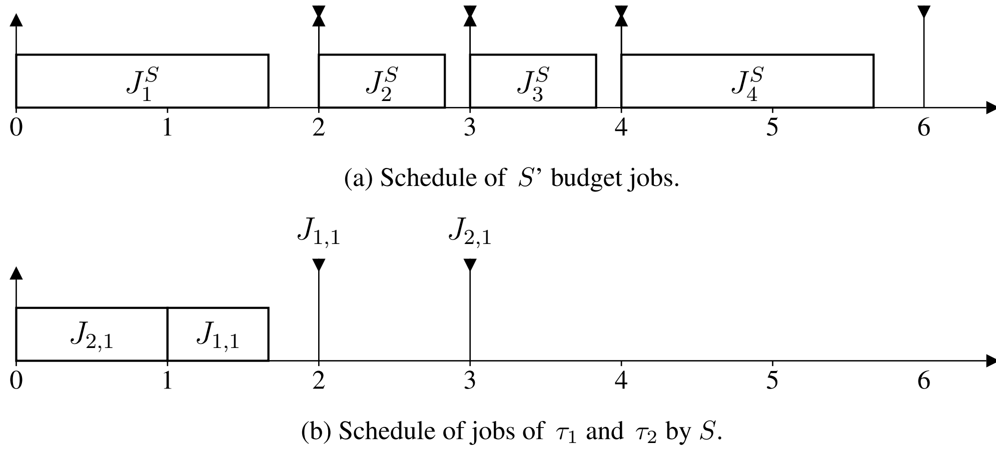
Note that since servers are themselves tasks, a set of servers of accumulated rate not greater than one can be served by another "meta" server. Hence, we may speak of a server for a set of servers. On the other hand, a server may have a single task as only client. In such a case, the budget jobs of the server have the same deadlines and execution time as the "real" jobs of the task. Since we assume that the scheduling policy of servers is work-conserving, there are no difference between scheduling a single task τ or its dedicated server $\mathrm{ser}(\{\tau\})$. Hence, we see that the concepts of fixed-rate task and server are largely interchangeable.

As task set with accumulated rate exactly equal to one will play a special role in this dissertation, we define a *unit set* and a *unit server*, both of which are feasible on a single processor.

Definition 3.4.2 (Unit Set/Unit Server)**.** *A set* Γ *of tasks/servers is a* unit set *if* $\rho(\Gamma) = 1$. *The server* $\mathrm{ser}(\Gamma)$ *for a unit set* Γ *is a* unit server.

We say that a server meets its deadlines when all of its budget jobs meet theirs deadlines. However, even if a server meets all its deadlines, it must use an appropriate scheduling policy to ensure that its clients meet theirs.

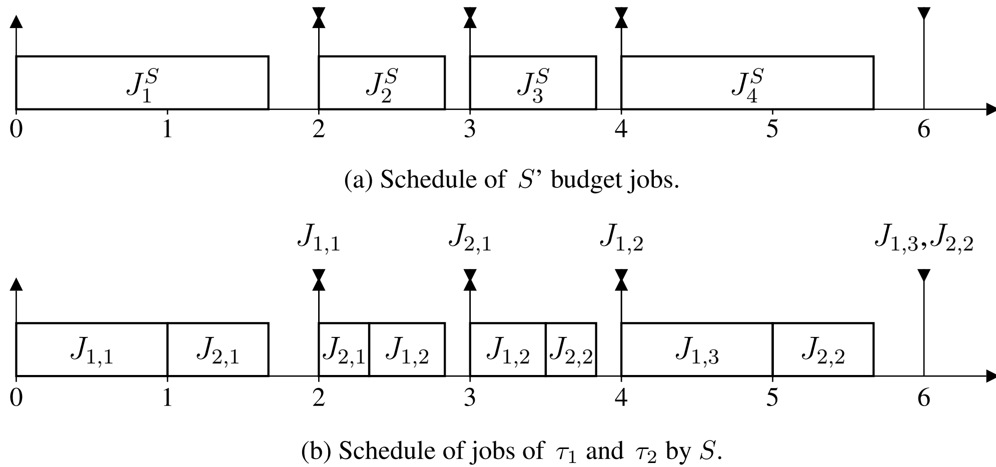
For example, consider two periodic tasks $\tau_1{:}[1/2, 2\mathbb{N}]$ and $\tau_2{:}[1/3, 3\mathbb{N}]$, with periods equal to 2 and 3 and rates $\rho(\tau_1) = 1/2$ and $\rho(\tau_2) = 1/3$, respectively. Assume a synchronous task system, *i.e.,* start times equal to zero. Consider a server S scheduling these two tasks on



(a) Schedule of S' budget jobs.

(b) Schedule of jobs of τ_1 and τ_2 by S.

Figure 3.3. Schedule of τ_1:[$1/2, 2\mathbb{N}$] and τ_2:[$1/3, 3\mathbb{N}$] by a single server S with $\mathrm{R}(S) = \{2, 3, 4, 6, \ldots\}$ and $\rho(S) = 5/6$ on a dedicated processor. If S schedules job $J_{2,1}$ of τ_2 first, then job $J_{1,1}$ of τ_1 misses its deadline at time 2.



(a) Schedule of S' budget jobs.

(b) Schedule of jobs of τ_1 and τ_2 by S.

Figure 3.4. Valid schedule of τ_1:[$1/2, 2\mathbb{N}$] and τ_2:[$1/3, 3\mathbb{N}$] by a single server S equipped with EDF with $\mathrm{R}(S) = \{2, 3, 4, 6, \ldots\}$ and $\rho(S) = 5/6$ on a dedicated processor.

a dedicated processor. We have $\mathrm{R}(S) = \{0, 2, 3, 4, 6, \ldots\}$ and $\rho(S) = 5/6$. For instance, the budget of S available during $[0, 2)$ equals $e(J_0^S, 0) = \rho(S)(2 - 0) = 5/3$; that is, S releases a budget job J_0^S at time $t = 0$ with workload $5/3$ and deadline 2.

Now, consider a valid schedule of S. For example, a valid schedule of the three first budget jobs of S is represented in Figure 3.3, assuming that S executes whenever its budget is non zero. In this figure, $J_{i,j}$ represents the jth job of τ_i. As can be seen, server S acquires the processor for exactly $5/3$ units of time during $[0, 2)$ in Σ. However, suppose that the scheduling policy used by S to schedule its client tasks gives higher priority to job $J_{2,1}$ of τ_2 at time 0. Then $J_{2,1}$ will consume one unit of time before $J_{1,1}$ begins its execution. Therefore, the remaining budget $e(J_0^S, 1) = 2/3$ will be insufficient to complete $J_{1,1}$ by its deadline at time 2.

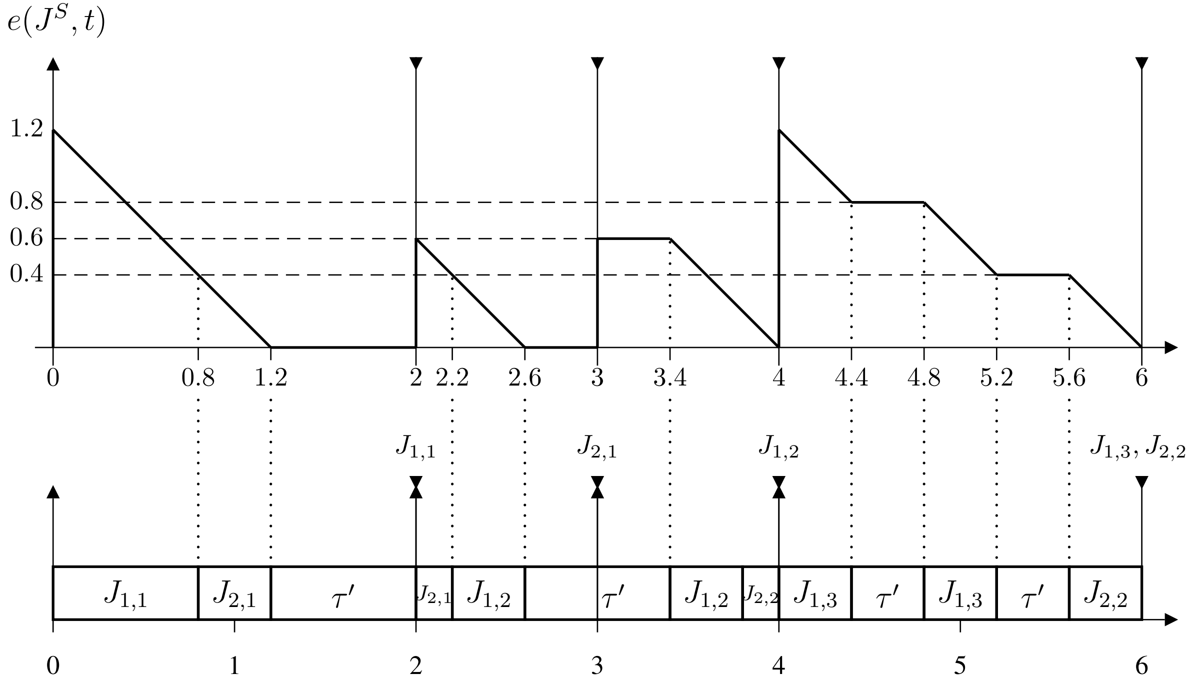


Figure 3.5. Budget management and schedule of EDF-server S with $\text{cli}(S) = \{\tau_1 :[0.4, 2\mathbb{N}], \tau_2 :[0.2, 3\mathbb{N}]\}$ and $\rho(S) = 0.6$. Task τ' represents the execution of external events which alternate with the execution of S.

This simple example shows that a server can meet its deadlines even when its clients do not. However, if the scheduling policy gives higher priority to τ_1 at time zero, as would do the earliest deadline first (EDF) algorithm for instance, this deadline miss would be avoided, as illustrated by Figure 3.4.

3.4.2 EDF Server

We use EDF as scheduling policy to equip servers in order to ensure optimality within each server.

Rule 3.4.1 (EDF Server)**.** *An* EDF server *is a server that schedules its client jobs according to the EDF scheduling policy.*

For example, consider a set of two periodic tasks $\mathcal{T} = \{\tau_1 :[0.4, 2\mathbb{N}], \tau_2 :[0.2, 3\mathbb{N}]\}$. Since $\rho(\mathcal{T}) = 0.6 \leqslant 1$, we can define an EDF server S to schedule $\mathcal{T}$ such that $\text{cli}(S) = \mathcal{T}$ and $\rho(S) = 0.6$. Figure 3.5 shows both the evolution of $e(J_S, t)$ during interval $[0, 6)$ and the schedule Σ of $\mathcal{T}$ by S on a single processor. As previously, $J_{i,j}$ represents the jth job of τ_i. During intervals $[1.2, 2)$, $[2.6, 3.4)$, $[4.4, 4.8)$ and $[5.2, 5.6)$, the execution of S alternates with the execution of external events represented by task τ'.

Note that a unit EDF server S has rate $\rho(S) = 1$ and must execute continuously in order to meet its clients' deadlines. As a consequence, deadlines of S have no effect since, whenever an S's budget job is null, a new budget job of S is released.

Theorem 3.4.1 (EDF Server)**.** *The EDF server $S = \mathrm{ser}(\Gamma)$ of a set of servers Γ produces a valid schedule of Γ when $\rho(\Gamma) \leqslant 1$ and all jobs of S meet their deadlines.*

The proof which follows is based on well-known results upon real-time task systems. In Appendix B, we give a direct proof of this theorem which follows the same sketch as the proof of EDF optimality as given by (LIU; LAYLAND, 1973).

Proof. By treating the servers in Γ as tasks, we can apply well known results for scheduling task systems. For convenience, we assume that S executes on a single processor; this need not be the case in general, as long as S does not execute on multiple processors in parallel.

Recall from Definition 3.4.1 that $\rho(\Gamma) = \sum_{S_i \in \Gamma}$. We first prove the theorem for $\rho(\Gamma) = \sum_{S_i \in \Gamma} \rho(S_i) = 1$ and thereafter for $\rho(\Gamma) < 1$.

Case $\rho(\Gamma) = 1$.

Let $\eta_\Gamma(t, t')$ be the execution demand within a time interval $[t, t')$, where $t < t'$. This demand gives the sum of all execution requests (*i.e.,* jobs) that are released no earlier than t and with deadlines no later than t'. By Definition 3.4.1 of server, this quantity is bounded above by

$$\eta_\Gamma(t, t') \leqslant (t' - t) \sum_{S_i \in \Gamma} \rho(S_i) = t' - t \tag{3.1}$$

Also, it is known that there is no valid schedule for Γ if and only if there is some interval $[t, t')$ such that $\eta_\Gamma(t, t') > t' - t$ (BARUAH; GOOSSENS, 2004; BARUAH et al., 1990). Since Equation 3.1 implies that this cannot happen, some valid schedule for Γ must exist. Because S schedules Γ using EDF and EDF is optimal (LIU; LAYLAND, 1973; BARUAH; GOOSSENS, 2004), S must produce a valid schedule.

Case $\rho(\Gamma) < 1$.

In order to use the result for case $\rho(\Gamma) = 1$, we introduce a slack-filling task τ', as illustrated in Figure 3.5, where $\mathrm{R}(\tau') = \mathrm{R}(S)$ and $\rho(\tau') = 1 - \rho(S)$. We let $\Gamma' = \Gamma \cup \{\tau'\}$, and let S' be an EDF server for Γ'. Since $\rho(\Gamma') = 1$, S' produces a valid schedule for Γ'.

Let us now consider the scheduling window $W_J = [J.r, J.d]$ for a budget job J of S. Since $\mathrm{R}(\tau') = \mathrm{R}(S)$, τ' also has a job J' where $J'.r = J.r$ and $J'.d = J.d$. Also, since S' produces a valid schedule, τ' and S do exactly $\rho(\tau')(J.d - J.r)$ and $\rho(S)(J.d - J.r)$ units of work, respectively, during W_J.

Further, by the definition of τ', there are no deadlines or release instants of τ' between $J.r$

and $J.d$. Consequently, the workload of τ' may be arbitrarily rearranged or subdivided within the interval W_J without compromising the correctness of the schedule. Also, we may do this for all budget jobs of S so as to reproduce *any* schedule of S where it meets its deadlines.

Finally, since S and S' both schedule tasks in Γ with EDF, S will produce the same *valid* schedule for Γ as S', giving our desired result. □

3.5 PARTIAL KNOWLEDGE

As first pointed by Greg Levin in (REGNIER et al., 2011), a server can correctly schedule its clients without the need to "know" all arrival times of its clients' jobs at the outset. Indeed, the assumed system model only requires that there are no gaps or overlaps between jobs of task or server. In other words, the deadline of a job of a task or server is the release instant of the next job of the same task or server. As a consequence, at any time, the knowledge of the earliest deadline of an EDF server S's clients is the minimum and sufficient information that S needs to take scheduling decision for its clients.

In practice, this knowledge is sufficient in order for a server to estimate its budget job for its next scheduling window. In other words, the required and sufficient knowledge horizon of a server is its next deadline. This is an important distinction with the PPID task model. Indeed, unlike periodic tasks where **all** deadlines are known at the outset, the fixed-rate task model allows for jobs whose complete set of deadlines is not known *a priori*.

However, the rates and first release instants of tasks and servers are parameters which must be known prior the execution of the system.

3.6 PARTITIONED PROPORTIONATE FAIRNESS

Unlike previous Proportionate Fairness (Pfair) based approaches, client tasks scheduled by servers do not receive their proportional shares between each system deadline nor between each server deadline. Instead, each aggregating server responsible for scheduling a group of servers, is guaranteed a constant processor bandwidth. Hence, according to the *Partitioned Proportionate Fairness* approach (PP-Fair), the total bandwidth available in the system is "fairly" shared between all aggregating servers, each of which is

i) guaranteed a budget proportional to the sum of its clients' rates between any two consecutive deadlines of its clients;

ii) responsible for scheduling its clients in some correct fashion (*e.g.,* EDF) between such deadlines.

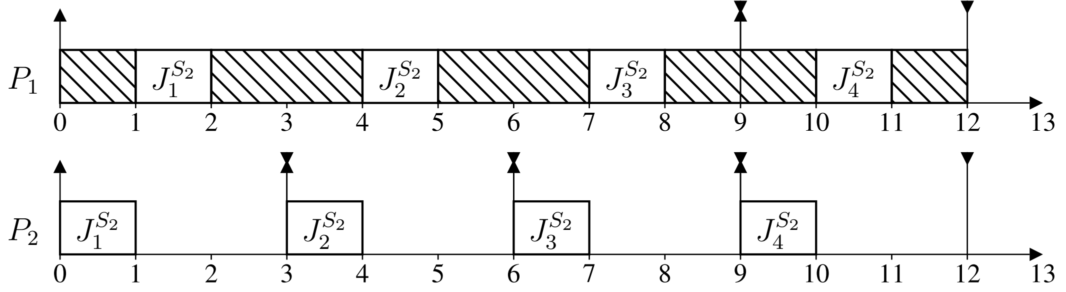


Figure 3.6. Crosshatch regions represent the scheduling windows available to S_1 for scheduling its clients, given a certain schedule of $S_2 = \text{ser}(\tau_3 : [2/3, 3\mathbb{N}])$.

In order to illustrate the strength of the PP-Fair approach, let us consider a task set $\mathcal{T}$ comprised of two tasks: $\mathcal{T} = \{\tau_1 : [1/3, 9\mathbb{N}],\ \tau_2 : [1/3, 12\mathbb{N}]\}$. Suppose that $\mathcal{T}$ is scheduled by a dedicated server $S_1 = \text{ser}(\mathcal{T})$ on a single processor P_1 and assume that the remaining rate available on P_1 is partially used by another server $S_2 = \text{ser}(\tau_3)$ of another task $\tau_3 : [2/3, 3\mathbb{N}]$. Also, assume that S_2 is partially scheduled on P_1 with rate of $1/3$ and partially scheduled on another processor P_2 with rate of $1/3$. Finally, suppose that jobs of S_2 always have highest priority among jobs scheduled on P_2 and that they are first scheduled on P_2 and, thereafter, scheduled on P_1.

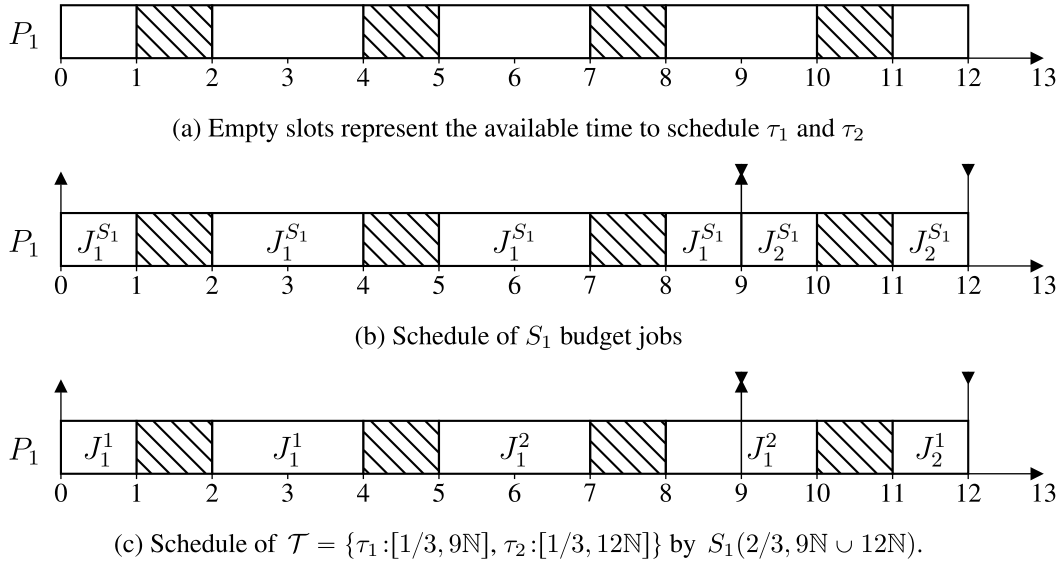
Note that these assumptions upon the schedule of S_2 do not affect the generality of this example since jobs of S_2 always have deadlines earlier or equal to those of S_1's jobs. Hence, as stated in the first proof of Theorem 3.4.1, the interference generated by any job J of S_2 on the execution of S_1 can be arbitrary distributed during J's scheduling window without consequences upon the correctness of the schedule of S_1's clients.

Figure 3.6 illustrates the assumed constraint generated by S_2 on P_1. Note that S_1 can execute on P_1 whenever no job of S_2 does. Thus, crosshatch regions in Figure 3.6 represent the time slots of processor P_1 which are available for the execution of S_1's client jobs.

From the point of view of S_1, the execution of S_2 on P_1 can be viewed as anonymous blocking times of P_1. We represent them by crosshatch regions in Figure 3.7(a),(b) and (c). Hence, Figure 3.7(a) depicts the empty slots available for scheduling jobs that are not clients of S_2 on P_1 and Figure 3.7(b) depicts the schedule of S_1's budget jobs on P_1.

Finally, the schedule of τ_1 and τ_2 by the EDF-server S_1 on P_1 is illustrated by Figure 3.7(c). As can be seen, jobs J_1^1 and J_1^2 suffer exactly one preemption each, caused by the blocking generated by the execution of S_2.

Now, consider the schedule of the same task set that would be generated by a Pfair algorithm on P_1, suffering the same interference from S_2. Each job of τ_1 and τ_2, constrained by the deadlines of S_2, would be split into sub-jobs of execution time 1 in each of the scheduling

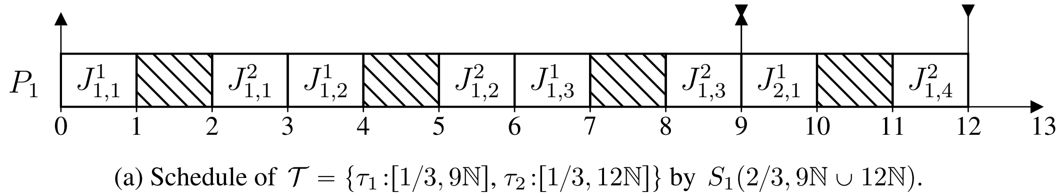


(a) Empty slots represent the available time to schedule τ_1 and τ_2

(b) Schedule of S_1 budget jobs

(c) Schedule of $\mathcal{T} = \{\tau_1 : [1/3, 9\mathbb{N}], \tau_2 : [1/3, 12\mathbb{N}]\}$ by $S_1(2/3, 9\mathbb{N} \cup 12\mathbb{N})$.

Figure 3.7. Schedule of $\mathcal{T} = \{\tau_1 : [1/3, 9\mathbb{N}],\ \tau_2 : [1/3, 12\mathbb{N}]\}$ by $S_1(2/3, 9\mathbb{N} \cup 12\mathbb{N})$. Crosshatch regions represent the constraints generated by S_2 on the execution of S_1 on P_1.



(a) Schedule of $\mathcal{T} = \{\tau_1 : [1/3, 9\mathbb{N}], \tau_2 : [1/3, 12\mathbb{N}]\}$ by $S_1(2/3, 9\mathbb{N} \cup 12\mathbb{N})$.

Figure 3.8. Schedule of $\mathcal{T} = \{\tau_1 : [1/3, 9\mathbb{N}],\ \tau_2 : [1/3, 12\mathbb{N}]\}$ by a proportionate fairness approach. J_1^1 and J_1^2 are split into 3 and 4 sub-jobs $J_{1,1}^1$, $J_{1,2}^1$, $J_{1,3}^1$ and $J_{1,1}^2$, $J_{1,2}^2$, $J_{1,3}^2$, $J_{1,4}^2$ with deadlines 3, 6, 9 and 3, 6, 9, 12, respectively.

window $[0, 3)$, $[3, 6)$, $[6, 9)$, $[9, 12)$. The resulting schedule is shown in Figure 3.8.

In this schedule, J_1^1 and J_1^2 suffer 2 and 3 preemptions respectively, which is more than twice as much as in the PP-Fair schedule. Although the Pfair schedule could be optimized in many aspects, the splitting of jobs into sub-jobs would still cause unnecessary preemptions as compared with the PP-Fair approach.

In summary, in PP-Fair scheduling, the execution of a server's job ensures that its set of clients collectively gets its proportional share of processor time between each server deadline, *i.e.,* between the deadlines of the server clients. Thus, according to Theorem 3.4.1, PP-Fair scheduling guarantees the correct scheduling of a server's clients. This approach applies much weaker over-constraints to the system than traditional proportionate fairness, and thus requires significantly fewer preemptions and migrations for optimal scheduling.

Also, we will show in Chapter 6, under the Dual Scheduling Equivalence (DSE) scheme, a deadline of a particular server can only generate one preemption on another server. The combination of these two facts explains that the RUN algorithm significantly reduces overhead compared to algorithms based on standard proportional fairness.

3.7 CONCLUSION

In this chapter, we have dealt with the server abstraction. In particular, we have shown that an EDF server is capable to optimally schedule its clients on a uniprocessor system, provided that all of its budget jobs meet their deadlines.

Moreover, as mentioned earlier, a server and its clients may migrate between processors, as long as no more than one client executes at a time. As a consequence, the server abstraction is a powerful instrument to schedule a general set of tasks on a multiprocessor platform in order to ensure partitioned proportionate fairness between subsets of tasks aggregated into servers.

This fair and partitioned sharing of the multiprocessor system bandwidths can be used to achieve optimality in multiprocessor systems, which is the topic of the next chapter.

Chapter 4

Virtual Scheduling by reduction to uniprocessor ensures partitioned proportionate fairness (PP-Fair)*, which imposes a less restrictive set of constraints than those imposed by* proportionate fairness *(Pfair) for scheduling periodic real-time tasks in multiprocessors. PP-Fairness can be achieved by the composition of duality and packing. Moreover, this efficient combination allows for reducing a general task system with integer utilization greater than or equal to two to a system of unit servers which can be efficiently scheduled on uniprocessor systems.*

VIRTUAL SCHEDULING

In this chapter, we describe three operations, DUAL, PACK and REDUCE, which iteratively reduce the number of processors in a multiprocessor fixed-rate task or server system until a set of uniprocessor server systems is obtained.

At execution time, the schedules for these unit servers are generated by the EDF scheduling policy. Then, from these uniprocessor schedules, the corresponding schedule for the original multiprocessor fixed-rate task system is deduced straightforwardly by following simple rules.

4.1 INTRODUCTION

As introduced in Section 1.7, the DUAL operation transforms a server S into the dual server S^*, whose execution time represents the idle time of S. Since $\rho(S^*) = 1 - \rho(S)$, the DUAL operation reduces the total rate and the number of required processors in systems where most tasks have high rates, *i.e.,* rates close or equal to one. Also, we will see in Section 4.2 that the Dual Scheduling Equivalence (DSE) can be efficiently used to deduce a valid schedule for those particular sets of high-rate servers from the schedule of the set of their dual servers.

Next, we will show in Section 4.3 that such high-rate servers can always be generated via a PACK operation. Indeed, sets of tasks whose rates sum to no more than one can be packed into servers, reducing the number of tasks and producing the high-rate servers needed by the DSE rule.

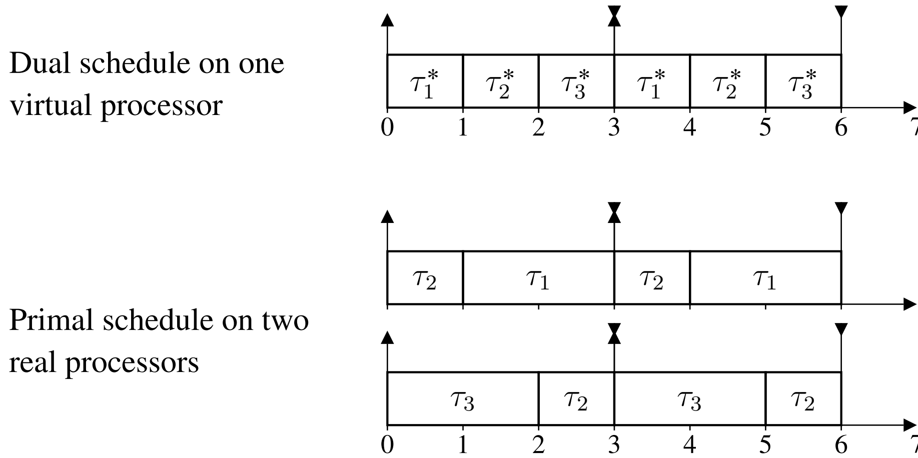


Figure 4.1. Dual Scheduling Equivalence (DSE) of the *primal* task set $\{\tau_1,\ \tau_2,\ \tau_3\}$ on two *real* processors and its *dual* task set $\{\tau_1^*,\ \tau_2^*\ \ \tau_3^*\}$ on one *virtual* processor.

Given this synergy, we compose the two operations DUAL and PACK into a single REDUCE operation, which will be defined in Section 4.4. The REDUCE operation permits to iteratively reduce the number of processors in a multiprocessor system until a set of uniprocessor systems is derived. Thus, after a sequence of REDUCE operations, the schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems. While the reduction from the original system to the virtual ones is carried out off-line, the generation of these various systems' schedules can be efficiently done on-line, leading to the optimal RUN algorithm proposed in this dissertation and that will be described in Chapter 5.

4.2 DUAL OPERATION

The simple example given in Section 1.7 with the primal task set $\mathcal{T}$ defined as $\{\tau_1\!:\!(2,3),\ \tau_2\!:\!(2,3),\ \tau_3\!:\!(4,6)\}$ is a particular case in which the number of tasks to be scheduled is precisely equal to the number of processors plus one, *i.e.,* $|\mathcal{T}| = m+1$. In this particular case previously discussed in (LEVIN et al., 2009), the schedule of $\mathcal{T}$ can be deduced by a simple procedure from the schedule of its dual task set $\mathcal{T}^* = \{\tau_1^*,\ \tau_2^*,\ \tau_3^*\}$ on a single processor. That is, whenever a dual task is scheduled on a virtual processor, its primal task does not execute, and vice versa. This is illustrated by Figure 1.6 from Section 1.7 reproduced in Figure 4.1.

In this dissertation, we enunciate the Dual Scheduling Equivalence (DSE), which is a generalization of previous results, in terms of servers and their dual servers defined as follows.

Definition 4.2.1 (Dual Server)**.** *The* dual server S^* *of a server* S *is a server with the same deadlines as* S *and with rate* $\rho(S^*)$ *equal to* $1-\rho(S)$*. If* Γ *is a set of servers, then its dual set* Γ^* *is the set of dual servers to those in* Γ, i.e., $S \in \Gamma$ *if and only if* $S^* \in \Gamma^*$.

Note that the dual server of a primal unit server S, which has rate $\rho(S) = 1$ and must execute continuously in order to meet its clients' deadlines, is a *null server*, which has rate $\rho(S) = 0$ and never executes.

As usual with duality, the relation $(S^*)^* = S$ holds. Hereafter, S is referred to as the *primal server* of its dual server S^*. We now enunciate the definition of the dual schedule of a schedule of primal servers.

Definition 4.2.2 (Dual Schedule)**.** *Let Γ be a set of primal servers and Γ^* be its dual set. Two schedules Σ of Γ and Σ^* of Γ^* are duals if, for all times t and all $S \in \Gamma$, $S \in \Sigma(t)$ if and only if $S^* \notin \Sigma^*(t)$; that is, S executes exactly when S^* is idle, and vice versa.*

Like for servers, Γ, and Σ are referred to as *primal* relative to their duals Γ^*, and Σ^*. Here again, $(\Gamma^*)^* = \Gamma$ and $(\Sigma^*)^* = \Sigma$. As a matter of fact, this latter identity is our main motivation for adopting the unusual definition of schedule as given in Section 1.4.1. Indeed, recall that according to Definition 1.4.1, a schedule does not specify **on which** processor each server executes at any time as usual in real-time literature. Instead, a schedule of a set of servers Γ just specifies **which** subset of servers in Γ execute at any time. Then, the assignment of the subset of server jobs' chosen to execute on the processors is done by the job-to-processor assignment step as previously described in Section 1.4.3.

This disjunction between the scheduling step and the job-to-processor assignment step is which allows for the identity $(\Sigma^*)^* = \Sigma$ to be true, as expected for any "good" notion of duality.

We now establish the Dual Scheduling Equivalence (DSE) which states that the schedule of a primal set of servers is valid precisely when its dual schedule is valid. This equivalence is enunciated for server set with integer rate. However, this assumption does not imply any loss of generality. Indeed, consider a set of servers Γ with non-integer accumulated rate $\rho(\Gamma)$. The minimal integer m of processors needed to feasibly schedule Γ equals $\lfloor \rho(\Gamma) \rfloor + 1$. As previously explained in Section 3.3, we can complete Γ to obtain an integer accumulated rate task system, by adding a slack-filling server of rate $m - \rho(\Gamma)$. Thus, the result presented here can be applied to any server system with non-integer rate by filling it to achieve an integer rate.

Theorem 4.2.1 (Dual Scheduling Equivalence)**.** *Let Γ be a set of $n = m + k$ servers with $k \geqslant 1$ and such that the accumulated rate $\rho(\Gamma)$ of Γ equals m, an integer. Consider a schedule Σ of Γ on m processors and let Σ^* and Γ^* be the duals of Σ and Γ, respectively. Then $\rho(\Gamma^*) = k$, and so Γ^* is feasible on k processors. Further, Σ is valid if and only if Σ^* is valid.*

Proof. First,

$$\begin{aligned}\rho(\Gamma^*) &= \sum_{S^* \in \Gamma^*} \rho(S^*) \\ &= \sum_{S \in \Gamma} (1 - \rho(S)) \\ &= n - \rho(\Gamma) \\ &= k\end{aligned}$$

so k processors are sufficient to feasibly schedule Γ^*. Next, we prove that if Σ is valid for Γ then Definitions 1.4.1 and 1.4.5 implies that Σ^* is valid for Γ^*.

Because Σ is a valid schedule on m processors and we assume full utilization, Σ always executes m distinct tasks as shown by Lemma 3.3.1. The remaining $k = n - m$ tasks are idle in Σ, and so are exactly the tasks executing in Σ^*. Hence Σ^* is always executing exactly k distinct tasks on its k (virtual) processors. Also, since Σ is valid, any job J of server $S \in \Gamma$ does exactly $J.c = \rho(S)(J.d - J.r)$ units of work between its release instant $J.r$ and its deadline $J.d$. During this same time, S^* has a matching job J^* where $J^*.r = J.r$, $J^*.d = J.d$, and

$$\begin{aligned}J^*.c &= \rho(S^*)(J^*.d - J^*.r) \\ &= (1 - \rho(S))(J.d - J.r) \\ &= (J.d - J.r) - J.c\end{aligned}$$

That is, J^*'s execution time during the interval $[J.d, J.r)$ is exactly the length of time that J must be idle. Thus, as J executes for $J.c$ during this interval in Σ, J^* executes for $J^*.c$ in Σ^*. Consequently, J^* satisfies condition (ii) of Definition 1.4.1 and also meets its deadline. Since this holds for all jobs of all dual servers, Σ^* is a valid schedule for Γ^*.

The converse also follows from the above argument, since $(\Sigma^*)^* = \Sigma$. □

Once again, see Figure 4.1 for a simple illustration. We now summarize this dual scheduling rule for future reference.

Rule 4.2.1 (Dual Scheduling Equivalence). *At any time, execute in Σ the servers of Γ whose dual servers are not executing in Σ^*, and vice versa.*

Finally, we define the DUAL operation φ as follows.

Definition 4.2.3 (DUAL Operation). *The* DUAL *operation φ from a set of servers Γ to its dual set Γ^* is the bijection which associates a server S with its dual server S^*,* i.e., $\varphi(S) = S^*$.

In this dissertation, we adopt the usual definition for the image of a subset. That is, if $f : E \to F$ is a mapping from E to F and $G \subseteq E$ is a subset of E, then the image $f(G)$ of G by f is defined as

$$f(G) = \{f(x), x \in G\}$$

For example, if Γ is a set of server, then the dual set of Γ is $\varphi(\Gamma) = \{S^*, S \in \Gamma\} = \Gamma^*$.

It is important to emphasize that Theorem 4.2.1 does not establish any scheduling rule to generate feasible schedules. It only states that determining a valid schedule for a given server set on m processors is equivalent to finding a valid schedule for the transformed set on $n - m$ virtual processors. Nonetheless, this theorem raises an interesting issue. Indeed, dealing with $n-m$ virtual processors instead of m can be advantageous if $n-m < m$. In order to illustrate this observation, consider our example set of three servers with utilization equal to $2/3$. Instead of searching for a valid schedule on two processors, one can focus on the schedule of the dual servers on just one virtual processor, a problem whose solution is well known.

In order to guarantee that dealing with dual servers is indeed advantageous, the PACK operation plays a central role.

4.3 PACK OPERATION

As seen in the previous section, the DUAL operation is a powerful mechanism to reduce the number of processors but it only works properly if $n-m < m$ where n and m are the number of tasks and processors respectively. However, this is not the case for general task sets.

Consider for instance a simple set $\mathcal{T}$ of 5 tasks, all with rate $2/5$. Here, $n = 5$, $m = \rho(\mathcal{T}) = 2$ and $n - m = 3 > 2$. In such a case, directly applying duality does not simplify the scheduling problem. Indeed, the dual $\mathcal{T}^*$ of $\mathcal{T}$ is comprised of 5 task all of which with rate $1 - 2/5 = 3/5$. Hence, the accumulated rate of $\mathcal{T}^*$ equals 3, which is greater than the initial number of processors needed to schedule $\mathcal{T}$. Hence, the DUAL operation directly applied to $\mathcal{T}$ leads to a more complex problem than the primal one.

As can be deduced from this simple example, whenever $n - m \geqslant m$, one needs to reduce the number of tasks/servers to be scheduled, aggregating them into servers. This is achieved by the PACK operation that we properly define in this section.

Definition 4.3.1 (Packing). *Let Γ be a set of servers. A partition $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ of Γ is a* packing *of Γ if $\rho(\Gamma_i) \leqslant 1$ for all i and $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ for all $i \neq j$. An algorithm A is a* packing algorithm *if it partitions any set of servers into a packing. In such a case, we denote the packing of Γ produced by A as $\pi_A[\Gamma]$.*

An illustrative example is given by Figure 4.2, where the three sets Γ_1, Γ_2 and Γ_3 show a packing of the set Γ of 7 servers.

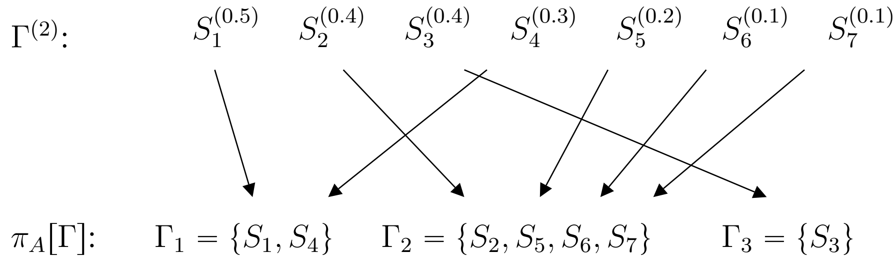


Figure 4.2. Packing algorithm applied to $\Gamma = \{S_1, S_2, \ldots, S_7\}$, resulting in the partition $\pi_A[\Gamma]$ of Γ into three subsets Γ_1, Γ_2 and Γ_3. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$.

Theorem 4.3.1. *The first-fit, worst-fit and best-fit bin-packing algorithms are packing algorithms.*

Proof. At any step of these three algorithms, a new bin can only be created if the current task to be allocated does not fit in any of the existing partially filled bins. Now suppose that $\rho(\Gamma_i) + \rho(\Gamma_j) \leqslant 1$ for some two bins, where Γ_j was created after Γ_i. Then the first item τ placed in Γ_j must have $\rho(\tau) \leqslant \rho(\Gamma_j) \leqslant 1 - \rho(\Gamma_i)$. That is, τ fits in bin Γ_i, contradicting the need to create Γ_j for it. Therefore $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ must hold for any pair of bins. □

Lemma 4.3.1. *Let Γ be a set of servers and A a packing algorithm. Then, there may exist at most one set $\Gamma_i \in \pi_A[\Gamma]$ such that $\rho(\Gamma_i) \leqslant 1/2$.*

Proof. Suppose that there exist two distinct sets Γ_i and Γ_j in $\pi_A[\Gamma]$ such that $\rho(\Gamma_i) \leqslant 1/2$ and $\rho(\Gamma_j) \leqslant 1/2$. Then, $\rho(\Gamma_i) + \rho(\Gamma_j) \leqslant 1$, contradicting the Definition 4.3.1 of $\pi_A[\Gamma]$. □

Hereafter, we assume that A is a packing algorithm. Since $\pi_A[\Gamma]$ is a partition of Γ, the relation $\mathcal{R}_A$ between two servers S and S' in Γ defined by

$$S \ \mathcal{R}_A \ S' \iff \exists \Gamma_i \in \pi_A[\Gamma], \ S \in \Gamma_i \text{ and } S' \in \Gamma_i$$

is an equivalence relation whose equivalence class are the elements in $\pi_A[\Gamma]$ (BOURBAKI, 1968). Also, we have $\pi_A[\Gamma] = \Gamma/\mathcal{R}_A$, where $\Gamma/\mathcal{R}_A$ is the quotient set of Γ by relation $\mathcal{R}_A$. We introduce p_A the canonical mapping of Γ onto $\pi_A[\Gamma]$, which maps a server in Γ to its equivalence class in $\pi_A[\Gamma]$, *i.e.*, $p_A(S) = p_A(S')$ if and only if $S \ \mathcal{R}_A \ S'$. Also, if $\Gamma_i \in \pi_A[\Gamma]$ and $S \in \Gamma_i$, then $p_A(S) = \Gamma_i$ and $\sigma_A(S) = \operatorname{ser}(\Gamma_i)$.

As stated by Lemma 4.3.1, the subsets of servers in $\pi_A[\Gamma]$ have all but possibly one accumulated rate close or equal to one. Since those aggregating subsets also need to be scheduled by a server, we define the PACK operation as the mapping which associates S in Γ_i to its aggregating server $\operatorname{ser}(\Gamma_i)$.

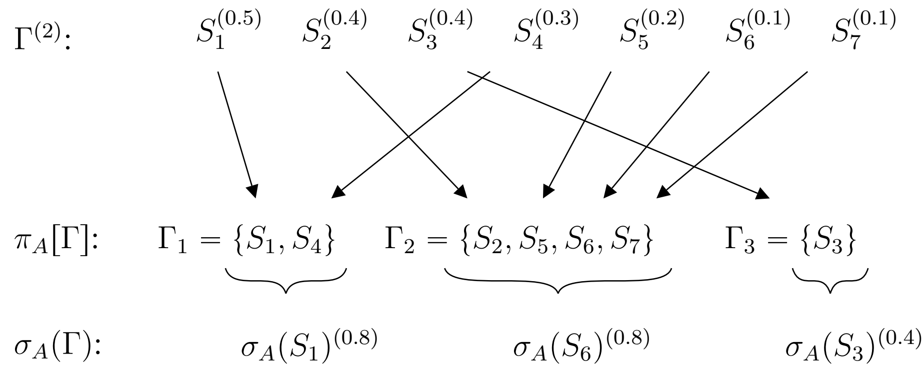


Figure 4.3. Packing and PACK operation applied to $\Gamma = \{S_1,\ S_2,\ \ldots, S_7\}$, resulting in three assigned server $\text{ser}(\Gamma_1) = \sigma_A(S_1)$, $\text{ser}(\Gamma_2) = \sigma_A(S_6)$ and $\text{ser}(\Gamma_3) = \sigma_A(S_3)$. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$.

Definition 4.3.2 (PACK operation)**.** *Let* Γ *be a set of servers,* A *a packing algorithm, and* $\pi_A[\Gamma]$ *the resultant packing. For each* $\Gamma_i \in \pi_A[\Gamma]$*, we assign it a dedicated server* $\text{ser}(\Gamma_i)$*. The* PACK operation σ_A *is the mapping from* Γ *onto* $\text{ser}(\pi_A[\Gamma])$ *defined by* $\sigma_A = \text{ser} \circ p_A$*, where* p_A *is the canonical mapping from* Γ *onto* $\pi_A[\Gamma]$ *and* $\text{ser}(\pi_A[\Gamma]) = \{\text{ser}(\Gamma_i), \Gamma_i \in \pi_A[\Gamma]\}$*. Hence,* σ_A *associates a server* S *in* Γ *with the server* $\sigma_A(S)$ *in* $\text{ser}(\pi_A[\Gamma])$ *responsible for scheduling* $p_A(S)$*.*

The mapping σ_A is *compatible* with $\mathcal{R}_A$, in the sense that it is constant within each equivalence class of $\mathcal{R}_A$ (BOURBAKI, 1968). That is, if S and S' are packed in the same subset Γ_i by packing algorithm A, then $\sigma_A(S) = \sigma_A(S')$. Note that this latter property also implies that $\sigma_A(\Gamma_i) = \{\sigma_A(S)\}$ for all S in Γ_i.

As previously stated, we use the notation $\sigma_A(\Gamma)$ as an equivalent for $\{\sigma_A(S),\ S \in \Gamma\}$. Thus, $\sigma_A(\Gamma) = \{\text{ser}(\Gamma_i),\ \Gamma_i \in \pi_A[\Gamma]\}$. In other words, $\sigma_A(\Gamma)$ is the set of servers each of which is in charge of scheduling the elements of its equivalence class in partition $\pi_A[\Gamma]$.

Rows 2 and 3 of Figure 4.3 show that $\sigma_A(S_1) = \text{ser}(\Gamma_1)$, $\sigma(S_6) = \text{ser}(\Gamma_2)$ and $\sigma_A(S_3) = \text{ser}(\Gamma_3)$. Note for instance that the single server $\sigma_A(S_6)$ is responsible for scheduling all the servers in Γ_2 with which S_6 is aggregated by packing algorithm A.

Definition 4.3.3 (Packed Server Set)**.** *A set of servers* Γ *is* packed *if it is a singleton, or if* $|\Gamma| \geqslant 2$ *and for any two distinct servers* S *and* S' *in* Γ*,* $\rho(S) + \rho(S') > 1$ *and* $\text{cli}(S) \cap \text{cli}(S') = \{\}$*.*

By this definition, the packing of a packed server set Γ is the collection of singleton sets $\pi_A[\Gamma] = \{\{S\}\}_{S\in\Gamma}$.

Since most of the results presented in this dissertation just require that the underlying bin-packing algorithm is a packing algorithm, as stated in Definition 4.3.1, we simply denote hereafter $\pi[\Gamma]$ a packing of Γ and σ the associated PACK operation when no confusion is introduced doing so.

4.4 REDUCE OPERATION

We now compose the DUAL and PACK operations, as defined in 4.2.3 and 4.3.2 respectively, into the REDUCE operation. As will be shown, a sequence of reductions transforms a multiprocessor scheduling problem to a collection of uniprocessor scheduling problems. Hence, the REDUCE operation can be viewed as a cornerstone of the RUN algorithm presented in this dissertation.

In order to see the effectiveness of the composition of the PACK and DUAL operations, we first establish a lemma which characterizes the convergence of this composition in terms of server set cardinality.

Lemma 4.4.1. *Let Γ be a packed set of servers, and let $\varphi(\Gamma)$ be the dual set of Γ. Suppose we apply a* PACK *operation σ to $\varphi(\Gamma)$. Then*

$$|\,\sigma \circ \varphi(\Gamma)\,| \leqslant \left\lceil \frac{|\Gamma|+1}{2} \right\rceil .$$

Proof. Let $n = |\Gamma|$. Since Γ is packed, there is at most one server S in Γ such that $\rho(S) \leqslant 1/2$ (by Lemma 4.3.1). This implies that at least $n-1$ servers in $\varphi(\Gamma)$ have rates less than $1/2$. When these $n-1$ dual servers are packed, they will be, at a minimum, paired off. Thus, π will pack $\varphi(\Gamma)$ into at most $\lceil (n-1)/2 \rceil + 1$ subsets. Hence,

$$|\sigma \circ \varphi(\Gamma)| \leqslant \left\lceil \frac{n+1}{2} \right\rceil .$$

□

Thus, packing the dual of a packed set reduces the number of servers by about half. Since we will use this pair of operations repeatedly, we define a REDUCE operation to be their composition.

Definition 4.4.1 (REDUCE Operation)**.** *Given a set of servers Γ and a packing algorithm A, a* REDUCE operation *on a server S in Γ, denoted $\psi(S)$, is the composition of the* DUAL *operation φ with the* PACK *operation σ associated with A,* i.e., $\psi = \varphi \circ \sigma$.

Figure 4.4 illustrates the steps of the REDUCE operation ψ. As we intend to apply REDUCE repeatedly until we are left with only one or more unit servers, we now define a *reduction sequence*.

Definition 4.4.2 (Reduction Level/Sequence)**.** *Let $i \geqslant 1$ be an integer, Γ a set of servers, and S a server in Γ. The operator ψ^i is recursively defined by $\psi^0(S) = S$ and $\psi^i(S) = \psi \circ \psi^{i-1}(S)$. $\{\psi^i\}_i$ is a* reduction sequence*, and the server system $\psi^i(\Gamma)$ is said to be at* reduction level i.

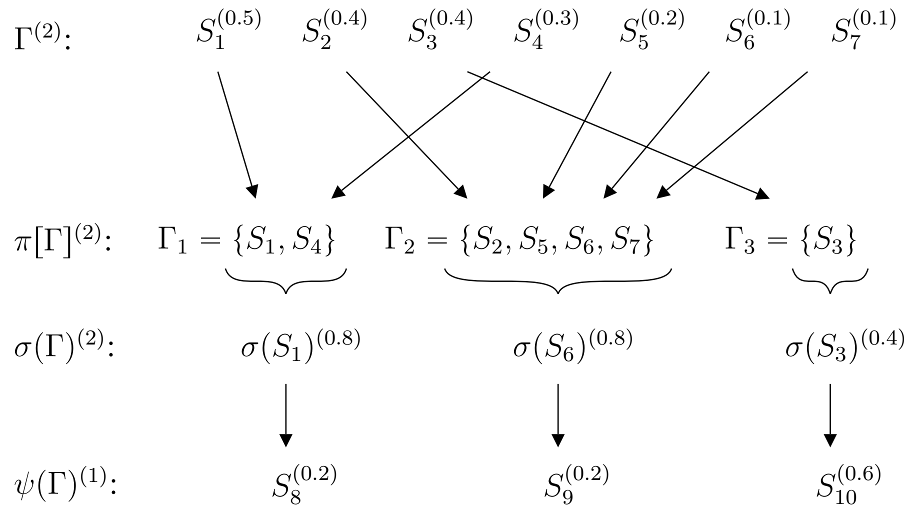


Figure 4.4. Packing, PACK operation, and duality applied to $\Gamma = \{S_1, S_2, \ldots, S_7\}$, resulting in a reduction to a unit set of three servers $\{S_8, S_9, S_{10}\}$ with $S_8 = \varphi \circ \sigma(S_1)$, $S_9 = \varphi \circ \sigma(S_6)$, $S_{10} = \varphi \circ \sigma(S_3)$. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$.

According to Lemma 4.4.1, the action of the DUAL operation applied to a packed set of servers allows for the generation of a set of servers whose accumulated utilization is less than the utilization of the original packed set.

For example, consider the reduction example illustrated in Figure 4.4. The three sets Γ, $\pi[\Gamma]$ and $\sigma(\Gamma)$ all have identical accumulated rate equal to 2 while $\psi(\Gamma) = \varphi \circ \sigma(\Gamma)$ has accumulated rate 1. As a consequence, $\psi(\Gamma)$ can be packed into a single unit server. We call such a unit server a *top-level* server.

In general, Theorem 4.4.1 states that a reduction sequence on a server set Γ with $\rho(\Gamma) = m$ eventually leads to a collection of top-level unit servers for some ad-hoc number of iterations of the REDUCE operation. Note that each of this top-level server can arise at different reduction level.

As illustration, Table 4.1 presents a simple reduction sequence applied to a primal set of 10 servers (or tasks) which is transformed into a unit server via two REDUCE operations and a final PACK operation. As can be seen, two top-level unit servers, indicated in the table by **1→**, appear before the terminal level.

We call *proper reduction tree* those servers and set of servers that arises at each level along the reduction sequence which leads to a single top-level unit server. Blank columns in Table 4.1 separate the three proper reduction tree. Also, we call *proper subset* a set of original tasks that gives birth to a single proper reduction tree and, *proper subsystem*, a proper reduction tree of tasks together with their real and virtual assigned processors.

For instance, in the original set Γ shown in Table 4.1, the first 5 servers with rate 0.6 (red color) form a first proper subset, the three next servers with rate 0.8, 0.6 and 0.6 (blue color)

Table 4.1. Sample Reduction and Proper Subsets

<table>
<tr><th></th><th colspan="10">Server Rate</th></tr>
<tr><td>$\psi^0(\Gamma)$</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.8</td><td>0.6</td><td>0.6</td><td>0.5</td><td>0.5</td></tr>
<tr><td>$\sigma(\psi^0(\Gamma))$</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.6</td><td>0.8</td><td>0.6</td><td>0.6</td><td colspan="2">**1**→</td></tr>
<tr><td>$\psi^1(\Gamma)$</td><td>0.4</td><td>0.4</td><td>0.4</td><td>0.4</td><td>0.4</td><td>0.2</td><td>0.4</td><td>0.4</td><td colspan="2">0</td></tr>
<tr><td>$\sigma(\psi^1(\Gamma))$</td><td colspan="2">0.8</td><td colspan="2">0.8</td><td>0.4</td><td colspan="5">**1**→</td></tr>
<tr><td>$\psi^2(\Gamma)$</td><td colspan="2">0.2</td><td colspan="2">0.2</td><td>0.6</td><td colspan="5">0</td></tr>
<tr><td>$\sigma(\psi^2(\Gamma))$</td><td colspan="10">**1**</td></tr>
</table>

form a second proper subset and the two last servers with rates 0.5 and 0.5 (green color) form a third proper subset.

Note that separating proper subsystems is natural since the scheduling problem if first solved using proper reduction tree, as will be shown in Chapter 5. Moreover, separating proper subsystems yields more efficient scheduling because tasks in one subsystem do not impose events on or migrate to other subsystems.

Also, observe that the dual of a unit server is a null server, which is packed along a reduction sequence, into another server in the next step. This explains that the two "0" that appears in Table 4.1 disappear after the next packing step. Also unnecessary, we adopt this "0" absorption procedure, cleverly proposed by Greg Levin (REGNIER et al., 2011), for the sake of concision of the proof of Theorem 4.4.1.

However, from the implementation point of view, it may be better to consider that a unit server, together with its associated proper subsystem, is assigned to execute on a separated set of virtual and real processors. Using such a partitioning approach allows for isolating the proper reduction tree associated to a top-level unit server and scheduling the corresponding proper task subset independently from the remaining tasks in the system.

We now provide two intermediate results which will be used to establish Theorem 4.4.1.

The following lemma establishes that the accumulated rate of a set of servers Γ is not greater than the number of servers assigned to schedule Γ by a PACK operation.

Lemma 4.4.2. *Let Γ be a set of servers, and let $\sigma(\Gamma)$ be the set of servers assigned to the packing $\pi[\Gamma]$ of some* PACK *operation on Γ. Then $\rho(\Gamma) \leqslant |\sigma(\Gamma)|$. Further, if not all servers in $\sigma(\Gamma)$ are unit servers, then $\rho(\Gamma) < |\sigma(\Gamma)|$*

Proof. A PACK operation does not change the utilization of servers in Γ. As a consequence, $\rho(\Gamma) = \rho(\sigma(\Gamma))$.

To show the inequality, recall from Definition 3.2.2 that $\rho(\sigma(\Gamma)) = \sum_{S \in \sigma(\Gamma)} \rho(S)$. Also,

since $\rho(S) \leqslant 1$ for all servers S in $\sigma(\Gamma)$ and

$$\sum_{S \in \sigma(\Gamma)} 1 = |\sigma(\Gamma)|,$$

it follows that $\rho(\sigma(\Gamma)) \leqslant |\sigma(\Gamma)|$. Moreover, if not all servers in $\sigma(\Gamma)$ are unit server, then there exists at least one server S in $\sigma(\Gamma)$ such that $\rho(S) < 1$ and the inequality is strict. □

Lemma 4.4.3. *Let Γ be a packed set of servers, not all of which are unit servers. If $\rho(\Gamma)$ is a positive integer, then $|\Gamma| \geqslant 3$.*

Proof. If $\Gamma = \{S_1\}$ and S_1 is not a unit server, then $\rho(\Gamma) < 1$, not a positive integer. If $\Gamma = \{S_1, S_2\}$ is a packed set, then $\rho(\Gamma) = \rho(S_1) + \rho(S_2) > 1$; but $\rho(\Gamma)$ is not 2 unless S_1 and S_2 are both unit servers. Thus $|\Gamma|$ is not 1 or 2. □

Theorem 4.4.1 (Reduction Convergence)**.** *Let Γ be a set of servers where $\rho(\Gamma)$ is a positive integer. Then for some $p \geqslant 0$, $\sigma(\psi^p(\Gamma))$ is a set of unit servers.*

Proof. We prove the theorem by finite induction on the number k of reduction level.

Let $\Gamma^{(k)} = \psi^k(\Gamma)$ and suppose that $\rho(\Gamma^k)$ is a positive integer. If $\sigma(\Gamma^k)$ is a set of unit servers, then $p = k$ and the induction is finished.

Otherwise, according to Lemma 4.4.3, $|\sigma(\Gamma^k)| \geqslant 3$. Next, consider $\Gamma^{k+1} = \psi^{k+1}(\Gamma)$ and observe that

$$\begin{aligned}
\sigma(\Gamma^{k+1}) &= \sigma \circ \psi^{k+1}(\Gamma) \\
&= \sigma \circ \psi \circ \psi^k(\Gamma) \\
&= \sigma \circ \varphi \circ \sigma(\Gamma^{(k)}) \\
&= (\sigma \circ \varphi)(\sigma(\Gamma^{(k)}))
\end{aligned}$$

Since $\sigma(\Gamma^k)$ is a packed set of servers, Lemma 4.4.1 tells us that

$$\sigma(\Gamma^{k+1}) \leqslant \left\lceil \frac{|\sigma(\Gamma^k)| + 1}{2} \right\rceil.$$

Since $|\sigma(\Gamma^k)| \geqslant 3$ and $\lceil (x+1)/2 \rceil < x$ for $x \geqslant 3$, we deduce that

$$|\sigma(\Gamma^{k+1})| < |\sigma(\Gamma^k)|$$

Now, recall we assume that $\rho(\sigma(\Gamma^k))$ is a positive integer. Moreover, since $\sigma(\Gamma^k)$ are not all unit servers, it follows from Lemma 4.4.2 that $\rho(\sigma(\Gamma^k)) < |\sigma(\Gamma^k)|$.

Further, Theorem 4.2.1 implies that $\rho(\varphi(\sigma(\Gamma^k)))$ is also a positive integer; as is

	First Packing				
$\psi^0(\Gamma)$	0.4	0.4	0.2	0.2	0.8
$\sigma(\psi^0(\Gamma))$	0.8		0.4		0.8
$\psi^1(\Gamma)$	0.2		0.6		0.2
$\sigma(\psi^1(\Gamma))$	1				

Second Packing				
0.4	0.4	0.2	0.8	0.2
1			1	

Table 4.2. Reduction Example with Different Outcomes.

$\rho(\sigma(\Gamma^{k+1}))$, since packing does not change total rate. Thus $\sigma(\Gamma^{k+1})$ satisfies the same conditions as $\sigma(\Gamma^k)$, but contains fewer servers.

Finally, starting with the packed set $\sigma(\Gamma^0) = \sigma(\Gamma)$, each iteration of $\sigma \circ \varphi$ either produces a set of unit servers or a smaller set with positive integer rate. This iteration can only occur a finite number of times, and once $|\sigma(\Gamma^k)| < 3$, Lemma 4.4.3 tells us that $\sigma(\Gamma^k)$ must be a set of unit servers, and thus, $p = k$. □

Theorem 4.4.1 states that a reduction sequence on any set of servers eventually produces a set of unit servers. It is important to note that some unit servers can be produced at any step of the reduction sequence before p. However, as pointed by Greg Levin in a personal communication, this is not an issue, since the dual of a unit server is a zero-utilization server which is "absorbed" at the following step of the reduction sequence, being packed together with any other non-zero utilization server.

Also, it is worth noticing that the ψ operator is a mapping whose outcome is dependent on the packing scheme used.

As an example, Table 4.2 shows two packings of the same set of servers by two different packing algorithms. One produces one unit server after one reduction level and the other produces two unit servers with no reductions.

However, while some packings may be "better" than others (*i.e.,* lead to a more efficient schedule in terms of preemption and migration), Theorem 4.4.1 implicitly proves that all PACK operations "work"; they all lead to a correct reduction to *some* set of unit servers.

4.5 CONCLUSION

In this chapter, we have precisely defined the DUAL and PACK operations and their composition into the REDUCE operation. We have shown that carefully using this operator allows one to reduce a general task system with integer utilization greater than or equal to two to a system of unit servers which can be efficiently scheduled on uniprocessor systems.

However, one must observe that the REDUCE operation applied to a set of task does not tell us anything about the on-line scheduling of that tasks. As a matter of fact, the reduction sequence associated to a given packing can be determined off-line.

Thereafter, one must combine the Dual Scheduling Equivalence and the server scheduling policy, assumed to be EDF in this dissertation, in order to deduce from the uniprocessor schedules of the reduced server system an on-line schedule of the primal set of tasks.

In the next chapter, we focus on proper set of tasks for which a reduction sequence produces a single unit server. For such sets, we show how one can use the associated proper reduction tree to generate an on-line schedule of the primal tasks.

In general, the RUN scheduling algorithm can be used to schedule many proper subsystems, since each of these subsystems are independent and can be scheduled in an isolated manner.

Chapter 5

An adequate sequence of REDUCE *operations transforms a general multiprocessor primal task system into a set of one or more unit servers which can be schedule on virtual uniprocessor systems. Then, the* on-line *schedule of the primal multiprocessor task system can be deduced from the (virtual) schedules of the derived uniprocessor systems. This is performed by combining the Dual Scheduling Equivalence and the EDF server scheduling policy.*

REDUCTION TO UNIPROCESSOR (RUN)

5.1 INTRODUCTION

In Chapters 1, 3 and 4, we have described our real-time system model, namely the fixed-rate task model as defined in Definition 3.2.1 for identical processors. We also have introduced new abstractions, namely the EDF server abstraction as defined in Definition 3.4.1 and the operations DUAL, PACK and REDUCE as defined in Definitions 4.2.3, 4.3.2 and 4.4.1, respectively. Hence, we can now describe the reduction to uniprocessor (RUN) scheduling algorithm which is the main contribution of this dissertation.

RUN is based on the original notion of *partitioned proportionate fairness* (PP-Fair), as introduced in Section 1.7. PP-Fairness imposes a less restrictive set of constraints when compared to those present in the notion of proportionate fairness (Pfair) from (BARUAH et al., 1993), which has been used in previous optimal solutions for the problem of scheduling periodic real-time tasks on multiprocessors up to now. Indeed, to the best of our knowledge, RUN is the first optimal multiprocessor scheduling algorithm for periodic real-time task systems not based on proportionate fairness.

Recall from Chapter 4 that the REDUCE operation is the composition of the DUAL and PACK operations. First, the PACK operation, precisely defined in Section 4.3 of Chapter 4, transforms a set of low-rate tasks compared to one into a set of high-rate servers compared to one. Indeed, those sets of tasks whose rates sum up to no more than one are packed into servers, reducing the number of tasks and producing a packed set of high-rate servers needed to apply the Dual Scheduling Equivalence (DSE) rule. Second, the DUAL operation, precisely defined in

Section 4.2 of Chapter 4, transforms a server S into its dual server S^*, whose execution time represents the idle time of S *i.e.,* $\rho(S^*) = 1 - \rho(S)$. Then, given a schedule of the dual system of a primal system of high-rate servers, the DSE rule allows for deducing a valid schedule for the primal set of servers.

For some particular task system, this sequence of operations may need to be iterated, as first pointed out by Ernesto Massa in (REGNIER et al., 2011), in order to obtain a set of unit servers, each of which is feasible on a uniprocessor system. Hence, carefully composing the DUAL and PACK operators into the REDUCE operator allows one to achieve partitioned proportional fairness by reduction of any general periodic task system with integer accumulated rate greater than or equal to two to a system of unit servers.

However, one must observe that the REDUCE operation applied to a primal set of tasks does not tell us anything about the *on-line* scheduling of those tasks. As a matter of fact, the reduction sequence associated to a given packing can be carried out *off-line*. As an interesting consequence, given some particular goal, one can look for a packing with nice properties according to this specific goal in off-line/during design time.

In this chapter, we show how the *on-line* schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems. This is performed by combining the Dual Scheduling Equivalence and the EDF server scheduling policy in order to deduce from the uniprocessor schedules of the reduced server system an on-line schedule of the primal set of tasks. This procedure leads us to the detailed presentation of RUN, the multiprocessor *on-line* and *optimal* scheduling algorithm for periodic task systems proposed in this dissertation.

For the sake of simplicity of this chapter, we focus on proper set of tasks for which a reduction sequence produces a single unit server. For such sets, we show how one can use the associated proper reduction tree to generate an on-line schedule of the primal tasks. This simplification does not cause any loss of generality since, if more than one proper subsystem are needed for the reduction of a general primal task system into proper subsystems, then each of these proper subsystems can be scheduled in an isolated and independent way by the RUN scheduling algorithm.

Structure of the chapter

Section 5.2 describes the RUN scheduling procedure and the associated on-line scheduling rules while Section 5.3 depicts an alternative interpretation of the RUN tree, which may be helpful for future works, as for example, a RUN based solution for the sporadic task model.

Table 5.1. Reduction example of $\Gamma = \{S_1\!:\![2/5, 5\mathbb{N}],\ S_2\!:\![2/5, 10\mathbb{N}],\ S_3\!:\![2/5, 15\mathbb{N}],\ S_4\!:\![2/5, 10\mathbb{N}],\ S_5\!:\![2/5, 5\mathbb{N}]\}$

	Server Rate				
Γ	0.4	0.4	0.4	0.4	0.4
$\sigma(\Gamma)$	0.8		0.8		0.4
$\psi(\Gamma)$	0.2		0.2		0.6
$\sigma(\psi(\Gamma))$	**1**				

5.2 RUN SCHEDULING

Now that we know how to transform a primal task set $\mathcal{T}$ with integer accumulated utilization greater than or equal to two into one or more unit servers schedulable on virtual uniprocessor systems, we show how to use this transformation to deduce a schedule for $\mathcal{T}$.

The basic idea here is to use the dual schedules to find the primal schedules and use EDF servers to schedule client servers and tasks. Theorem 4.4.1 says that a reduction sequence produces a collection of one or more unit servers. As shown in Table 4.1, the original task set may be partitioned into the proper subsets represented by these unit servers, which may be scheduled independently. In this section, we assume that $\mathcal{T}$ is a proper subset, *i.e.,* that it is handled by a single top-level unit server at the terminal reduction level.

The scheduling process is illustrated by inverting the reduction tables from the previous section and creating a *scheduling reduction tree*, or *simply RUN tree*, whose nodes are the servers generated by iterations of the PACK and DUAL operations. The unit server becomes the root server, which represents the top-level virtual uniprocessor system. The root's children are the top-level unit server's clients, which are scheduled by EDF.

In order to clarify our discussion, let us consider the simple 5-server proper set example given in Table 5.1 which requires exactly one reduction to be reduced to a unit server.

Figure 5.1 shows a packing of Γ and the associated assigned servers $\sigma(S_1) = \sigma(S_2) = S_6$, $\sigma(S_3) = \sigma(S_4) = S_7$ and $\sigma(S_5) = S_8$. Next, Figure 5.2 illustrates the complete RUN tree used to reduce Γ to a single unit server. Finally, an example of schedule of Γ is shown in Figure 5.3.

In Figure 5.4, which shows the scheduling decision based on the RUN tree of $\Gamma = \{S_1, \ldots, S_5\}$ from Table 5.1, at time $t = 4$ for the schedule shown in Figure 5.3, the servers executing at each level are red colored. The schedule for Γ (the leaves of the tree) is obtained by propagating the schedule down the tree using Rules 3.4.1 (schedule clients with EDF) and 4.2.1 (use Σ^* to find Σ). Hence, at time 4, the top-level unit server schedules S_7^* since neither S_6^* nor S_8^*

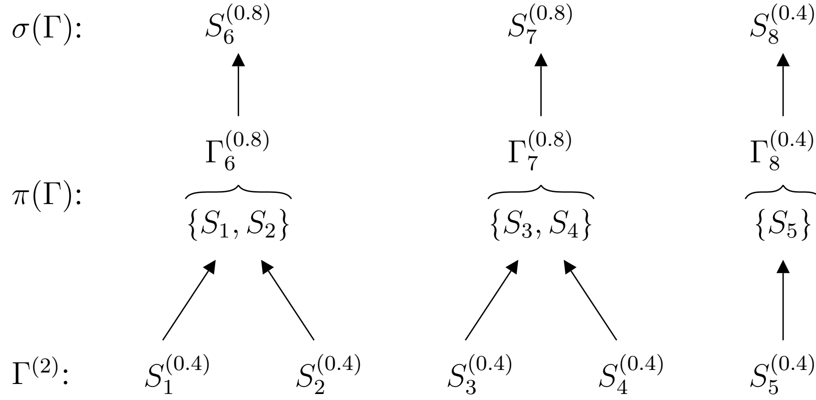


Figure 5.1. Packing of $\Gamma = \{S_1, \ldots, S_5\}$ as defined in Table 5.1. Notation S_i^μ means that $\rho(S_i) = \mu$.

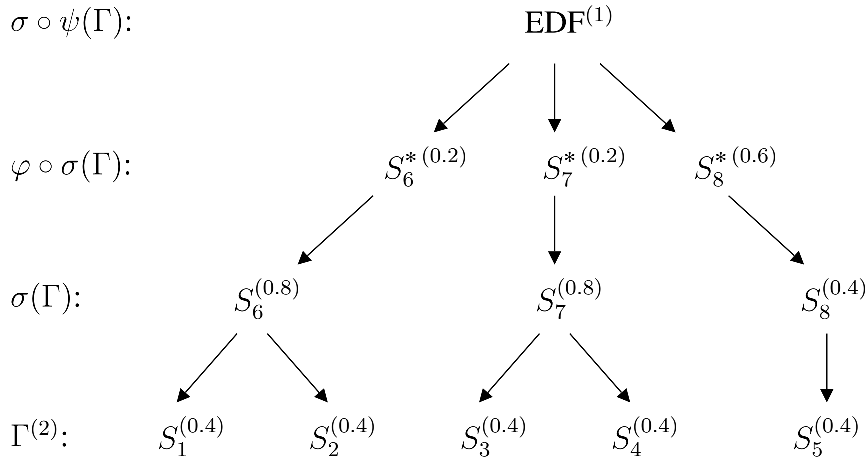


Figure 5.2. RUN tree used to schedule $\mathcal{T} = \{S_1, \ldots, S_5\}$ from Table 5.1 by Rules 5.2.1 and 5.2.2 at scheduling instant 4. Notation S_i^μ means that $\rho(S_i) = \mu$.

has jobs ready to execute. But, if S_7^* executes in Σ^*, then S_7 does not execute in Σ. In turn, this implies that S_6 and S_8 execute in Σ. Yet, the first job of S_1 is completed by time 4. Hence, S_6 schedules S_2 at time 4. On the other hand, the first job of S_5, which has the earliest deadline 5 at time 4 is not yet completed by time 4. Hence, S_8 schedules S_5 at time 4 and this completes the scheduling decision to be taken at time 4.

As regards each server node in the RUN tree, the on-line scheduling rules may be restated as follows.

Rule 5.2.1 (EDF Server)**.** *If a packed server is executing (circled and red colored), execute the child node with the earliest deadline among those children with work remaining; if a packed server is not executing (not circled and black colored), execute none of its children.*

Rule 5.2.2 (Dual Server)**.** *Execute (circled and red colored) the child (packed server) of a dual server if and only if the dual server is* not *executing (not circled and black colored).*

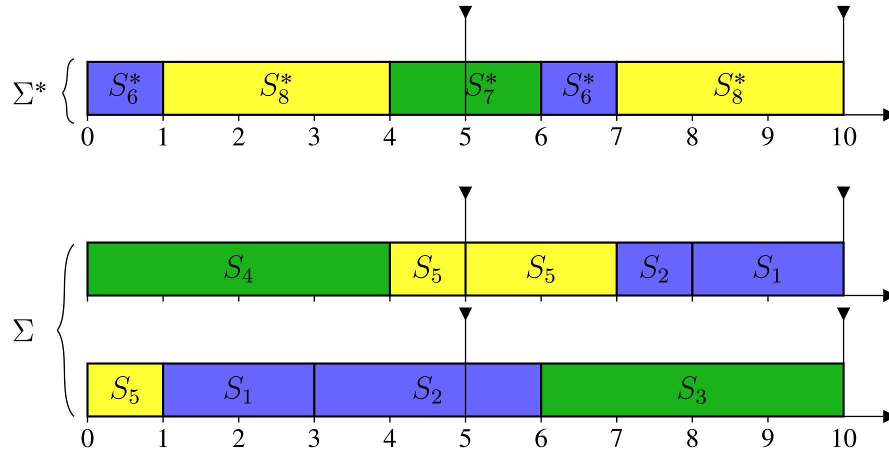


Figure 5.3. RUN schedule example with $\Gamma = \{S_1, S_2, S_3, S_4, S_5\}$ with $S_1 = \text{ser}(2/5, 5\mathbb{N}^*)$, $S_2 = \text{ser}(2/5, 10\mathbb{N}^*)$, $S_3 = \text{ser}(2/5, 15\mathbb{N}^*)$, $S_4 = \text{ser}(2/5, 10\mathbb{N}^*)$, $S_5 = \text{ser}(2/5, 5\mathbb{N}^*)$. Σ is the schedule of Γ on 2 physical processors and Σ^* is the schedule of $\psi(\Gamma) = \{S_6, S_7, S_8\}$ on 1 virtual processor with $S_6 = \text{ser}(\{S_1, S_2\})$, $S_7 = \text{ser}(\{S_3, S_4\})$ and $S_8 = \text{ser}(\{S_5\})$

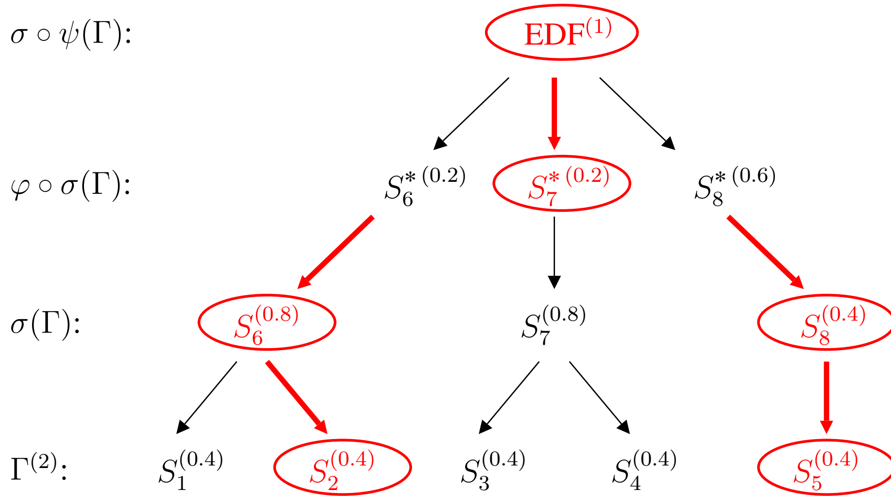


Figure 5.4. Run tree and scheduling rules applied to schedule $\Gamma = \{S_1, \ldots, S_5\}$ from Table 5.1 by Rules 5.2.1 and 5.2.2 at scheduling instant 4. The notation S_i^{μ} means that $\rho(S_i) = \mu$.

We now give a slightly more complex example of task set as defined in Table 5.2 which requires two reductions to be reduced to a unit server. Observe that this is the first example given in Table 4.1 and that the first reduction of this task set leads to our previous example as given in Table 5.1.

Figure 5.5 shows the RUN tree for this new task set. To the five tasks with rate 0.6, we assign the deadline sets $5\mathbb{N}^*$, $10\mathbb{N}^*$, $15\mathbb{N}^*$, $10\mathbb{N}^*$, and $5\mathbb{N}^*$, respectively. Rule 5.2.1 is seen in the tree edges $\{e_1, e_4, e_5, e_9, e_{10}, e_{11}\}$. Rule 5.2.2 is seen in the tree edges $\{e_2, e_3, e_6, e_7, e_8\}$. With these two simple rules, at any time t, we can determine which tasks in $\mathcal{T}$ should be executing by circling the root and propagating circles down the tree into the leaves. In practice, we only

Table 5.2. Reduction example of $\Gamma = \{S_1:[3/5, 5\mathbb{N}],\ S_2:[3/5, 10\mathbb{N}],\ S_3:[3/5, 15\mathbb{N}],\ S_4:[3/5, 10\mathbb{N}],\ S_5:[3/5, 5\mathbb{N}]\}$

	Server Rate				
Γ	0.6	0.6	0.6	0.6	0.6
$\sigma(\Gamma)$	0.6	0.6	0.6	0.6	0.6
$\psi(\Gamma)$	0.4	0.4	0.4	0.4	0.4
$\sigma(\psi(\Gamma))$	0.8		0.8		0.4
$\psi^2(\Gamma)$	0.2		0.2		0.6
$\sigma(\psi^2(\Gamma))$	**1**				

Algorithm 5.1: Outline of the RUN algorithm

1 **I. OFF-LINE**;
2 A. Generate a reduction sequence for $\mathcal{T}$;
3 B. Invert the sequence to form a RUN tree;
4 C. For each proper subsystem $\mathcal{T}$' of $\mathcal{T}$;
5 Define the client/server at each virtual level;
6 **II. ON-LINE**;
7 Upon a scheduling event: ;
8 A. If the event is a job release event at level 0 ;
9 1. Update deadline sets of servers on path up to root;
10 2. Create jobs for each of these servers accordingly;
11 B. Apply Rules 1 & 2 to schedule jobs from root to leaves, determining the m jobs to schedule at level 0;
12 C. Assign the m chosen jobs to processors, according to some task-to-processor assignment scheme;

need to execute the rules when some subsystem's EDF scheduler generates a scheduling event (*i.e.*, WORK COMPLETE or JOB RELEASE). Figure 5.5 shows the scheduling decision process at $t = 4$, and Figure 5.6 shows the full schedule for all three reduction levels for ten time units.

At all level of the RUN tree, each child server, scheduled by its parent server, must keep track of its own workloads and deadlines. These deadlines and workloads are based on the own server clients of the child server. Recall that the process of setting deadlines and allocating workloads for virtual server jobs has been already detailed in Section 3.4.1. In a few words, each server node of the RUN tree which is not a task in $\mathcal{T}$ simulates the behavior of a task so that its parent node can schedule it along with its siblings in its virtual system.

The process described so far, from reducing a task set to unit servers to the scheduling of

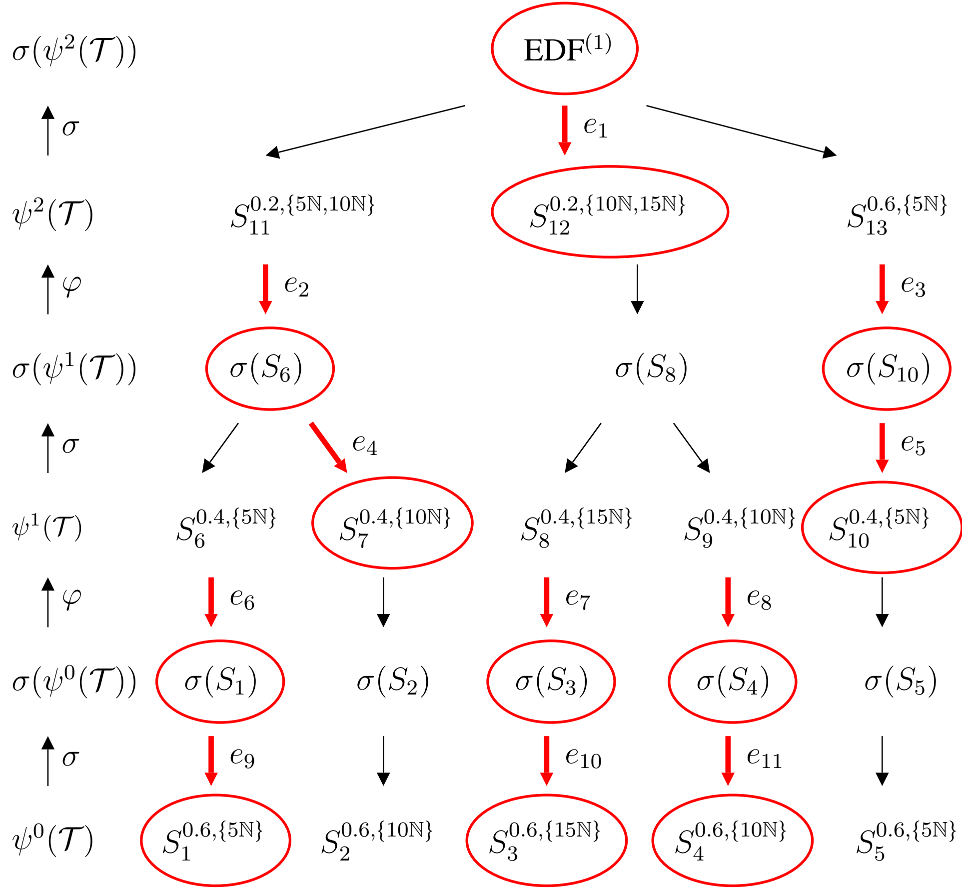


Figure 5.5. RUN tree used to schedule $\Gamma = \{S_1, \ldots, S_5\}$ from Table 4.1 by Rules 5.2.1 and 5.2.2 at scheduling instant 4. The notation $S_i^{\mu,D}$ means that $\rho(S_i) = \mu$ and $\mathrm{R}(S_i) = D$.

those tasks with EDF servers and duality, is collectively referred to as the RUN algorithm and is summarized in Algorithm 5.1. We now finish proving it is correct.

Theorem 5.2.1 (Reduction Schedule)**.** *If Γ is a proper set of tasks under the reduction sequence $\{\psi^i\}_{i \leqslant p}$, then the RUN algorithm produces a valid schedule Σ for Γ.*

Proof. Again, let $\Gamma^k = \psi^k(\Gamma)$ and $\Gamma^k_\sigma = \sigma(\Gamma^k)$ with $k < p$. Also, let Σ^k and Σ^k_σ be the schedules generated by RUN for Γ^k and Γ^k_σ, respectively.

By Definition of the PACK operation σ given in 4.3.2, Γ^k_σ is the set of servers in charge of scheduling the packing of Γ^k. Hence, $\rho(\Gamma^k) = \rho(\Gamma^k_\sigma)$. Let $\mu^k = \rho(\Gamma^k) = \rho(\Gamma^k_\sigma)$, which, as seen in the proof of Theorem 4.4.1, is always an integer.

We will work inductively on the number k of reduction level to show that schedule correctness propagates down the reduction tree, *i.e.,* that the correctness of Σ^{k+1} implies the correctness of Σ^k.

Suppose that Σ^{k+1} is a valid schedule for $\Gamma^{k+1} = \varphi(\Gamma^k_\sigma)$ on μ^{k+1} processors, where

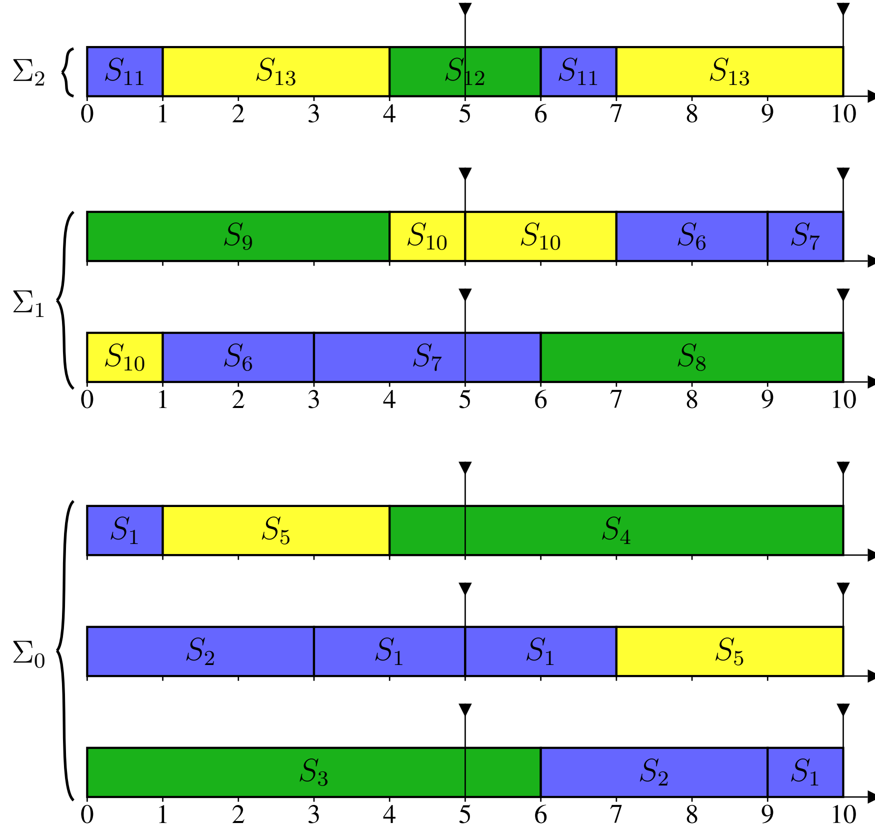


Figure 5.6. RUN schedule example with $\mathcal{T} = \{S_1, S_2, S_3, S_4, S_5\}$ with $S_1 = \text{ser}(3/5, 5\mathbb{N}^*)$, $S_2 = \text{ser}(3/5, 10\mathbb{N}^*)$, $S_3 = \text{ser}(3/5, 15\mathbb{N}^*)$, $S_4 = \text{ser}(3/5, 10\mathbb{N}^*)$, $S_5 = \text{ser}(3/5, 5\mathbb{N}^*)$. Σ_0 is the schedule of $\mathcal{T}$ on 3 physical processors. Σ_1 is the schedule of $\psi(\mathcal{T}) = \{S_6, S_7, S_8, S_9, S_{10}\}$ on 2 virtual processors, and Σ_2 is the schedule of $\psi^2(\mathcal{T}) = \{S_{11}, S_{12}, S_{13}\}$ on 1 virtual processor with $S_{11}^* = \text{ser}(\{S_6, S_7\})$, $S_{12}^* = \text{ser}(\{S_8, S_9\})$ and $S_{13}^* = \text{ser}(\{S_{10}\})$.

$k + 1 \leqslant p$. Since $k < p$, Γ_σ^k is not the terminal level set, and so must contain more than one server, as does its equal-sized dual Γ^{k+1}. Further, since Γ^{k+1} is the dual of a packed set, none of these servers can be unit servers and so $|\Gamma^{k+1}| > \mu^{k+1}$. The conditions of Theorem 4.2.1 are satisfied (where $n = |\Gamma^{k+1}|$, $m = \mu^{k+1}$, and $k > 1$), so our assumption that Σ^{k+1} is valid implies that $\Sigma_\sigma^k = (\Sigma^{k+1})^*$ is a valid schedule for Γ_σ^k on μ^k processors.

Moreover, since Γ_σ^k is a collection of aggregated servers for Γ^k, it follows from Theorem 3.4.1 that Σ^k is a valid schedule for Γ^k (*i.e.,* scheduling the servers in Γ_σ^k correctly ensures that all of their client tasks in Γ^k are also scheduled correctly). Thus the correctly of Σ^{k+1} implies the correctness of Σ^k, as desired.

Since uniprocessor EDF generates a valid schedule Σ^p for the clients of the unit server at terminal reduction level p, it follows inductively that $\Sigma = \Sigma^0$ is valid for Γ on $\rho(\Gamma)$ processors. $\square$

5.3 PARALLEL EXECUTION REQUIREMENT

An interesting way of interpreting the RUN tree was first pointed by Ernesto Massa in a personal communication.

We first introduce or clarify the notions of grandparent server and grandchild server. Considering a RUN tree, we say that S'' is a grandchild server of a server S if $S = \psi^2(S'')$. In such a case, we also say that server S is the grandparent server of S''. For instance, U_1^* is a grandparent server of $S_{2,2}$ in Figure 5.7. Also, in this figure

$$\bigcup_{1 \leqslant i \leqslant p} \bigcup_{1 \leqslant j \leqslant k_i} \{S_{i,j}\}$$

is the set of all grandchild servers of U_1^*.

Looking at the schedule represented in Figure 5.6, one can perceive that whenever a grandparent server is scheduled at virtual level Σ_2, then its two associated grandchild servers execute in parallel at real level Σ_0. In other words, a grandparent server at some even level represents the rate of parallelism that exists between its grandchild servers two levels below.

In order to formalize this interpretation of the RUN tree, we introduce some new definitions. Consider a packed set of servers Γ and its dual set Γ^*. The packing $\pi[\Gamma^*]$, as defined by Definition 4.3.1, defines a partition of Γ^* and, consequently, this partition induces a partition of Γ since φ is a bijection. Also, the elements of each set in this partition are the leaf nodes servers of a *subtree* with a single grandparent server as root.

Definition 5.3.1. *We define a RUN* subtree *of a general RUN tree as the nodes of the RUN tree comprised of a single grandparent server, referred to as the subtree* root *server, together with its child servers and grandchild servers.*

Figure 5.7 shows an example of RUN *subtree* of a general RUN tree. In this figure, grandparent server U_1^* is a root server, $\{T_i^*\}_i$ is the collection of child servers of U_1, and $\{S_{i,j}\}_{i,j}$ is the collection of grandchild servers of U_1^*. Note that, in the context of a subtree, we use the term *child server* as a synonym for a client of server U_1, as illustrated by Figure 5.7.

Definition 5.3.2 (Dual-Packed Set)**.** *Let Γ be a set of servers and $\pi[\Gamma] = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_p\}$ be the packing of Γ by a packing algorithm A. The packing of $\psi(\Gamma)$ by A defines a partition of $\pi[\Gamma]$ into a family of* dual-packed set *(of server set), denoted $\{\Omega_k\}_k$, such that for all $\Gamma_i, \Gamma_j \in \Omega_k$, if $\Gamma_i \neq \Gamma_j$ then $\psi(\mathrm{ser}(\Gamma_i)) = \psi(\mathrm{ser}(\Gamma_j))$ for all k, $1 \leqslant k \leqslant |\psi(\Gamma)|$.*

If $\Omega_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ is a dual packed set of a set of servers Γ, then for all $S_i, S_j \in \bigcup_{\Gamma_k \in \Omega_1} \Gamma_k$, $\psi^2(S_i) = \psi^2(S_j)$. In other words, all the grandchild servers in the set of servers in Ω_1 have the same grandparent server $U_1^* = \psi^2(S_i)$. Thus, $\bigcup_{\Gamma_k \in \Omega_1} \Gamma_k$ is the set of all grandchild servers of the subtree with root server U_1^*.

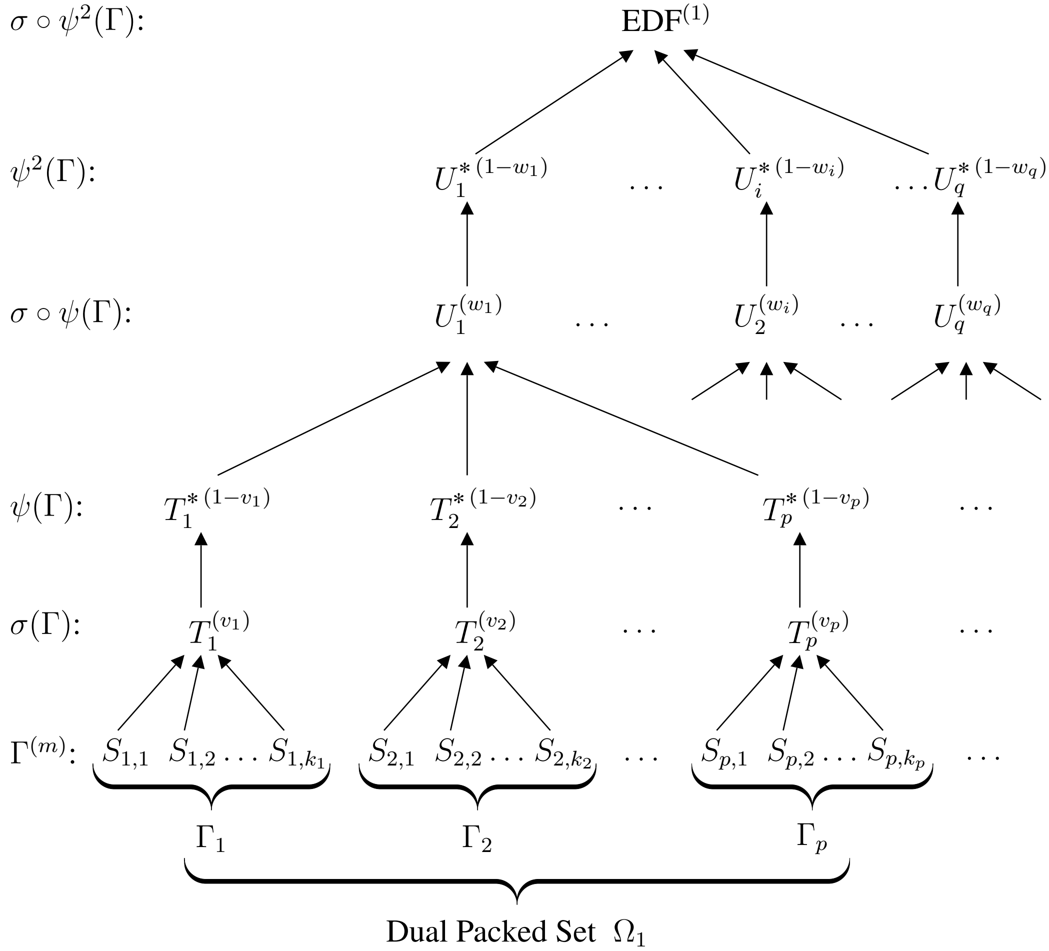


Figure 5.7. RUN subtree. U_1^* is a root server, $\{T_i^*\}_i$ is the collection of its child servers, and $\{S_{i,j}\}_{i,j}$ is the collection of its grandchild servers. Moreover, $\rho(\Omega_1) = p - 1 + \rho(U_1^*)$.

Lemma 5.3.1 (Parallel Execution Requirement)**.** *Let* Γ *be a set of servers and* $\pi[\Gamma] = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_p\}$ *be the packing of* Γ *by a packing algorithm* A*. Consider* $\Omega_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ *a dual packed set with* $k > 1$ *and let* $U_1^* = \psi^2(S_{i,j})$ *for some server* $S_{i,j}$ *in* Γ_j *and* Γ_j *in* Ω_1*. Then, there exists a real number* x*, called* excess*, with* $0 \leqslant x < 1$ *such that* $\rho(\Omega_1) = p - 1 + x$ *where* $p = |\Omega_1|$*. Moreover,* $\rho(U_1^*) = x$*. Excess* x *represents the amount of parallel execution required by* Ω_1*.*

Proof. By Definition 4.4.1 of the REDUCE operator, $0 \leqslant \rho(U_1^*) < 1$. Moreover,

$$
\begin{aligned}
\rho(U_1^*) &= 1 - \rho(U_1) \\
&= 1 - \sum_{i=1}^{p} (1 - \rho(\Gamma_i))) \\
&= 1 - p + \sum_{i=1}^{p} \rho(\Gamma_i)
\end{aligned}
$$

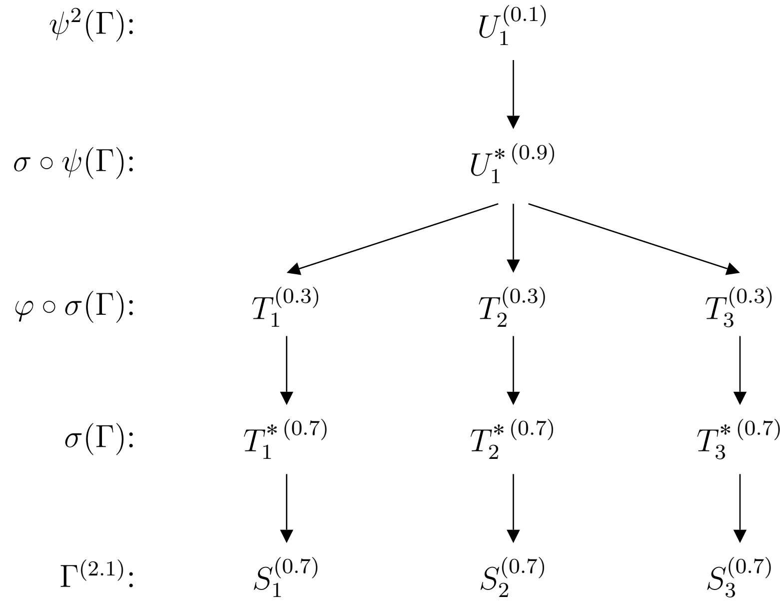


Figure 5.8. A reduction subtree of the primal packed set of servers S_1, S_2 and S_3. The notation S_i^{μ} means that $\rho(S_i) = \mu$.

$$= 1 - p + \rho(\Omega_1)$$

Hence, $\rho(\Omega_1) = p - 1 + x$ with $x = \rho(U_1^*)$. □

It is important to emphasize that this latter equality means that a dual-packed set can be scheduled on $|\Omega_i| - 1$ fully utilized processors and one partially utilized processor with rate x.

Also, note that if $x = 0$, then the dual set of Ω_i is a unit server and the scheduling problem of Ω_i can be solved, as shown by Theorem 4.2.1.

Let us consider a simple subtree example where root server U_1 has only three grandchild servers S_1, S_2 and S_3, all of them with utilization 0.7, as illustrated in Figure 5.8. Observe that, in this example, $x = 0.1$ and that $\rho(\Gamma) = 2.1$. Thus, two processors are "almost" enough to schedule Γ. More precisely, the valid schedule of Γ requires two full processors and a fraction 0.1 of a third processor. In other words, two processors must be executing continuously two of the three servers in Γ, and, when the third processor executes a server in Γ, for a fraction 0.1 of its bandwidth, then the three servers in Γ are executing in parallel. As can be seen, 0.1 is the computing requirement excess which prevents Γ to be feasible on two processors.

In this example, root server U_1^* has its rate precisely equal to excess 0.1. Thus, U_1^* deals with the amount of parallelism on three processors that the 3-server set Γ requires. This is coherent with the RUN scheduling rules exposed earlier. Indeed, if U_1^* executes at time t, then U_1 does not by Rule 5.2.2. Hence, by Rule 5.2.1 at time t, T_1^*, T_2^* and T_3^*, the U_1's clients, do not execute. In turn, this implies, by Rule 5.2.2, that T_1, T_2 and T_3 execute and finally, by Rule 5.2.1, that S_1, S_2 and S_3 execute in parallel on three processors at time t.

On the other hand, when U_1^* does not execute at time t, then U_1 does. Hence, one of the three servers T_1^*, T_2^* and T_3^* executes whereas the two others do not. Assume for instance that T_2^* is the server with earliest deadline that executes at time t. Hence, T_1^* and T_3^* do not, and, as a consequence, both T_1 and S_1, and T_3 and S_3 execute at time t while T_2 and S_2 do not.

Summarizing, if $\Omega_1 = \{\Gamma_1, \Gamma_2, \dots, \Gamma_p\}$ is a dual packed set of accumulated rate $p-1+x$ with $0 \leqslant x < 1$, then, for any $S_i \in \Gamma_j$ with $\Gamma_j \in \Omega_1$, the grandparent server $U_1^* = \psi^2(S_i)$ of S_i has rate x, and whenever U_1^* executes, all servers $\mathrm{ser}(\Gamma_j)$ in Ω_1 execute on p processors in parallel. Otherwise, when U_1^* does not execute, then $p-1$ servers in Ω_1 execute on $p-1$ processors.

Also, the dual level $\psi(\Omega_1)$ guarantees the correct exclusion between executions of the p servers in Ω_1 while they are scheduled on $p-1$ processors.

In appendix C, we will see how the decomposition of a general RUN tree into distinct subtrees can possibly be used to develop a RUN-based solution for the sporadic task model with implicit deadlines.

5.4 CONCLUSION

In this chapter, we have enunciated and explained the on-line scheduling rules used by the RUN algorithm once computed the off-line reduction tree of a set of tasks.

Although the RUN tree incurs some complexity in the overall algorithm, it is computed off-line. The on-line scheduling decisions use the previously computed RUN tree but follows straightforward rules.

Further, an alternative interpretation of the RUN tree, based on its decomposition in distinct subtrees, has been presented. More precisely, we have shown that in each subtree, comprised of a grandparent root server, its child and grandchild servers, the parallel execution requirements existing at the grandchild server level strictly corresponds to the executions of the root server.

In the next chapter, the whole RUN scheduling framework will be evaluated by simulation.

Chapter 6

The number of reduction levels in a RUN tree is a logarithmic function of the total number of primal tasks. As a consequence, RUN significantly outperforms existing optimal algorithms with an upper bound of $O(\log m)$ average preemptions per job on m processors ($\leqslant 3$ per job in all of our simulated task sets).

ASSESSMENT

6.1 INTRODUCTION

Now that we have completely and precisely described the reduction to uniprocessor real-time scheduling algorithm, we establish in this chapter results on the number of preemption and migration per job. Also, we characterize the complexity of the RUN algorithm.

As previously stated, for some particular task system, one or more iterations of the DUAL and PACK operations may be needed in order to reach a set of unit servers. Hence, the complexity of RUN depends upon the number m of identical processors, the total number n of tasks of the primal set to be scheduled and the number of reduction levels required by this task system.

However, as will be seen in this chapter, the number of reduction levels is a logarithmic function of the total number of primal tasks. As a consequence, we establish an upper bound on the average number of preemptions per job, which is a function of m and n.

Structure of the chapter

Section 6.2 deals with implementation details, describing how the bin-packing procedure can take profit of some slack in the task system. Then, the overall complexity of the RUN scheme is shown in Section 6.3.

The theoretical results presented in Section 6.4 and 6.5 were originally proposed by Greg Levin in (REGNIER et al., 2011). For the sake of completeness, we include these results in the dissertation.

In Section 6.6, the RUN algorithm is compared with many other optimal scheduling algorithms through intensive simulations, using randomly generated tasks sets.

6.2 RUN IMPLEMENTATION

At the first reduction level, we have m bins, *i.e.,* processors, each of which has size, *i.e.,* bandwidth, equal to one. On the other hand, we have a real-time set of n primal tasks that must be packed into at least m servers of rate less than or equal to one according to some bin-packing policy with property enunciated in Definition 4.3.1. Thereafter, at each supplementary level of reduction needed, child servers are packed into parent servers to be scheduled on less virtual processors than what would be needed at the child level.

In order to pack tasks at the primal level as well as at each reduction level, our implementation of RUN uses the worst-fit bin-packing heuristic, which runs in $O(k \log k)$ time where k is the number of tasks to be packed.

Also, our reduction procedure isolates off proper subsystems as soon as unit servers are found. In other words, each unit server and its descendants make an isolated scheduling reduction tree in which servers and primal tasks are scheduled by the RUN algorithm applied to this isolated subsystem, independently from all other subsystems required to schedule the complete primal task set.

As for the job-to-processor assignment algorithm, at each scheduler invocation, once the set of m running tasks is determined by the RUN algorithm (as in Figure 5.5), we use a simple greedy assignment scheme. In three passes through these m tasks, we first leave executing tasks on their current processors; second, we assign idle tasks to their last-used processor, when available, to avoid unnecessary migrations; and third, we assign remaining tasks to the remaining free processors arbitrarily.

Recall from Chapter 4 that duality is only defined for task sets with 100% utilization. For the sake of simplicity, we have assumed in Chapter 3 a fully-utilized system of m identical processors. However, when a primal task set does not fully-utilizes the m processors in the system, one can define dummy tasks to fill in the difference when needed. In such a case, it is possible to take advantage of the possible slack in the task system to improve performance.

To this end, we introduce the *slack packing* heuristic, as originally formalized by Greg Levin in (REGNIER et al., 2011), to distribute a task system's *slack* (defined as $m - \rho(\mathcal{T})$) among the aggregated servers at the end of the initial PACK step. Servers are filled to become unit servers, and then isolated from the system. The result is that some or all processors are assigned only non-migrating tasks and behave as they would in a partitioned schedule.

For example, suppose that the task set from Figure 5.5 runs on four processors instead

of three. The initial PACK can only place one 0.6 utilization task per server. From the 1 unit of slack provided by our fourth processor, we create a dummy task S_1^d with $\rho(S_1^d) = 0.4$ (and arbitrarily large deadline), pack it with S_1 to get a unit server and give it its own processor. Similarly, S_2 also gets a dedicated processor. Since S_1 and S_2 never need preempt or migrate, the schedule is more efficient. With 5 processors, this approach yields a fully partitioned system, where each task has its own processor. With low enough utilization, the first PACK usually results in m or fewer servers. In these cases, slack packing gracefully reduces RUN to Partitioned EDF.

It is important to note here that RUN does not rely on task synchronization like in previous optimal approaches based on Pfair. As a consequence, RUN is more compatible with symmetric multiprocessor (SMP) architectures than previous approaches since RUN generates less bus contention than Pfair approaches. Indeed, the quantum-based approach of Pfair implies that tasks may need to reload data into local caches at the start of each quantum, resulting in a period of increased bus traffic (HOLMAN, 2004). Since, under Pfair scheduling, quanta begin synchronously on all processors, the resulting bus traffic bursts generate a heavy bus contention at the start of each quantum. However, no such synchronization occurs under RUN scheduling, resulting in less bus contention than Pfair scheduling approaches.

6.3 REDUCTION COMPLEXITY

We now observe that the time complexity of a reduction procedure is polynomial and is dominated by the PACK operation. However, as there is no specific requirement on the (off-line) reduction procedure, any polynomial-time heuristic suffices. There are, for example, linear and log-linear time packing algorithms available (COFFMAN JR. et al., 1997; HOCHBAUM, 1997).

The following lemma establishes an upper bound on the number of servers obtained by packing an arbitrary number of servers.

Lemma 6.3.1. *Let Γ be a set of servers. Then, $|\sigma(\Gamma)| < 2\,\rho(\Gamma)$.*

Proof. Let $q = |\sigma(\Gamma)|$ and $u_i = \rho(S_i)$ for $S_i \in \sigma(\Gamma)$. Since $\sigma(\Gamma)$ is packed, there exists at most one server in $\sigma(\Gamma)$, as stated by Lemma 4.3.1, say S_q, such that $u_q < 1/2$. All other servers have utilization greater that $1/2$. Thus,

$$\sum_{i=1}^{q-2} u_i > \frac{(q-2)}{2}.$$

As $u_{q-1} + u_q > 1$, it follows that

$$\sum_{i=1}^{q} u_i = \rho(\Gamma) > n/2.$$

□

Theorem 6.3.1 (Reduction Complexity)**.** *RUN's off-line generation of a reduction sequence for n tasks on m processors requires $O(\log m)$ reduction steps and $O(f(n))$ time, where $f(n)$ is the time needed to pack n tasks.*

Proof. Let $\{\psi^i\}_{i \leqslant p}$ be a reduction sequence on $\mathcal{T}$, where p is the terminal level described in Theorem 4.4.1. Lemma 4.4.1 shows that a REDUCE operation, at worst, reduces the number of servers by about half, so $p = O(\log n)$.

Also, since $\mathcal{T}$ is a full utilization task set, $\rho(\mathcal{T}) = m$. If we let $n' = |\sigma(\mathcal{T})|$, Lemma 6.3.1 tells us that $m = \rho(\mathcal{T}) = \rho(\sigma(\mathcal{T})) > n'/2$. But as $\sigma(\mathcal{T})$ is just the one initial packing, it follows that p also is $O(\log n')$, and hence $O(\log m)$.

Finally, since constructing the dual of a system primarily requires computing n dual rates, a single REDUCE operation requires $O(f(n) + n)$ time. Hence, the time needed to perform the entire reduction sequence is described by $T(n) \leqslant T(n/2) + O(f(n) + n)$, which gives $T(n) = O(f(n))$. □

6.4 ON-LINE COMPLEXITY

As already seen, the RUN reduction is computed off-line, *i.e.,* during design time. Thus, the on-line complexity of RUN can be estimated using the off-line computed RUN tree and calculating the time and complexity introduced by on-line scheduling according to Rules 3.4.1 and 4.2.1. In order to do so, we consider a time window during which j jobs are released by the system of n tasks to be scheduled.

Theorem 6.4.1 (On-line Complexity)**.** *Each scheduler invocation of RUN takes $O(n)$ time, for a total of $O(jn \log m)$ scheduling overhead during any time interval when n tasks releasing a total of j jobs are scheduled on m processors.*

Proof. First, let's count the nodes in the RUN tree. In practice, a primal/dual pair comprised by a server S and its dual server S^* may be implemented as a single node. Also, there are n leaves at the primal level of the RUN tree, and as many as n servers in $\sigma(\mathcal{T})$. Above that, each level of the RUN tree has at most (approximately) half as many nodes as the preceding level. This gives us an approximate node bound of $n+n+n/2+n/4+\ldots = n+n(1/(1-1/2)) = 3n$

Next, consider the scheduling process described by Rules 3.4.1 and 4.2.1. The comparison of clients performed by EDF in Rule 3.4.1 does no worse than inspecting each client once. If we assign this cost to the client rather than the server, each node in the tree is inspected at most once per scheduling invocation. Also, Rule 4.2.1 is constant time for each primal/dual pair node. Thus the selection of m tasks to execute is constant time per node, of which there are at most $3n$. The previously described task-to-processor assignment requires 3 passes through a set of m tasks, and so may be done in $O(m) \leqslant O(n)$ time. Therefore, each scheduler invocation is accomplished in $O(n)$ time.

Since we only invoke the scheduler at WORK COMPLETE or JOB RELEASE events, any given job (real or virtual) can cause at most two scheduler invocations. The virtual jobs of servers are only released at the release times of their leaf descendants, so a single real job can cause no more than $O(\log m)$ virtual jobs to be released, since there are at most $O(\log m)$ reduction levels (Theorem 6.3.1).

Thus j real jobs result in no more than $jO(\log m)$ virtual jobs, so a time interval where j jobs are released will see a total scheduling overhead of $O(jn \log m)$. □

6.5 PREEMPTION BOUND

We now prove an upper bound on the average number of preemptions per job through a series of lemmas. To do so, as cleverly suggested by Greg Levin (REGNIER et al., 2011), we count the preemptions that a job *causes*, rather than the preemptions that a job *suffers*.

As a matter of fact, the number of preemptions that a single job can suffer is unbounded as can be seen through the simple following example. Consider two tasks $\tau_1:(1-\varepsilon, T)$ and $\tau_2:(\varepsilon, 1)$ and let k be the number of preemptions of the first job of τ_1 by jobs of τ_2. It is clear that k tends to infinity when T tends to infinity. However, for this example, the total number of jobs is $n+1$. Thus, the average number of preemptions per job equals $n/(n+1)$ which tends to one. As can be seen, while an arbitrarily long job may be preempted arbitrarily many times, the *average* number of preemptions per job is bounded.

In order to establish a general upper bound on the the average number of preemptions per job, we begin by defining some terminology. First, we say that a *context switch* occurs at time t when a new job, say J', starts executing at t and the previous job, say J, stops executing at t, either because J has completed or has lower priority than J' at time t. Second, when a context switch occurs where A begins running and B becomes idle, we say that A *replaces* B; moreover, if the current job of B still has work remaining, we say that A *preempts* B.

Since all scheduling decisions are made by EDF, we need only consider the preemptions caused by two types of scheduling events: *work complete events* (WCE), and *job release events*

(JRE). Also, while a WCE may possibly occur at a job deadline, it is always the case that a JRE occurs at a job deadline.

Lemma 6.5.1. *Each job from a task or server has exactly one* JRE *and one* WCE*. Further, the servers at any one reduction level cannot release more jobs than the original task set over any time interval.*

Proof. The first claim is obvious and is merely noted for convenience.

Next, since servers inherit deadlines from their clients and jobs are released at deadlines, a server cannot have more deadlines, and hence cannot release more jobs, than its clients. Also, a server's dual has the same number of jobs as the server itself. Thus, moving inductively up the RUN tree, it follows that a set of servers at one level cannot have more deadlines, or equivalently, more job releases, than the set of primal tasks at leaf level. ☐

Lemma 6.5.2. *Scheduling a system $\mathcal{T}$ of $n = m + 1$ tasks on m processors with RUN produces an average of no more than one preemption per job.*

Proof. When $n = m + 1$, there is only one reduction level and no packing; $\mathcal{T}$ is scheduled by applying EDF to its uniprocessor dual system. In such a case, we claim that dual JREs cannot cause preemptions in the primal system.

We first observe that when a dual JRE happens, it could only cause a preemption in the primal system if it were to cause a context switch in the dual system.

Now, consider an instant t at which a JRE happens in the dual system. Let J_i^* be the arriving dual job from task τ^* at time t, J_{i-1} be the last job of τ released before t and J_k^* be the job of task τ'^* running in the dual system just before t, with $\tau \neq \tau'$. By the definition of J_{i-1} and J_i, $t = J_i.r = J_{i-1}.d$, as illustrated in Figure 6.1, where diagonal crosshatch regions represent execution of other jobs.

In order for the arrival of J_i^* to cause a context switch, *i.e.,* to preempt J_k^* at time t, it must be that J_i^* has an earlier deadline than J_k^* at time t. However, in such a case, by Rule 3.4.1, J_k does not execute just before t in the primal system. As a consequence, τ's previous job J_{i-1} must be executing in the primal system just before t.

Thus, J_i^* *starts* executing at time t in the dual system precisely when τ's previous job J_{i-1} *stops* executing at time $t = J_{i-1}.d$ in the primal system. As a consequence, time t is both a JRE of J_i^* in the dual system and a WCE of J_{i-1} in the primal system. And, since this WCE in the primal system does not cause a preemption, the dual JRE at time t does not count a preemption in the primal system.

Hence, only WCE in the dual system can cause preemption in the primal system. Since there can be at most one WCE per job in the dual by Lemma 6.5.1, and, consequently, one preemption

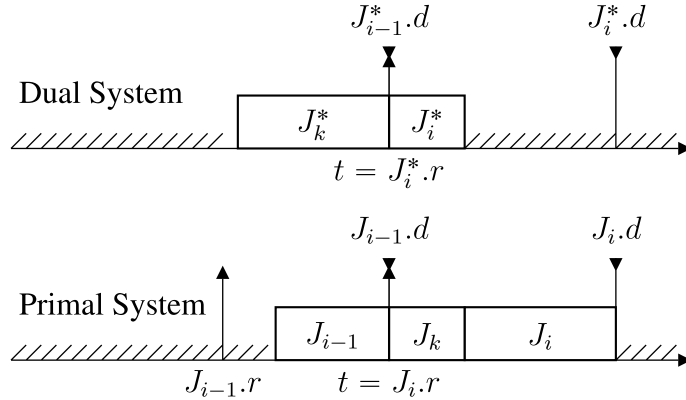


Figure 6.1. In the dual, the arrival of J^*_i preempts J^*_k. The matching primal event is just the previous job J_{i-1} finishing its work at its deadline, and is not a preemption.

in the primal, we conclude that there can be only one preemption in the primal system for each job released by a task in $\mathcal{T}$, as desired. □

Lemma 6.5.3. *A context switch at any level of the RUN tree causes exactly one context switch between two primal leaf tasks in $\mathcal{T}$.*

Proof. We proceed by induction on the number of levels, showing that a context switch at any level of the RUN tree causes exactly one context switch in the next level below (less reduced than) it.

Consider some tree level where a context switch occurs at time t and suppose we have a pair of client nodes (not necessarily of the same server parent) $C_{+,0}$ and $C_{-,1}$, where $C_{+,0}$ replaces $C_{-,1}$. We use the $+$ and $-$ signals to indicate that $C_{+,0}$ preempts $C_{-,1}$. Moreover, indexes 0 and 1 allow us to distinguishes between clients of a same server. All other jobs' "running" statuses at this level remain unchanged at time t.

Now, let $S_{+,0}$ and $S_{-,1}$ be the dual children of $C_{+,0}$ and $C_{-,1}$ in the RUN tree, respectively (*i.e.,* $C_{+,0} = S^*_{+,0}$ and $C_{-,1} = S^*_{-,1}$). By the dual scheduling Rule 4.2.1, it must be that $S_{-,1}$ replaces $S_{+,0}$ (see Figure 6.2 for node relationships).

Now, when server $S_{+,0}$ was running, it was executing exactly one of its client children, say $C_{+,0,1}$, and when $S_{+,0}$ gets switched off, so does $C_{+,0,1}$. Similarly, when $S_{-,0}$ was off, none of its clients were running, and when it gets switched on, exactly one of its clients, say $C_{-,1,0}$, begins to execute.

Also, just as the context switch at the higher (more reduced) level only effects the two servers $C_{+,0}$ and $C_{-,1}$, so too are these two clients $C_{+,0,1}$ and $C_{-,1,0}$ the only clients at this

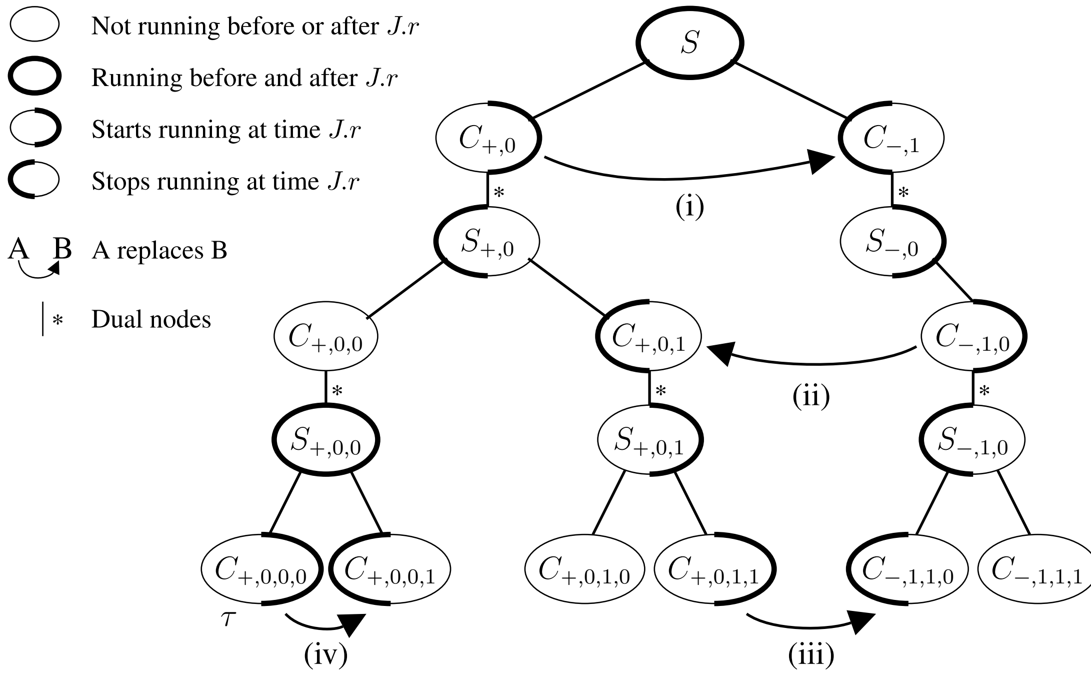


Figure 6.2. Two Preemptions from one job release In this 3-level part of a RUN tree, only relevant nodes are shown. A job release by τ corresponds to a job release and context switch at the top level (i), which propagates down to the right of the tree (ii, iii). That same job release by τ can cause it to preempt (iv) another client $C_{+,0,0,1}$ of its parent server $S_{+,0,0}$.

lower level affected by this operation; thus, $C_{-,1,0}$ must be replacing $C_{+,0,1}$. So here we see that a context switch at one client level of the RUN tree causes only a single context switch at the next lower client level of the tree (in terms of Figure 6.2, (i) causes (ii)).

This one context switch propagates down to the leaves, so inductively, a context switch anywhere in the RUN tree causes exactly one context switch in $\mathcal{T}$. □

Lemma 6.5.4. *If RUN requires p reduction levels for a task set $\mathcal{T}$, then any* JRE *by a task $\tau \in \mathcal{T}$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$.*

Proof. Suppose task τ releases job J at time $J.r$. This causes a job release at each ancestor server node above τ in the RUN tree (*i.e.,* on the path from leaf τ to the root). We will use Figure 6.2 for reference. Note that this figure represents only the relevant nodes for our discussion of a particular subtree, as stated in Definition 5.3.1, of a general RUN tree.

Let S be the highest ancestor server of τ in the RUN tree (S may be the root of the RUN tree) for which this JRE causes a context switch among its clients. As a consequence, some client of S, say $C_{+,0}$, has a job arrive with an earlier deadline than the currently executing client, say $C_{-,1}$, so $C_{+,0}$ preempts $C_{-,1}$. As described in the proof of Lemma 6.5.3, $C_{-,1}$'s dual $S_{-,1}$ replaces $C_{+,0}$'s dual $S_{+,0}$, and this context switch propagates down to a context switch between two tasks in $\mathcal{T}$, *i.e.,* preemption (iii) in Figure 6.2.

However, as no client of $S_{+,0}$ remains running at time $J.r$, the arrival of a job for τ's ancestor $C_{+,0}$ at this level cannot cause a JRE preemption at this time (it may cause a different client of $S_{+,0}$ to execute when $S_{+,0}$ begins running again, but this context switch will be charged to the event that causes $S_{+,0}$ to resume execution). Thus, when an inherited JRE time causes a context switch at one level, it cannot cause a *different* (second) context switch at the next level down. However, it may cause a second context switch two levels down, as for example, preemption (iv) in Figure 6.2. As can be seen, this figure shows two context switches, (iii) and (iv), in $\mathcal{T}$ that result from a single JRE of τ. One is caused by a job release by τ's ancestor child of the root, which propagates down to another part of the tree (iii). τ's parent server is not affected by this, stays running, and allows τ to preempt its sibling client when its new job arrives (iv).

Finally, while S is shown as the root and τ as a leaf in Figure 6.2, this argument would still apply if there were additional nodes above and below those shown, and τ were a descendant of node $C_{+,0,0,0}$. If there were additional levels, then τ's JRE could cause an additional preemption in $\mathcal{T}$ for each two such levels. Thus, if there are p reduction levels (*i.e.,* $p + 1$ levels of the RUN tree), a JRE by some original task τ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. □

Theorem 6.5.1. *Suppose RUN performs p reductions on task set $\mathcal{T}$ in reducing it to a single EDF system. Then RUN will suffer an average of no more than $\lceil (3p + 1)/2 \rceil = O(\log m)$ preemptions per job (and no more than 1 on average when $n = m + 1$) when scheduling $\mathcal{T}$.*

Proof. The $n = m + 1$ bound comes from Lemma 6.5.2. Otherwise, we use Lemma 6.5.1 to count preemptions based on jobs from $\mathcal{T}$ and the two EDF event types. By Lemma 6.5.4, a JRE by $\tau \in \mathcal{T}$ can cause at most $\lceil (p + 1)/2 \rceil$ preemptions in $\mathcal{T}$. The context switch that happens at a WCE in $\mathcal{T}$ is, by definition, not a preemption. However, a job of $\tau \in \mathcal{T}$ corresponds to one job released by each of τ's p ancestors, and each of these p jobs may have a WCE which causes (at most, by Lemma 6.5.3) one preemption in $\mathcal{T}$. Thus we have at most $p+\lceil (p+1)/2 \rceil = \lceil (3p + 1)/2 \rceil$ preemptions that can be attributed to each job from $\mathcal{T}$, giving our desired result since $p = O(\log m)$ by Theorem 6.4.1. □

In our simulations, we almost never observed a task set that required more than two reductions. Also, for $p = 2$, Theorem 6.5.1 gives a bound of 4 preemptions per job. While we never observe more than 3 preemptions per job in our randomly generated task sets, it is possible to do worse. The following 6-task set on 3 processors

$$\mathcal{T} = \{(.57, 4000), (.58, 4001), (.59, 4002), (.61, 4003), (.63, 4004), (.02, 3)\}$$

averages 3.99 preemptions per job, suggesting that our proven bound is tight.

Also, there exist task sets that require more than 2 reductions. For instance, the 11-task

Table 6.1. Reduction example of a taskset $\mathcal{T}$ comprised of 11 tasks with identical rate $\frac{7}{11}$, and with total utilization $\rho(\mathcal{T}) = 7$.

	Server Rate										
	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}
Γ	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$
$\sigma(\Gamma)$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$	$\frac{7}{11}$
$\psi(\Gamma)$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$	$\frac{4}{11}$
$\sigma(\psi(\Gamma))$	$\frac{8}{11}$		$\frac{8}{11}$		$\frac{8}{11}$		$\frac{8}{11}$		$\frac{8}{11}$		$\frac{4}{11}$
$\psi^2(\Gamma)$	$\frac{3}{11}$		$\frac{3}{11}$		$\frac{3}{11}$		$\frac{3}{11}$		$\frac{3}{11}$		$\frac{7}{11}$
$\sigma(\psi^2(\Gamma))$	$\frac{9}{11}$						$\frac{6}{11}$				$\frac{7}{11}$
$\psi^3(\Gamma)$	$\frac{2}{11}$						$\frac{5}{11}$				$\frac{4}{11}$

set with all rates equals to $7/11$ requires three reductions, independently of the bin-packing algorithm used for the reduction, with the sequence shown in Table 6.1.

As another example, the 47-taskset with all rates equals to $30/47$ requires four reduction level as shown in Table 6.2. In this example again, any bin-packing algorithm would require four reductions.

Although in the two above examples, the number of reduction level is independent of the bin-packing algorithm, this is not the case in general. For instance, consider a 41-taskset $\mathcal{T}$ comprised of 17 tasks with rate $\frac{14}{23}$, 24 tasks with rate $\frac{15}{23}$ and with total utilization $\rho(\mathcal{T}) = 26$. If tasks are ordered as in Table 6.3, the first fit bin-packing algorithm would require four levels of reduction as shown in Table 6.3. However, the worst-fit algorithm would require only three reduction levels, as shown in Table 6.4

As can be seen, such built task sets require narrowly constrained rates and randomly generated task sets requiring 3 or more reductions are rare. A 3-reduction task set was observed on 18 processors, and a 4-reduction set appeared on 24 processors, but even with 100 processors and hundreds of tasks, 3- and 4-reduction sets occur in less than 1 in 600 of the random task sets generated.

6.6 SIMULATION

We have evaluated RUN via extensive simulation using task sets generated for various levels of n tasks, m processors, and total utilization $\rho(\mathcal{T})$. Task rates were generated in the range of $[0.01, 0.99]$ following the Emberson procedure (EMBERSON et al., 2010) using the aleatory task generator (EMBERSON et al., 2011). Task periods were drawn independently from a uniform integer distribution in the range $[5, 100]$ and simulations were run for 1000 time units. Values reported for migrations and preemptions are *per job* averages, that is, total counts

Table 6.2. Reduction example of a 47-taskset $\mathcal{T}$ comprised of 47 tasks with rate $\frac{30}{47}$, and with total utilization $\rho(\mathcal{T}) = 30$.

	Server Rate																							
	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}
Γ	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$
$\sigma(\Gamma)$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$
$\psi(\Gamma)$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$
$\sigma(\psi(\Gamma))$	$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$	
$\psi^2(\Gamma)$	$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$	
$\sigma(\psi^2(\Gamma))$	$\frac{39}{47}$						$\frac{39}{47}$						$\frac{39}{47}$						$\frac{39}{47}$					
$\psi^3(\Gamma)$	$\frac{8}{47}$						$\frac{8}{47}$						$\frac{8}{47}$						$\frac{8}{47}$					
$\sigma(\psi^3(\Gamma))$	$\frac{40}{47}$																							
$\psi^4(\Gamma))$	$\frac{7}{47}$																							

(continue)	τ_{25}	τ_{26}	τ_{27}	τ_{28}	τ_{29}	τ_{30}	τ_{31}	τ_{32}	τ_{33}	τ_{34}	τ_{35}	τ_{36}	τ_{37}	τ_{38}	τ_{39}	τ_{40}	τ_{41}	τ_{42}	τ_{43}	τ_{44}	τ_{45}	τ_{46}	τ_{47}
Γ	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$
$\sigma(\Gamma)$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$	$\frac{30}{47}$
$\psi(\Gamma)$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$	$\frac{17}{47}$
$\sigma(\psi(\Gamma))$	$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{34}{47}$		$\frac{17}{47}$
$\psi^2(\Gamma)$	$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{13}{47}$		$\frac{30}{47}$
$\sigma(\psi^2(\Gamma))$	$\frac{39}{47}$						$\frac{39}{47}$						$\frac{39}{47}$						$\frac{26}{47}$				$\frac{30}{47}$
$\psi^3(\Gamma)$	$\frac{8}{47}$						$\frac{8}{47}$						$\frac{8}{47}$						$\frac{21}{47}$				$\frac{17}{47}$
$\sigma(\psi^3(\Gamma))$							$\frac{37}{47}$																$\frac{17}{47}$
$\psi^4(\Gamma))$							$\frac{10}{47}$																$\frac{30}{47}$

were divided by the number of jobs released during the simulation, averaged over all task sets. For each data point shown, 1000 task sets were generated.

For direct evaluation, we generated one thousand random n-task sets for each value $n = 17, 18, 20, 22, \ldots, 52$ (we actually took n up to 64, but results were nearly constant for $n \geqslant 52$). Each task set fully utilizes a system with 16 processors. We measured the number of reduction levels and the number of preemption points. Job completion is not considered a preemption point.

Figure 6.3(a) shows the number of reduction levels; none of the task sets generated require more than two reductions. For 17 tasks, only one level is necessary, as seen in Figure 1.6, and implied by Theorem 4.2.1. One or two levels are needed for $n \in [18, 48]$. None of our observed task sets require a second reduction for $n > 48$. With low average task rates, the first PACK gives servers with rates close to 1; the very small dual rates then sum to 1, yielding the terminal level.

Table 6.3. Reduction example of a 41-taskset $\mathcal{T}$ comprised of 17 tasks with rate $\frac{14}{23}$, 24 tasks with rate $\frac{15}{23}$ and with total utilization $\rho(\mathcal{T}) = 26$.

<table>
<tr><td></td><td colspan="24">Server Rate</td></tr>
<tr><td></td><td>τ_1</td><td>τ_2</td><td>τ_3</td><td>τ_4</td><td>τ_5</td><td>τ_6</td><td>τ_7</td><td>τ_8</td><td>τ_9</td><td>τ_{10}</td><td>τ_{11}</td><td>τ_{12}</td><td>τ_{13}</td><td>τ_{14}</td><td>τ_{15}</td><td>τ_{16}</td><td>τ_{17}</td><td>τ_{18}</td><td>τ_{19}</td><td>τ_{20}</td><td>τ_{21}</td><td>τ_{22}</td><td>τ_{23}</td><td>τ_{24}</td></tr>
<tr><td>Γ</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\sigma(\Gamma)$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\psi(\Gamma)$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td></tr>
<tr><td>$\sigma(\psi(\Gamma))$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td></tr>
<tr><td>$\psi^2(\Gamma)$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td></tr>
<tr><td>$\sigma(\psi^2(\Gamma))$</td><td colspan="6">$\frac{18}{23}$</td><td colspan="6">$\frac{18}{23}$</td><td colspan="6">$\frac{18}{23}$</td><td colspan="6">$\frac{18}{23}$</td></tr>
<tr><td>$\psi^3(\Gamma)$</td><td colspan="6">$\frac{5}{23}$</td><td colspan="6">$\frac{5}{23}$</td><td colspan="6">$\frac{5}{23}$</td><td colspan="6">$\frac{5}{23}$</td></tr>
<tr><td>$\sigma(\psi^3(\Gamma))$</td><td colspan="24">$\frac{20}{23}$</td></tr>
<tr><td>$\psi^4(\Gamma))$</td><td colspan="24">$\frac{3}{23}$</td></tr>
</table>

<table>
<tr><td>(continue)</td><td>τ_{25}</td><td>τ_{26}</td><td>τ_{27}</td><td>τ_{28}</td><td>τ_{29}</td><td>τ_{30}</td><td>τ_{31}</td><td>τ_{32}</td><td>τ_{33}</td><td>τ_{34}</td><td>τ_{35}</td><td>τ_{36}</td><td>τ_{37}</td><td>τ_{38}</td><td>τ_{39}</td><td>τ_{40}</td><td>τ_{41}</td></tr>
<tr><td>Γ</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\sigma(\Gamma)$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{14}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\psi(\Gamma)$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{9}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td><td>$\frac{8}{23}$</td></tr>
<tr><td>$\sigma(\psi(\Gamma))$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{17}{23}$</td><td colspan="2">$\frac{16}{23}$</td><td colspan="2">$\frac{16}{23}$</td><td colspan="2">$\frac{16}{23}$</td><td>$\frac{8}{23}$</td></tr>
<tr><td>$\psi^2(\Gamma)$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{6}{23}$</td><td colspan="2">$\frac{7}{23}$</td><td colspan="2">$\frac{7}{23}$</td><td colspan="2">$\frac{7}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\sigma(\psi^2(\Gamma))$</td><td colspan="6">$\frac{18}{23}$</td><td colspan="6">$\frac{19}{23}$</td><td colspan="4">$\frac{14}{23}$</td><td>$\frac{15}{23}$</td></tr>
<tr><td>$\psi^3(\Gamma)$</td><td colspan="6">$\frac{5}{23}$</td><td colspan="6">$\frac{4}{23}$</td><td colspan="4">$\frac{9}{23}$</td><td>$\frac{8}{23}$</td></tr>
<tr><td>$\sigma(\psi^3(\Gamma))$</td><td colspan="16">$\frac{18}{23}$</td><td>$\frac{8}{23}$</td></tr>
<tr><td>$\psi^4(\Gamma))$</td><td colspan="16">$\frac{5}{23}$</td><td>$\frac{15}{23}$</td></tr>
</table>

The box-plot in Figure 6.3(b) shows the distribution of preemption points as a function of the number of tasks. We see a strong correlation between the number of preemptions and number of reduction levels; where there is mostly only one reduction level, preemptions per job is largely independent of the size of the task set. Indeed, for $n \geqslant 36$, the median preemption count stays nearly constant just below 1.5. Even in the worst case, no task set ever incurs more than 2.8 preemptions per job on average.

Next, we ran comparison simulations against other optimal algorithms. In Figure 6.4, we count migrations and preemptions made by RUN, LLREF (CHO et al., 2006), EKG (ANDERSSON; TOVAR, 2006) and DP-Wrap (LEVIN et al., 2010) (with these last two employing the simple *mirroring* heuristic) while increasing processor count from 2 to 32. Most of LLREF's results are not shown to preserve the scale of the rest of the data. Whereas the performance of LLREF, EKG and DP-Wrap get substantially worse as m increases, the overhead for RUN quickly levels off, showing that RUN scales quite well with system size.

Table 6.4. Reduction example of a 41-taskset $\mathcal{T}$ comprised of 17 tasks with rate $\frac{14}{23}$, 24 tasks with rate $\frac{15}{23}$ and with total utilization $\rho(\mathcal{T}) = 26$ using the worst-fit bin-packing algorithm.

	Server Rate																					
	τ_{1}	τ_{2}	τ_{3}	τ_{4}	τ_{5}	τ_{6}	τ_{7}	τ_{8}	τ_{9}	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}
Γ	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$
$\sigma(\Gamma)$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{14}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$
$\psi(\Gamma)$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{9}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$
$\sigma(\psi(\Gamma))$	$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{18}{23}$		$\frac{17}{23}$		$\frac{16}{23}$		$\frac{16}{23}$	
$\psi^2(\Gamma)$	$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{5}{23}$		$\frac{6}{23}$		$\frac{7}{23}$		$\frac{7}{23}$	
$\sigma(\psi^2(\Gamma))$	$\frac{20}{23}$								$\frac{20}{23}$								$\frac{20}{23}$					
$\psi^3(\Gamma)$	$\frac{3}{23}$								$\frac{3}{23}$								$\frac{3}{23}$					

(continue)	τ_{23}	τ_{24}	τ_{25}	τ_{26}	τ_{27}	τ_{28}	τ_{29}	τ_{30}	τ_{31}	τ_{32}	τ_{33}	τ_{34}	τ_{35}	τ_{36}	τ_{37}	τ_{38}	τ_{39}	τ_{40}	τ_{41}
Γ	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$
$\sigma(\Gamma)$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$	$\frac{15}{23}$
$\psi(\Gamma)$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$	$\frac{8}{23}$
$\sigma(\psi(\Gamma))$	$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{16}{23}$		$\frac{8}{23}$
$\psi^2(\Gamma)$	$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{7}{23}$		$\frac{15}{23}$
$\sigma(\psi^2(\Gamma))$	$\frac{21}{23}$						$\frac{21}{23}$						$\frac{21}{23}$						$\frac{15}{23}$
$\psi^3(\Gamma)$	$\frac{2}{23}$						$\frac{2}{23}$						$\frac{2}{23}$						$\frac{8}{23}$

Finally, we simulated EKG, RUN, and Partitioned EDF at lower task set rates (LLREF and DP-Wrap were excluded, as they consistently perform worse than EKG). Because 100% utilization is unlikely in practice, and because EKG is optimized for utilizations in the 50-75% range, we felt these results to be of particular interest. For RUN, we employed the slack-packing heuristic. Because this often reduces RUN to Partitioned EDF for lower utilization task sets, we include Partitioned EDF for comparison in Figure 6.5's preemptions per job plot. Values for Partitioned EDF are only averaged over task sets where a successful partition occurs, and so stop at 94% utilization. The second plot shows the fraction of task sets that achieve successful partition onto m processors, and consequently, where RUN reduces to Partitioned EDF.

6.7 CONCLUSION

With its few migrations and preemptions at full utilization, its efficient scaling with increased task and processor counts, and its frequent reduction to Partitioned EDF on lower utilization task sets, RUN represents a substantial performance improvement in the field of optimal schedulers.

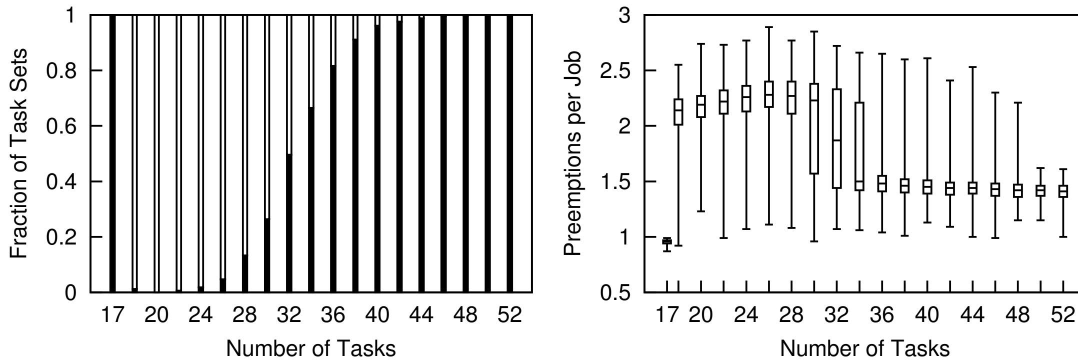


Figure 6.3. Fraction of task sets requiring 1 (filled box) and 2 (empty box) reduction levels; Distributions of the average number of preemptions per job, their quartiles, and their minimum and maximum values. All RUN simulations on 16 processor systems at full utilization.

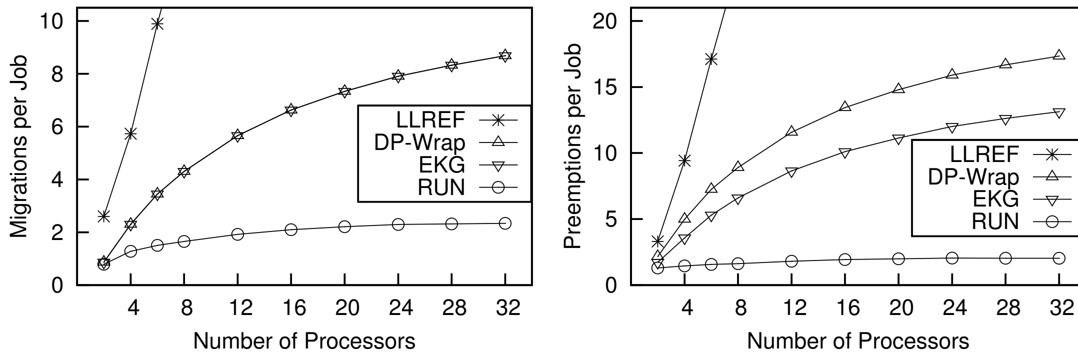


Figure 6.4. Migrations- and preemptions-per-job by LLREF, DP-Wrap, EKG, and RUN as number of processors m varies from 2 to 32, with full utilization and $n = 2m$ tasks. Note: DP-Wrap and EKG have the same migration curves.

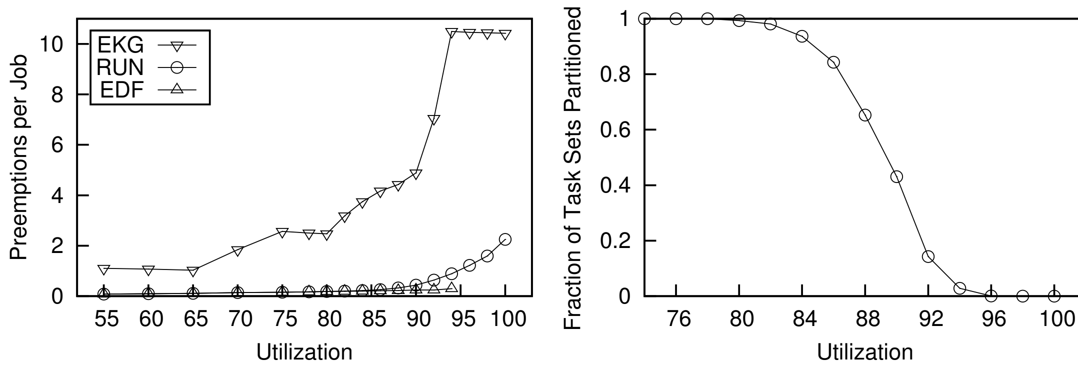


Figure 6.5. Preemptions per job for EKG, RUN, and Partitioned EDF as utilization varies from 55 to 100%, with 24 tasks on 16 processors; Partitioning success rate for worst-fit bin packing under the same conditions.

Chapter 7

Reduction to Uniprocessor: a seminal path to optimality.

CONCLUSION

We have presented the optimal RUN multiprocessor real-time scheduling algorithm. RUN transforms the problem of scheduling a set of periodic real-time task with implicit deadlines on two or more processors into a collection of one or more of the same problem on uniprocessor systems. As a consequence, the RUN algorithm furnishes a polynomial transformation from the multiprocessor to the uniprocessor scheduling problem, showing that the first problem is not more complicated than the latter.

RUN employs a semi-partitioned approach, but partitions tasks among servers rather than processors. RUN also does not proportional fairness but instead partitioned proportionate fairness. That is, each server generates a job between consecutive deadlines of any client tasks, and that job is assigned a workload proportional to the server's rate. Thus, servers globally shares the total processing bandwidth. As regards the jobs of a server clients, they collectively perform a proportionally "fair" amount of work between any two client deadlines, but such deadlines do not demand fairness among the individual client tasks. As a consequence, tasks in different branches of the server tree may have little influence on each others' scheduling. This is in stark contrast to previous optimal algorithms, where every unique system deadline imposes a new time slice and such slices cause preemptions for many or all tasks.

The limited isolation of groups of tasks provided by server partitioning and the reduced context switching imposed by minimal proportional fairness make RUN significantly more efficient than previous optimal algorithms.

Instead of statically allocating tasks to specific processors, the approach described in this work controls migration at run-time via the dual and packing operations. A series of operation is carried out aiming at transforming a multiprocessor scheduling problem into equivalent uniprocessor scheduling problems. Like partition-based approaches, solutions to uniprocessor scheduling can be used. Differently from other migration-control schemes, the proposed

approach makes use of servers as a means of scheduling transformed systems and generating migration necessary points.

It is worth emphasizing that the approach being proposed here shares some aspects with global-based approaches. Among these aspects are the possibility of optimality and the absence of static allocation to processors.

As regards the overhead of RUN, a theoretical upper bound of $O(\log m)$ average preemptions per job on m processors. Also, extensive simulations have shown that only a few preemption points per job are generated on average, allowing the RUN algorithm to significantly outperform prior optimal algorithms. Simulations of a varying number of processors have also shown that RUN scales well as the number of tasks and processors increase.

For non fully-utilized systems, it was shown that the system slack can be efficiently shared between processors in order to increase the chance for the bin packing procedure to find a proper partition. In such a case, RUN reduces to the more efficient partitioned approach of Partitioned EDF.

These results have both practical and theoretical implications. The overhead of RUN is low enough to justify implementation on actual multiprocessor architectures.

At present, our approach only works for fixed-rate task sets with implicit deadlines. Theoretical challenges include extending the model to more general problem domains such as sporadic tasks with constrained deadlines.

Also, the use of uniprocessor scheduling results to solve the multiprocessor problem raises interesting questions in the analysis of fault tolerance, energy consumption and adaptability.

We believe that this novel approach to optimal scheduling introduces a fertile field of research to explore and further build upon. Examples of open research topics could be:

- Avoid the necessity of the reduction tree, or at least, of the dual scheduling level;
- Study the possibility of eliminating some preemption points by skipping unnecessary deadlines inherited by a server from its clients;
- Explore the impact of using other uniprocessor scheduling algorithm different from EDF as policy for servers;
- Take profit of the possible slack in the task system to improve bin-packing and/or reduce the needed number of reduction levels;
- Extend the RUN algorithm to the sporadic task model with implicit or constrained deadlines;
- Explore the impact of using some other optimal uniprocessor scheduling algorithm than EDF as policy for servers;

- Characterize the impact of the floating point arithmetic or discrete arithmetic used by the practical multiprocessor system on the schedulability of tasksets using less or 100% of the processing power.

BIBLIOGRAPHY

ANDERSON, J.; SRINIVASAN., A. Pfair scheduling: Beyond periodic task systems. In: *Proceedings of the 7th International Conference on Real-time Computing Systems and Applications*. [S.l.: s.n.], 2000. p. 297–306.

ANDERSON, J.; SRINIVASAN., A. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, v. 68, p. 157–204, February 2004.

ANDERSSON, B.; BLETSAS, K.; BARUAH, S. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: *IEEE RTSS*. [S.l.: s.n.], 2008. p. 385–394.

ANDERSSON, B.; TOVAR, E. Multiprocessor scheduling with few preemptions. In: *IEEE Embedded and Real-Time Computing Systems and Applications*. [S.l.: s.n.], 2006. p. 322–334.

BARUAH, S. Scheduling periodic tasks on uniform multiprocessors. *Inf. Process. Lett.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 80, p. 97–104, October 2001. ISSN 0020-0190. Disponível em: <http://dl.acm.org/citation.cfm?id=511722.511727>.

BARUAH, S.; CHEN, D.; GORINSKY, S.; MOK, A. Generalized multiframe tasks. *Real-Time Systems*, Springer Netherlands, v. 17, p. 5–22, 1999. ISSN 0922-6443. 10.1023/A:1008030427220. Disponível em: <http://dx.doi.org/10.1023/A:1008030427220>.

BARUAH, S.; COHEN, N. K.; PLAXTON, C. G.; VARVEL, D. A. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, v. 15, n. 6, p. 600–625, 1996.

BARUAH, S.; GEHRKE, J.; PLAXTON, C. Fast scheduling of periodic tasks on multiple resources. In: *Proceedings of the 9th International Parallel Processing Symposium*. [S.l.: s.n.], 1995. p. 280–288.

BARUAH, S.; GOOSSENS, J. Scheduling real-time tasks: Algorithms and complexity. In: LEUNG, J. Y.-T. (Ed.). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. [S.l.]: Chapman Hall/CRC Press, 2004.

BARUAH, S.; MOK, A.; ROSIER, L. Preemptively scheduling hard-real-time sporadic tasks on one processor. In: *IEEE RTSS*. [S.l.: s.n.], 1990. p. 182 –190.

BARUAH, S. K.; COHEN, N. K.; PLAXTON, C. G.; VARVEL, D. A. Proportionate progress: a notion of fairness in resource allocation. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1993. (STOC '93), p. 345–354. ISBN 0-89791-591-7. Disponível em: <http://doi.acm.org/10.1145/167088.167194>.

BASTONI, A.; BRANDENBURG, B.; ANDERSON, J. Is semi-partitioned scheduling practical? In: *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*. [S.l.: s.n.], 2011. p. 125 –135. ISSN 1068-3070.

BERTOGNA, M. *Real-Time Scheduling Analysis for Multiprocessor Platforms*. Tese (Doutorado) — Scuola Superiore Sant'Anna, Pisa, 2007.

BOURBAKI, N. *Theory of Sets*. [S.l.]: Addison-Wesley, 1968. (Elements of Mathematics).

BURNS, A.; WELLINGS, A. *Real-Time Systems and Programming Languages*. 4. ed. [S.l.]: Addison Wesley Longmain, 2009.

BUTTAZZO, G. C. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 29, p. 5–26, January 2005. ISSN 0922-6443. Disponível em: <http://dl.acm.org/citation.cfm?id=1035387.1035388>.

CARPENTER, J.; FUNK, S.; HOLMAN, P.; SRINIVASAN, A.; ANDERSON, J.; BARUAH, S. A categorization of real-time multiprocessor scheduling problems and algorithms. In: *Handbook on Scheduling Algorithms, Methods, and Models*. [S.l.]: Chapman Hall/CRC, Boca, 2004.

CHO, H.; RAVINDRAN, B.; JENSEN, E. D. An optimal real-time scheduling algorithm for multiprocessors. In: *IEEE RTSS*. [S.l.: s.n.], 2006. p. 101–110.

CHO, S.; LEE, S.-K.; AHN, S.; LIN, K.-J. Efficient real-time scheduling algorithms for multiprocessor systems. *IEICE Trans. Communications*, Gothenburg, Sweden, E85-B, n. 12, p. 2859–2867, 2002.

COFFMAN JR., E. G.; GAREY, M. R.; JOHNSON, D. S. Approximation algorithms for bin packing: a survey. In: ____. Boston, MA, USA: PWS Publishing Co., 1997. p. 46–93. ISBN 0-534-94968-1. Disponível em: <http://dl.acm.org/citation.cfm?id=241938.241940>.

DENG, Z.; LIU, J. W.-S.; SUN, J. Scheme for scheduling hard real-time applications in open system environment. In: *ECRTS*. [S.l.: s.n.], 1997. p. 191–199.

DERTOUZOS, M.; MOK, A. Multiprocessor Online Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 15, n. 12, p. 1497–1506, 1989. ISSN 0098-5589.

DERTOUZOS, M. L. Control robotics: The procedural control of physical processes. In: *IFIP Congress'74*. [S.l.: s.n.], 1974. p. 807–813.

EASWARAN, A.; SHIN, I.; LEE, I. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 43, n. 1, p. 25–59, 2009. ISSN 0922-6443.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. I. Techniques for the synthesis of multiprocessor tasksets. In: *WATERS*. [S.l.: s.n.], 2010. p. 6–11.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. I. *A taskset generator for experiments with real-time task sets*. Jan. 2011. http://retis.sssup.it/waters2010/data/taskgen-0.1.tar.gz.

FISHER, N.; GOOSSENS, J.; BARUAH, S. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 45, n. 1-2, p. 26–71, 2010. ISSN 0922-6443.

FISHER, N. W. *The Multiprocessor Real-Time Scheduling of General Task Systems*. Tese (Doutorado) — University of North Carolina, Chapel Hill, 2007.

FUNAOKA, K.; KATO, S.; YAMASAKI, N. Work-conserving optimal real-time scheduling on multiprocessors. In: *IEEE ECRTS*. [S.l.: s.n.], 2008. p. 13–22.

FUNK, S. LRE-TL: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, v. 46, p. 332–359, 2010.

FUNK, S. H. *EDF Scheduling on Heterogeneous Multiprocessors*. Tese (Doutorado) — University of North Carolina, 2004.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. [S.l.]: W. H. Freeman and Company, 1979.

GEORGE, L.; RIVIERRE, N.; SPURI, M. *Preemptive and Non-Preemptive Real-Time Uniprocessor Scheduling*. [S.l.], 1996.

HILDEBRANDT, J.; GOLATOWSKI, F.; TIMMERMANN, D. Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. *Real-Time Systems, Euromicro Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 0208, 1999.

HOCHBAUM, D. S. (Ed.). *Approximation algorithms for NP-hard problems*. Boston, MA, USA: PWS Publishing Co., 1997. ISBN 0-534-94968-1.

HOLMAN, P.; ANDERSON, J. H. Adapting Pfair Scheduling for Symmetric Multiprocessors. *Journal of Embedded Computing*, IOS Press, v. 1, n. 4, p. 543–564, 2005.

HOLMAN, P. L. *On the Implementation of Pfair-scheduled Multiprocessor Systems*. Tese (Doutorado) — University of North Carolina, Chapel Hill, 2004.

HONG, K.; LEUNG, J.-T. On-Line Scheduling of Real-Time Tasks. In: *In Proceedings of the Real-Time Systems Symposium*. Huntsville, AL, USA: IEEE Computer Society, 1988. p. 244–250.

HORN, W. A. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, Wiley Subscription Services, Inc., A Wiley Company, v. 21, n. 1, p. 177–185, 1974.

KATO, S.; YAMASAKI, N.; ISHIKAWA, Y. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: *IEEE ECRTS*. [S.l.: s.n.], 2009. p. 249–258.

KOREN, G.; AMIR, A.; DAR, E. The power of migration in multi-processor scheduling of real-time systems. In: *ACM-SIAM symposium on Discrete algorithms*. [S.l.: s.n.], 1998. (SODA ’98), p. 226–235.

LEVIN, G.; FUNK, S.; SADOWSKI, C.; PYE, I.; BRANDT, S. DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In: *IEEE ECRTS*. [S.l.: s.n.], 2010. p. 3–13.

LEVIN, G.; SADOWSKI, C.; PYE, I.; BRANDT, S. *SNS: a simple model for understanding optimal hard real-time multi-processor scheduling*. [S.l.], 2009.

LIU, C. L. Scheduling algorithms for multiprogram in a hard real-time environment. *JPL Space Programs Summary*, II, p. 37–60, 1969.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogram in a hard real-time environment. *Journal of ACM*, v. 20, n. 1, p. 40–61, 1973.

LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice-Hall, 2000.

MASSA, E.; LIMA, G. A bandwidth reservation strategy for multiprocessor real-time scheduling. In: *IEEE RTAS*. [S.l.: s.n.], 2010. p. 175 –183.

MCNAUGHTON, R. Scheduling with deadlines and loss functions. *Management Science*, v. 6, n. 1, p. 1–12, 1959.

MOIR, M.; RAMAMURTHY, S. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In: *IEEE RTSS*. [S.l.: s.n.], 1999. p. 294 –303.

MOK, A. K.-L. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. Tese (Doutorado) — Massachusetts Institute of Technology, 1983.

NELISSEN, G.; BERTEN, V.; GOOSSENS, J.; MILOJEVIC, D. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*. [S.l.: s.n.], 2011. v. 1, p. 15 –24. ISSN 1533-2306.

PARK, M.; HAN, S.; KIM, H.; CHO, S.; CHO, Y. Zl scheme: Generalization of edzl scheduling algorithm for real-time multiprocessor systems. *Information: An International Interdisciplinary Journal*, v. 8, n. 5, p. 683–691, October 2005.

PIAO, X.; HAN, S.; KIM, H.; PARK, M.; CHO, Y.; CHO, S. Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. In: *Proc. of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. [S.l.: s.n.], 2006. p. 359–364.

REGNIER, P.; LIMA, G.; MASSA, E.; LEVIN, G.; BRANDT, S. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. [S.l.: s.n.], 2011. p. 104 –115. ISSN 1052-8725.

SAHNI, S. Preemptive Scheduling with Due Dates. *Operations Research*, v. 27, n. 5, p. 925–934, 1979. Disponível em: <http://or.journal.informs.org/cgi/content/abstract/27/5/925>.

SPURI, M.; BUTTAZZO, G. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Syst.*, v. 10, n. 2, p. 179–210, 1996.

UTHAISOMBUT, P. Generalization of EDF and LLF: Identifying all optimal online algorithms for minimizing maximum lateness. *Algorithmica*, Springer New York, v. 50, p. 312–328, 2008. ISSN 0178-4617. 10.1007/s00453-007-9083-9. Disponível em: <http://dx.doi.org/10-.1007/s00453-007-9083-9>.

ZHU, D.; MOSSÉ, D.; MELHEM, R. Multiple-resource periodic scheduling problem: how much fairness is necessary? In: *IEEE RTSS*. Washington, DC, USA: IEEE Computer Society, 2003. p. 142–151. ISBN 0-7695-2044-8. Disponível em: <http://portal.acm.org/citation-.cfm?id=956418.956616>.

ZHU, D.; QI, X.; MOSSÉ, D.; MELHEM, R. An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, v. 71, n. 10, p. 1411 – 1425, 2011. ISSN 0743-7315.

APPENDIX

Appendix A

Scheduling idle time is somehow equivalent to scheduling execution time.

IDLE SERIALIZATION

During the first two years of this PhD research, we have been actively working on the idea of scheduling both execution and idle times in order to improve the efficiency for generating a schedule.

As a first attempt of idle scheduling procedure, we have developed a new approach based on serializing idle time. We give here a brief description of this idle serialization approach, since this idea has finally led us to devise our actual proposal, RUN, an optimal algorithm for periodic task set with implicit deadlines.

We warn the reader that the material presented here has not be validated by any referee based procedure. Thus, it may contain some imprecision. However, since we have developed an algorithm based on idle serialization and estimated its efficiency through simulations, we find it convenient to expose this material here.

A.1 FRAME

Time is mapped to the non-negative real set and time intervals are the usual intervals of $\mathbb{R}$.

We call *frame*, denoted $[s, f)_k$, the execution time available on a processor P_k during time interval $[s, f)$. An idle frame is one during which no job executes. We denote $[s, f)_{k,i}$ the frame in processor during which job J_i executes continuously.

At any time t, a scheduling policy assigns frames to the current active jobs. The set of active jobs at t, denoted $A(t)$, represents all jobs released at or before t but not yet finished by t. Note that $A(t)$ contains jobs partially executed by t and so, can be defined as the set of jobs such that $J_i.r \leqslant t$ and $e(J_i, t) > 0$.

Definition A.1.1 (Serialized, Parallel, Concurrent and Adjacent Frames)**.** *Consider two frames* $F_j = [s, f)_j$ *and* $F_k = [s', f')_k$ *on two processors* P_j *and* P_k*, respectively.*

- F_j *and* F_k *are serialized if both* $j = k$ *and* $[s, f) \cap [s', f') = \{\}$*;*
- F_j *and* F_k *are serializable if* $[s, f) \cap [s', f') = \{\}$*. In words, serializable frames are those that can be serialized in the same processor;*
- F_j *and* F_k *are adjacent if* F_j *and* F_k *are serialized and if* $f = s'$ *or* $f' = s$*.*

A.2 MAPPING

Definition A.2.1 (Mapping of a job)**.** *A mapping of a job* J_i *on a multiprocessor system* Π*, denoted* $M_i(t)$*, is a set of frames reserved at time* t *on a subset of* Π *for the future execution of* J_i *such that:*

- *The first frame of* $M_i(t)$ *begins after the release time of* J_i*;*
- *The frames of* $M_i(t)$ *are pairwise serializable and they do not overlap with frames of any other mapping;*
- *The cumulative length of all frames of* $M_i(t)$ *equals* $e(J_i, t)$*.*

Upon arrival of a job J_i at time t, three scenarios are possible. First, a mapping may be assigned to J_i immediately. Second, J_i may be rejected according to some admission criterion. Third, the mapping assignment of J_i may be delayed to some future instant. In this later case, J_i remains in the ready queue $\mathbb{Q}(t)$ until the eventual assignment of a mapping to J_i or the rejection of J_i. Thus, $\mathbb{Q}(t)$ is the set of released jobs at t, not yet mapped nor rejected.

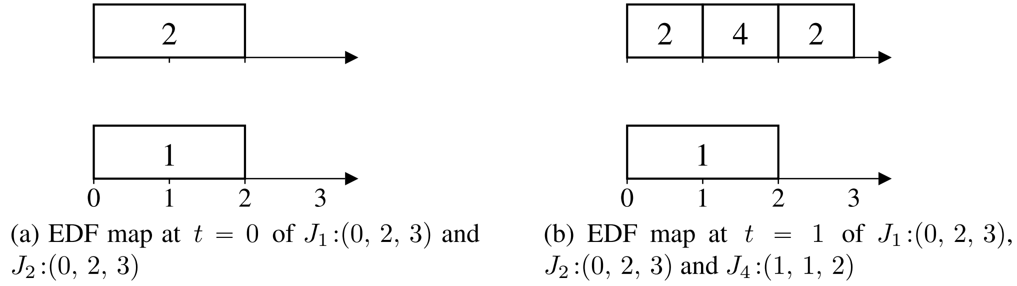
Definition A.2.2 (Map)**.** *A map* $\mathbb{M}(t)$ *at time* t *is the set of all mappings defined on* Π *at* t*. Formally,*

$$\mathbb{M}(t) = \{M_i(t), J_i \in \mathcal{J} \backslash \mathbb{Q}(t) \ \wedge \ r_i \leqslant t\}$$

A processor map $\mathbb{M}_k(t)$ *is the set of all frames of* $\mathbb{M}(t)$ *reserved on processor* P_k *for the execution of some job. Formally,*

$$\mathbb{M}_k(t) = \{F \in \mathbb{M}(t), F \cap [0, +\infty)_k = F\}$$

For example, consider the 3-task set $\mathcal{T} = \{\tau_1{:}(2, 3), \tau_2{:}(2, 3), \ \tau_3{:}(4, 6)\}$. The mappings assigned to $J_1{:}(0,\ 2,\ 3)$ and $J_2{:}(0,\ 2,\ 3)$ by EDF are shown in Figure A.1a. The resulting processor maps $\mathbb{M}_1(0)$ and $\mathbb{M}_2(0)$ equals $\{[0, 2)_{1,1}\}$ and $\{[0, 2)_{2,2}\}$, respectively; the map $\mathbb{M}(0)$ equals $\mathbb{M}_1(0) \cup \mathbb{M}_2(0)$ and the ready queue $\mathbb{Q}(0)$ equals $\{J_3{:}(0,\ 4,\ 6)\}$.



(a) EDF map at $t = 0$ of J_1:(0, 2, 3) and J_2:(0, 2, 3)

(b) EDF map at $t = 1$ of J_1:(0, 2, 3), J_2:(0, 2, 3) and J_4:(1, 1, 2)

Figure A.1. EDF map examples.

Frames of a map $\mathbb{M}(t)$ can either be free or locked. A free frame can be modified at some future scheduling instant, while locked frames are immutable. For instance, suppose that at time $t = 1$ a job J_4:(1, 1, 2) is added to our 3-task set example. Assuming that $[0, 2)_{2,1}$ is a free frame at $t = 0$, then, the resulting map $\mathbb{M}(1)$ assigned by the EDF scheduling policy, and shown by Figure A.1b, would be $\mathbb{M}(1) = \{[0, 2)_{1,1}, [0, 1)_{2,2}, [1, 2)_{2,4}, [2, 3)_{2,2}\}$ and $\mathbb{Q}(1) = \{J_3$:(0, 4, 6)$\}$.

A.3 LEVEL

Definition A.3.1 (Processor level in a map)**.** *At time t, the level $\lambda_k(\mathbb{M}(t))$ of a processor P_k regarding a map $\mathbb{M}(t)$ is the instant of the end of the last frame assigned to P_k in $\mathbb{M}(t)$ if any. If there is no frame allocated to P_k, $\lambda_k(\mathbb{M}(t)) = t$. More formally,*

$$\lambda_k(\mathbb{M}(t)) = \begin{cases} \max(\{f, [s, f)_k \in \mathbb{M}(t)\}) & \textit{if } \exists\ [s, f)_k \in \mathbb{M}(t) \\ t & \textit{otherwise} \end{cases}$$

Definition A.3.2 (Continuous map and mapping)**.** *A processor map $\mathbb{M}_k(t)$ is continuous if any two consecutive frames of $\mathbb{M}_k(t)$ are adjacent. A map $\mathbb{M}(t)$ is continuous if for all $k \in [\![1, m]\!]$, $\mathbb{M}_k(t)$ is continuous. A mapping M_i is continuous if the resulting map $\mathbb{M}(t)$ is continuous.*

Definition A.3.3 (Valid Mapping and Map)**.** *A mapping M_i of a job J_i on Π is valid if the finish time of the latest frame of M_i is not later than the J_i's deadline. A map $\mathbb{M}(t)$ is valid if all its mappings are valid.*

Definition A.3.4 (Feasible Job)**.** *A job J_i is feasible on P at time t if there exists a valid mapping assignment M_i to J_i on Π and if the resulting map $\mathbb{M}(t)$ is a valid map.*

The history map $\mathbb{H}(t)$ of a system of map $\mathbb{M}(t)$ is the set of frames already assigned before t. Formally, $\mathbb{H}(t)$ is the history set of the system at t iff for all $t' < t$, $\mathbb{H}(t') = \mathbb{M}(t')$. Is is important to emphasize here that, in general, $\mathbb{H}(t)$ is not a subset of $\mathbb{M}(t)$. For example,

in Figure A.1, $\mathbb{H}(1) = \mathbb{M}(0)$ and $\mathbb{H}(1) \not\subset \mathbb{M}(1)$ because the free frame $[0, 2)_{2,2}$ of $\mathbb{H}(1)$ must be modified in order to assign a valid mapping of J_4. Consequently, $[0, 2)_{2,2} \in \mathbb{H}(1)$ but $[0, 2)_{2,2} \notin \mathbb{M}(1)$.

Regarding the incremental process of mapping assignment to jobs, we assume that mappings of two different jobs are not assigned simultaneously. When two or more jobs are mapped at time t, we define an order on jobs which is used in the map assignment process. In other words, the definition of $\mathbb{M}(t)$ is sequential regarding the jobs mapped at t. Hence, the building of a map $\mathbb{M}(t)$ is an incremental process, starting from $\mathbb{M}(t')$ for t' just before t and assigning, one after the other, in an established order, the mappings of ready jobs at time t. This process may modify, if necessary, all the free frames of $\mathbb{M}(t')$.

A.4 IDLE SERIALIZATION

Definition A.4.1 (Maximum and Minimum Idle Serialization Map and Mapping)**.** *Consider a continuous history map $\mathbb{H}(t)$, a subset Π' of Π ($\Pi' \subseteq \Pi$), and J_i a ready job, feasible but not yet mapped before t. Assume that the mapping M_i of J_i is continuous and that M_i is the only mapping assigned at time t. Finally, let $\mathbb{P}(t) = \bigcup_{P_k \in \Pi'} \mathbb{M}_k(t)$ be the resulting partial map and let $\theta(\mathbb{P}(t)) = \min(\{\lambda_k(t), P_k \in \Pi'\})$.*

The mapping M_i assigned to J_i results in a minimum or maximum idle serialized partial map $\mathbb{P}(t)$ iff any other continuous mapping assigned to J_i on Π' results in a partial map $\mathbb{P}'(t)$ such that $\theta(\mathbb{P}'(t)) \leqslant \theta(\mathbb{P}(t))$ or $\theta(\mathbb{P}'(t)) \geqslant \theta(\mathbb{P}(t))$, respectively. In such a case, M_i is a minimum or maximum idle serializing mapping (ISM) regarding Π', respectively.

Without loss of generality, we assume for the remaining sections that processors are ordered by non increasing order of their level. Also, when not specified, we consider that all frames are locked. In such a case, if M_i is the only mapping assigned to J_i before or at $t' > t$, then $\mathbb{M}(t') = \mathbb{H}(t) \cup M_i$.

Lemma A.4.1. *Consider a continuous and locked history map $\mathbb{H}(t)$ at time t, represented by the crosshatch regions in Figure A.2. Suppose that J_i is a ready job, feasible but not yet mapped at t. Let M_i be a valid continuous mapping of J_i assigned at $t' > t$ and assume that no other mapping is assigned during $[t, t']$. Finally, let Π_i be the set of processors on which some frame of M_i can be assigned at t, i.e. $\Pi_i = \{P_j \in \Pi, \lambda_j(t) < J_i.d\}$, and P_k be the processor of higher level of Π_i on which J_i is feasible at t, i. e. $\lambda_k = \max\{\lambda_j, P_j \in \Pi_i \wedge \lambda_j(t) + e(J_i, t) \leqslant J_i.d\}$. The following properties hold:*

***Maximum ISM**: (i) If $\lambda_k \neq \max\{\lambda_j, P_j \in \Pi_i\}$, then the mapping $M_i = \{F_{k-1,i}, F_{k,i}\}$ with $F_{k-1,i} = [\lambda_{k-1}(t), J_i.d)_{k-1,i}$ and $F_{k,i} = [\lambda_k(t), \lambda_k(t) + e(J_i, t') - (J_i.d - \lambda_{k-1}(t))_{k,i}$ is a maximum ISM of J_i regarding Π. Moreover, M_i is a maximum ISM of J_i regarding*

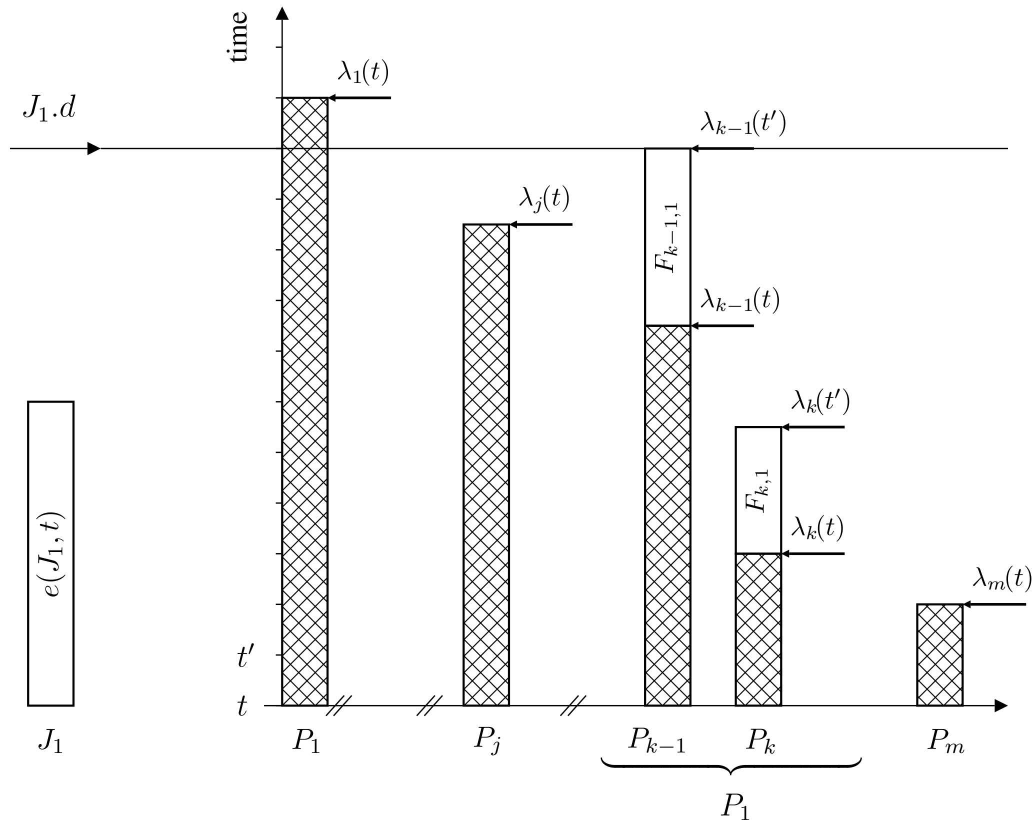


Figure A.2. The history map $\mathbb{H}(t)$ (crosshatch region) and the mapping $M_i(t)$ of J_i. Case $\lambda_k(t) \neq \max\{\lambda_j(t), P_j \in \Pi_i\}$ $(\forall l \in [\![1,m]\!]\backslash\{k-1,k\}, \lambda_l(t) = \lambda_l(t'))$

$\{P_{k-1}, P_k\}$.

(ii) If $\lambda_k = \max\{\lambda_j, P_j \in \Pi_i\}$*, then the mapping* $M_i = \{F_{k,i}\}$ *with* $F_{k,i} = [\lambda_k(t), \lambda_k(t) + e(J_i,t'))_{k,i}$ *is a maximum ISM of* J_i *regarding* Π*. Moreover,* M_i *is a maximum ISM of* J_i *regarding* P_k *and any other processor of* Π.

Minimum ISM*: The mapping* $M_i = \{F_{m,i}\}$ *with* $F_{m,i} = [\lambda_m(t), \lambda_m(t) + e(J_i,t'))_{m,i}$ *is a minimum ISM of* J_i *regarding* Π.

*Proof.*As $\mathbb{H}(t)$ is a locked map, $\mathbb{M}(t') = \mathbb{H}(t) \cup M_i$.

Maximum ISM: (i) This is the case illustrated by Figure A.2. We first prove that M_i is a maximum ISM of J_i regarding $\{P_{k-1}, P_k\}$. Let $\mathbb{P}(t') = \mathbb{M}_{k-1}(t') \cup \mathbb{M}_k(t')$. By the definition (i) of M_i, $\theta(\mathbb{P}(t')) = \lambda_k(t) + e(J_i,t') - (J_i.d - \lambda_{k-1}(t))$. The continuous mapping M_i assigns to J_i the frame $F_{k-1,i}$ of maximum length in the sense that the assignment of any longer frame would produce a non valid mapping of J_i. Consequently, any other valid continuous mapping of J_i on P_k and P_{k-1} would assign to J_i a shorter frame than $[\lambda_k(t), J_i.d)_{k-1}$ on P_{k-1} and a longer frame than $[\lambda_k(t), e(J_i,t') - (J_i.d - \lambda_{k-1}(t)))_k$ on P_k, resulting in an increase in the idle time on P_{k-1} and a decrease in the idle time on P_k. Thus, according to Definition A.4.1,

M_i is the maximum ISM of J_i regarding $\{P_{k-1}, P_k\}$.

To prove that M_i is the maximum ISM of J_i regarding Π, we distinguish two cases. First, if $k \neq m$, then $\theta(\mathbb{M}(t')) = \lambda_m(t)$ and M_i is a maximum ISM regarding Π. Second, if $k = m$, then $\lambda_m(t') = \lambda_m(t) + e(J_i, t') - (J_i.d - \lambda_{m-1}(t))$ and we have $\min\{\lambda_j(t'),\ j \in [\![1, m]\!]\} = \lambda_m(t')$ because $\lambda_m(t') < \lambda_{m-1}(t)$, by the definition of M_i. Hence, $\theta(\mathbb{M}(t')) = \lambda_m(t')$ which proves that, in this case also, M_i is a maximum ISM regarding Π.

(ii) Let P_j be a processor different from P_k. We must prove that M_i is a maximum ISM regarding $\{P_j, P_k\}$. However, as J_i is feasible on P_k, the mapping M_i defined by (ii) is valid. Moreover, as M_i defined by (ii) is the only continuous mapping of J_i on the single processor P_k, the assignment of part of the execution time of J_i to another frame $F_{j,i}$ on $P_j \neq P_k$ would result in an later idle time on P_j. Thus, according to A.4.1, M_i is the maximum ISM of J_i regarding $\{P_j, P_k\}$.

Minimum ISM: Here again, we distinguish two cases. First, if $\lambda_m(t) + e(J_i, t'))_{m,i} \leqslant \lambda_{m-1}(t)$, then $\theta(\mathbb{M}(t') = \lambda_m(t')$. Thus, any other mapping of J_i would result in an earlier idle time on P_m. Otherwise, $\theta(\mathbb{M}(t')) = \lambda_{m-1}(t)$. However, as M_i is continuous, the only frame that could be assigned to P_{m-1} by another mapping of J_i would be $[\lambda_{m-1}(t), \lambda_{m-1}(t) + e(J_i, t') - (\lambda_{m-1}(t) - \lambda_m(t)))_{m-1,i}$. But, such an assignment would result in the same value of $\theta(\mathbb{M}(t'))$ that would be achieve on P_m instead of P_{m-1}. This establishes the Lemma. □

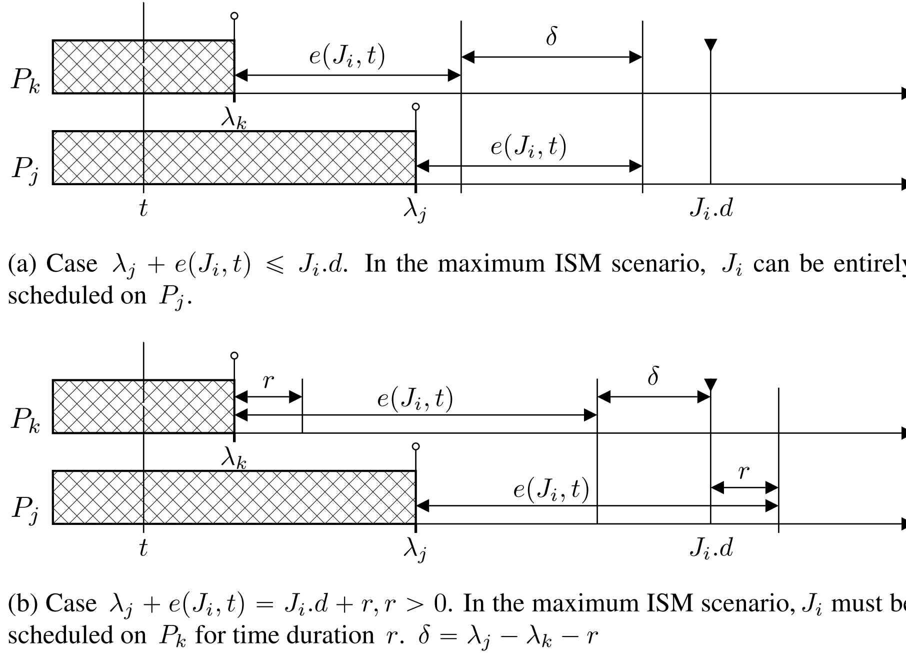
Given an history map at time t, Lemma A.4.1 characterizes which different mappings of a job achieved minimum and maximum idle serialization. The next lemma quantifies the differences between each of this mappings in terms of idle serialization.

Lemma A.4.2. *Consider the minimum and maximum ISM of a ready job $J_i \in \mathbb{Q}(t)$ on two processors P_j and P_k at time t. The length δ of the idle time that happens earlier in the maximum ISM of J_i than in the minimum ISM of J_i is $\delta = \min(e(J_i, t), (\lambda_j - \lambda_k) - \max(0, e(J_i, t) - (J_i.d - \lambda_j(t)))$.*

*Proof.*This is a consequence of the Definition of maximum and minimum ISM, as illustrated in Figure A.3. Note that we must distinguish whether $\lambda_j + e(J_i, t) \geqslant J_i.d$ or not. □

A.5 ON-LINE SCHEDULING

We enunciate here the criterion of an on-line scheduling policy for the **I**dle **S**erialization **Ba**sed (ISBa) scheduling algorithm. When a job $J_i \in \mathbb{Q}(t)$ is considered for mapping at time t, ISBa needs to choose between the maximum or minimum idle serialization mapping. However each of this two choices has consequences. Choosing the maximum ISM scenario may make a ready job feasible, taking advantage of the full length of the longest idle time. On



(a) Case $\lambda_j + e(J_i, t) \leqslant J_i.d$. In the maximum ISM scenario, J_i can be entirely scheduled on P_j.

(b) Case $\lambda_j + e(J_i, t) = J_i.d + r, r > 0$. In the maximum ISM scenario, J_i must be scheduled on P_k for time duration r. $\delta = \lambda_j - \lambda_k - r$

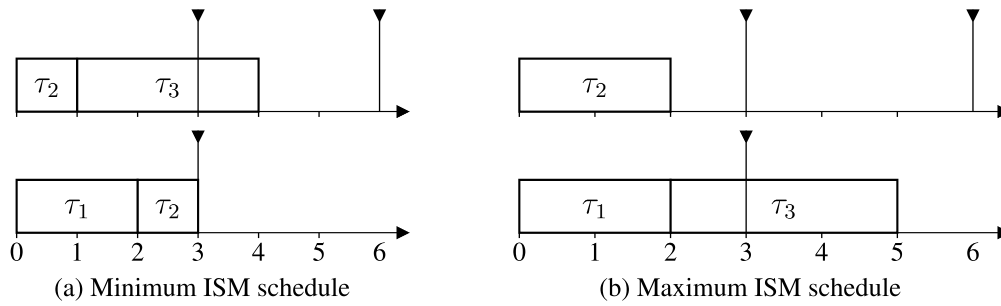
Figure A.3. Idle serialization comparison between minimum and maximum ISM schedules. In the minimum ISM scenario, J_i is scheduled on P_k while in the maximum ISM, J_i is scheduled on P_j.

the other hand, choosing the minimum ISM may make two jobs with low laxity feasible, yet to be released.

For instance, let us consider the simple job set J_1:$(0, 2, 3)$, J_2:$(0, 2, 3)$ and J_3:$(0, 3, 6)$, ordered by non decreasing laxity. At time $t = 0$, J_3 is ready and the minimum ISM scenario can be chosen, resulting in Figure A.4. Doing so, the schedule of two jobs J_4:$(3, 2, 6)$ and J_5:$(3, 2, 6)$ becomes feasible. On the other hand the schedule of a job J_4:$(2, 4, 6)$ would only be feasible if the maximum ISM scenario were chosen at time 0. Such impossibility to make the right choice for all scenarios is in strong agreement with the result of Dertouzos (DERTOUZOS; MOK, 1989) which states that no optimal multiprocessor scheduling algorithm exists in the general sporadic job model.

This simple example illustrates the guide-lines that we have adopted for the ISBa algorithm. While no ready jobs can execute thanks to the idle serialization, ISBa chooses the minimum idle serialization schedule. Otherwise, ISBa opts for the maximum idle serialization schedule. In other word, ISBa only chooses a maximum ISM schedule when this choice does not cause the idling of a processor. Otherwise, ISBa chooses the minimum ISM schedule.

We have successfully implemented the ISBa algorithm. However, after more than a year of intensive work, the obtained results were disapointing since ISBa was only capable to schedule about the same number of fully-utilization task sets as EDZL when using random task set generated by the open-source random task generator developed by Emberson (EMBERSON et al., 2010; EMBERSON et al., 2011). Since the ISBa implementation was much more complicated



(a) Minimum ISM schedule

(b) Maximum ISM schedule

Figure A.4. The minimum ISM schedule turns J_4:$(3,\ 2,\ 6)$ and J_5:$(3,\ 2,\ 6)$ feasible and J_4:$(2,\ 4,\ 6)$ unfeasible, while the maximum ISM schedule turns J_4:$(3,\ 2,\ 6)$ and J_5:$(3,\ 2,\ 6)$ unfeasible and J_4:$(2,\ 4,\ 6)$ feasible.

than that of EDZL, we conclude that, in general, they were no gain in using the idle serialization approach.

Appendix

B

Elegance likes shortness. Beauty do not necessarily.

EDF SERVER THEOREM: ANOTHER PROOF

In order to give a direct proof of Theorem 3.4.1, we first present some intermediate results.

B.1 SCALING

Definition B.1.1. *Let S be a server, Γ a set of servers with $\rho(\Gamma) \leqslant 1$, and α a real such that $0 < \alpha \leqslant 1/\rho(S)$. The α-scaled server of S, denoted αS, is the server with utilization $\alpha\,\rho(S)$ and deadlines equal to those of S. The α-scaled set of Γ is the set of the α-scaled servers of all servers in Γ.*

As illustration, consider $\Gamma = \{S_1, S_2, S_3\}$ a set of servers with $\rho(S_1) = 0.1$, $\rho(S_2) = 0.15$, $\rho(S_3) = 0.25$ and $\rho(\Gamma) = 0.5$. The 2-scaled set of Γ is $\Gamma' = \{S'_1, S'_2, S'_3\}$ with $\rho(\Gamma') = 1$, $\rho(S'_1) = 0.2$, $\rho(S'_2) = 0.3$ and $\rho(S'_3) = 0.5$.

Lemma B.1.1. *Let Γ be a set of EDF servers with $\rho(\Gamma) \leqslant 1$. Consider the EDF servers S and S' associated to Γ and Γ' where Γ' is α-scaled set of Γ and let Σ and Σ' are their corresponding schedules, respectively. Then Σ is valid if and only if Σ' is valid.*

*Proof.*Suppose that Σ is valid. Consider a deadline d in $\mathrm{R}(S)\backslash\{0\}$. Since S and S' use EDF and $\mathrm{R}(S) = \mathrm{R}(S')$, S and S' execute their client jobs in the same order. As a consequence, all the executions of servers in $\mathrm{cli}(S)$ during $[0, d)$ must have a corresponding execution of a server in $\mathrm{cli}(S')$ during $[0, d)$.

Also, since S executes for $\rho(S)d$ during $[0, d)$ and $\alpha \leqslant 1/\rho(S)$, the execution time $\rho(S')d$ of S' during $[0, d)$ satisfies $\alpha\,\rho(S)d \leqslant d$. Hence, a client job of S' corresponding to

an execution which completes in Σ before d, completes before d in Σ'. Hence, since Σ is valid, so is Σ'.

The converse also follows from the above argument, using a scale factorale equal to $\alpha' = 1/\alpha$. □

B.2 DIRECT PROOF OF THE EDF SERVER THEOREM

The proof presented now of Theorem 3.4.1 is an adaptation of the proof of Theorem 7 from (LIU; LAYLAND, 1973). Since our server model is a generalization of the PPID task model, this direct proof does not use more recent results established for this model.

Lemma B.2.1. *The unit EDF server $S = \mathrm{ser}(\Gamma)$ of a set of synchronous servers Γ with $\rho(\Gamma) = 1$ produces a valid schedule of Γ if all jobs of S meet their deadlines.*

*Proof.*We proceed by contradiction.

Assume that there exists an instant D in $\mathrm{R}(S)$ at which a deadline miss occurs for a budget job J of some client server of S in Γ. Also, without loss of generality, assume that no deadline miss occurs before D *i.e.,* J is the first job after time $t = 0$ which misses its deadline at time $D = J.d$.

We define t' as the start time of the latest idle time interval before $J.d$ if such idle time exists and $t' = 0$ otherwise. Consider D' the earliest deadline in $\mathrm{R}(S)$ after or at t'. It must be that $D' < D$ otherwise no job of server in Γ would be released between t' and D, contradicting the fact that J misses its deadline at time D.

If D' is not equal to zero, then the processor must be idle during $[t', D')$. Indeed, if there were some job J' executing just before D', it would have been released after t' since t' is the start time of an idle time. Consequently, the release instant of J' would be a deadline in $\mathrm{R}(S)$ occurring before D' and after t', which would contradict the definition of D'.

We now show that the total demand of servers in Γ within interval $[D', D)$ is not greater than $D - D'$, reaching a contradiction, since no idle time exists within $[D', D)$. There are two cases to be distinguished depending on whether some lower priority server executes within $[D', D)$.

Case 1

Illustrated by Figure B.1. Assume that no job of servers in Γ with lower priority than J executes within $[D', D)$. Since there is no processor idle time within $[D', D)$ and a deadline miss occurs at time D, it must be that the accumulated execution time of all budget jobs in Γ

released at or after D' and with deadline less than or equal to D is strictly greater than $D - D'$.

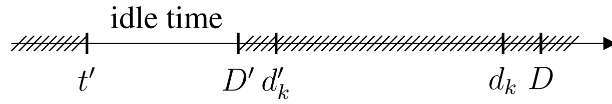


Figure B.1. A deadline miss occurs for job J at time D and no job with lower priority than J executes before D

Now, consider a server S_k in Γ whose budget jobs have their release instants and deadlines within $[D', D)$. Let d'_k and d_k be the first release instant and the last deadline of such jobs, respectively. Since the processor is idle before D', a job of S_k released before D' must have completed before D'. Also, the job of S_k released at time d_k has lower priority than J and does not contribute to the workload necessarily executed before J. Hence, the demand $\eta_\Gamma(D', D)$ of servers in Γ which prevents J's execution during $[D', D)$ is

$$\eta_\Gamma(D', D) = \sum_{S_k \in \Gamma} \rho(S_k)(d_k - d'_k)$$

As $d_k - d'_k \leqslant D - D'$ for all S_k in Γ and $\sum_{S_k \in \Gamma} \rho(S_k) = \rho(S) = 1$, we deduce that

$$\eta_\Gamma(D', D) \leqslant \rho(S)(D - D') \leqslant D - D'$$

On the other hand, the accumulated budget of S during $[D', D)$ is precisely equal to $D - D'$ since all jobs of S meet their deadlines and S is a unit server. It follows that no deadline miss can occur during $[D', D)$ since the total demand of jobs of servers in Γ during $[D', D)$ is no greater than the accumulated budget available for their execution during $[D', D)$, leading to a contradiction.

Case 2

Illustrated by Figure B.2. Assume now that there exist some budget jobs of servers in Γ with lower priority than J that execute within $[D', D)$. Let D'' be the earliest deadline in $[D', D)$ after which no such job execute and consider $J.r$ the release instant of J. Since J misses its deadline, no job with lower priority than J can execute after $J.r$. Thus, we must have $D'' \leqslant J.r < D$. Also, there is no processor idle time within $[D'', D)$. Thus, for a deadline miss to occur at time D, it must be that the accumulated execution time of all servers in Γ released at or after D'' and with deadline less than or equal to D is strictly greater than $D - D''$.

Now, it must be that a lower priority job was executing just before D''. Indeed, if J', a job with higher priority than J, was executing just before D'', its release time $J'.r$ would be

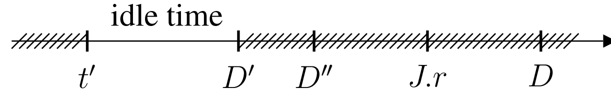


Figure B.2. A deadline miss occurs for job J at time D and some lower priority job than J executes before D

before D'' and no job with lower priority than J could have executed after $J'.r$, contradicting the definition of D''. Thus, no job released before D'' and with higher priority than J executes between D'' and D.

Hence, the demand $\eta_\Gamma(D'', D)$ of servers in Γ which prevents J's execution during $[D'', D)$ is

$$\eta_\Gamma(D'', D) = \sum_{S_k \in \Gamma} \rho(S_k)(d_k - d''_k)$$

where d'_k and d_k are the first release instant and the last deadline of jobs with release instants and deadlines within $[D', D)$, respectively. Thus,

$$\eta_\Gamma(D'', D) \leqslant \rho(S)(D - D'') \leqslant D - D''$$

As previously, the accumulated budget of S during $[D'', D)$ is precisely equal to $D - D''$ since all jobs of S meet their deadlines and S is a unit server. Henceforth, the accumulated execution time of all servers during $[D'', D)$ in not greater than $D - D''$, the available budget of S and no deadline miss can occur, reaching a contradiction. □

Finally, the combination of Lemma B.1.1 with Lemma B.2.1 permits to complete the direct proof of Theorem 3.4.1.

*Proof.*Consider a set of servers $\Gamma = \{S_1, S_2, \dots, S_n\}$ such that $\rho(\Gamma) \leqslant 1$ and assume that Γ is to be scheduled by an EDF server S. Let Γ' be the $1/\rho(\Gamma)$-scaled server set of Γ.

By Definition B.1.1, $(\Gamma') = \sum_{i=1}^{n} \rho(S_i)/\rho(\Gamma) = 1$. Hence, by Lemma B.1.1, the schedule Σ of Γ by S is valid if and only if the schedule Σ' of Γ' by $S' = \mathrm{ser}(\Gamma')$ is valid. Since, by Lemma B.2.1, the schedule Σ' produced by unit server S' is valid, we deduce that so is Σ. □

Appendix

C

Why shall one use a complex solution whenever a simple exists?

X-RUN: A PROPOSAL FOR SPORADIC TASKS

In this appendix, we discuss some of our ideas to extend RUN to the sporadic task model with implicit deadlines. Since none of the material presented here is confirmed by theoretical proofs or simulation results, we can not guarantee its correctness. However, we believe that an optimal solution for scheduling sporadic task systems with implicit deadlines should emerge soon from this documented discussion.

C.1 TASK MODEL

We consider a *sporadic* task model with implicit deadline, further referred as STID model. According to this model, two jobs of a task τ_i of period T_i are separated by at least T_i. That is, T_i is the minimum inter-arrival time between any two jobs of task τ_i. Formally, if J_k and J_{k+1} are two consecutive jobs of task τ_i, then $J_{k+1}.r - J_k.r \geqslant T_i$. Note that, since we assume implicit deadlines, for any job J_k of a task τ_i, we have $J_k.d = J_k.r + T_i$.

We say that a server S_i is *active* whenever there are one or more client's jobs of S_i ready to execute. Otherwise, we say that S_i is *idle*.

C.2 RUN SUBTREE

One of the key idea we want to present here for the extension of RUN to the STID model is based on the concept of subtree that we recall now.

As stated by Definition 5.3.1, a RUN *subtree* of a general RUN tree is comprised of a single grandparent server, referred to as root server of the subtree, together with its child servers and grandchild servers.

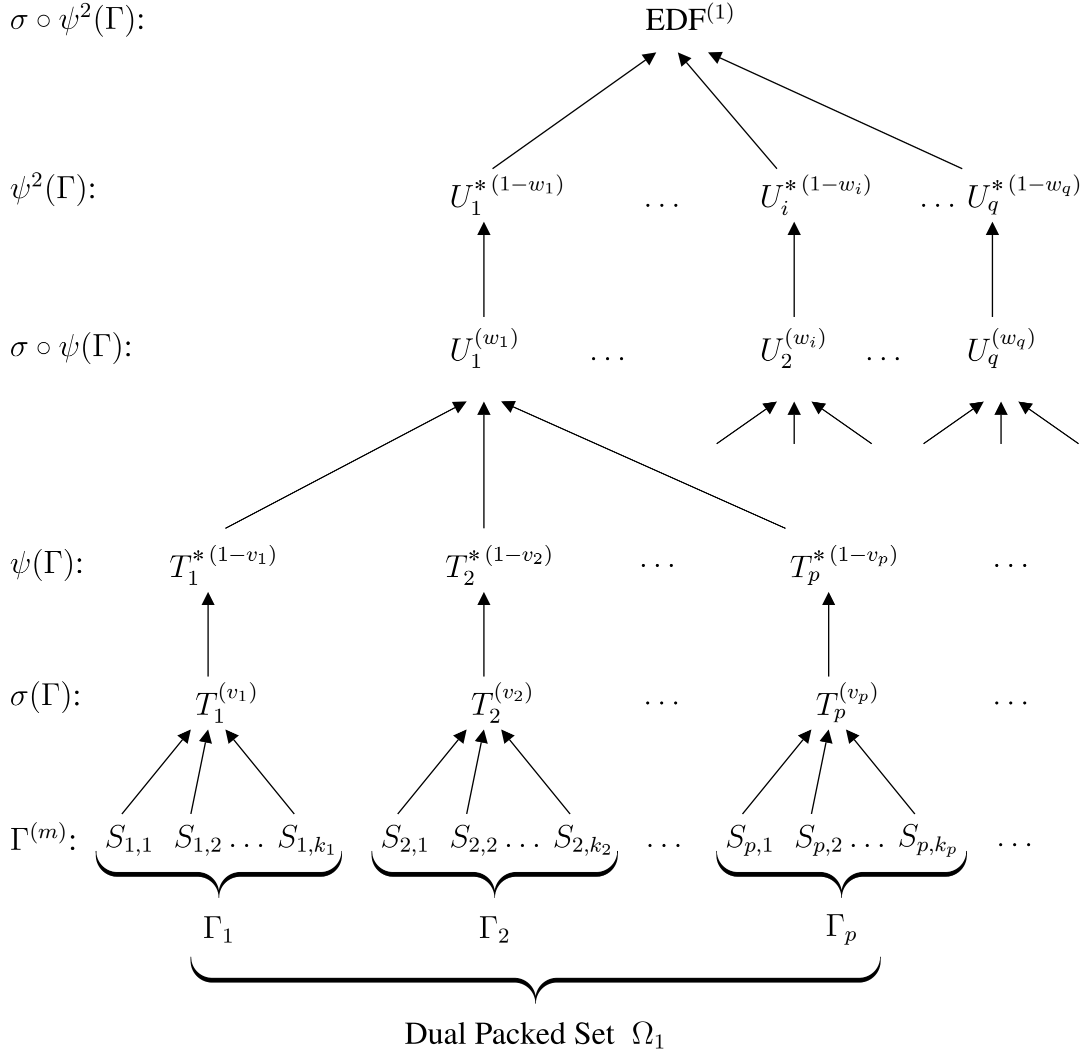


Figure C.1. RUN subtree. U_1 is the root server, $\{T_i\}_i$ is the collection of child servers, and $\{S_{i,j}\}_{i,j}$ is the collection of grandchild servers. Moreover, $\rho(\Omega_1) = p - 1 + \rho(U_1)$.

Figure C.1, reproduced from Figure 5.7, shows an example of RUN *subtree* of a general RUN tree. In this figure, U_1^* is the grandparent root server, $\{T_i^*\}_i$ is the collection of child servers of U_1, and $\{S_{i,j}\}_{i,j}$ is the collection of grandchild servers of U_1.

We recall now the Definition 5.3.2 of a dual-packed set and the associated lemma 5.3.1, since our proposal for sporadic task scheduling is built upon both.

Definition C.2.1 (Dual-Packed Set)**.** *Let* Γ *be a set of servers and* $\pi[\Gamma] = \{\Gamma_1, \Gamma_2, \dots, \Gamma_p\}$ *be the packing of* Γ *by a packing algorithm* A*. The packing of* $\psi(\Gamma)$ *by* A *defines a partition of* $\pi[\Gamma]$ *into a family of* dual-packed set *(of server set), denoted* $\{\Omega_k\}_k$*, such that for all* $\Gamma_i, \Gamma_j \in \Omega_k$*, if* $\Gamma_i \neq \Gamma_j$ *then* $\psi(\mathrm{ser}(\Gamma_i)) = \psi(\mathrm{ser}(\Gamma_j))$ *for all* k*,* $1 \leqslant k \leqslant |\psi(\Gamma)|$.

If $\Omega_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ is a dual packed set of the reduction tree of a set of servers Γ, then for all $S_i, S_j \in \bigcup_{\Gamma_k \in \Omega_1} \Gamma_k$, $\psi^2(S_i) = \psi^2(S_j)$. In other words, all the grandchild servers in the set of servers in Ω_1 have the same grandparent server $S = \psi^2(S_i)$. Thus, $\bigcup_{\Omega_1} \Gamma_i$ is the set of all grandchild servers of the subtree with root server S.

Lemma C.2.1 (Parallel Execution Requirement)**.** *Let Γ be a set of servers and $\pi[\Gamma] = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_p\}$ be the packing of Γ by a packing algorithm A. Consider $\Omega_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ a dual packed set with $k > 1$ and let $U_1^* = \psi^2(S_{i,j})$ for some server $S_{i,j}$ in Γ_j and Γ_j in Ω_1. Then, there exists a real number x, called* excess*, with $0 \leqslant x < 1$ such that $\rho(\Omega_1) = p-1+x$ where $p = |\Omega_1|$. Moreover, $\rho(U_1^*) = x$. Excess x represents the amount of parallel execution required by Ω_1.*

Recall that $\rho(\Omega_1) = p - 1 + x$ means that a dual-packed set can be scheduled on $|\Omega_i| - 1$ fully utilized processors and one partially utilized processor with rate x.

C.3 X-RUN: SWITCHING APPROACH

We assume here that the general RUN tree is divided into distinct subtrees and we discuss our ideas for the X-RUN algorithm development considering a single subtree, as illustrated by Figure C.1. Note that there are $p - 1$ full processors and fraction x of another processor associated with this subtree at the grandchild server level.

Our first key idea is to only use the RUN scheme when it is strictly necessary, *i.e.,* whenever there exists some parallel execution requirement at the grandchild server level of the subtree. Otherwise, we believe that any work-conserving scheduling policy (WCS) is sufficient to correctly schedule the grandchildren in the subtree. More precisely, whenever all child servers T_i in the subtree are active, then we use the RUN algorithm to generate their schedule. This corresponds to the usual behavior of RUN since, if the p servers T_i are active, there exists a parallel execution requirement which must be handled by root server U_1^*.

Otherwise, if one (or more) child server T_i is idle, then $p - 1$ (or less) child servers T_j are active, for $j \neq i$. Since there are $p - 1$ processors available in the subtree, we can simply schedule those active servers using a WCS policy *i.e.,* scheduling the active servers on the available processors.

According to this switching policy, the X-RUN algorithm, restricted to one subtree, would alternate between RUN windows and WCS windows, as illustrated in Figure C.2.

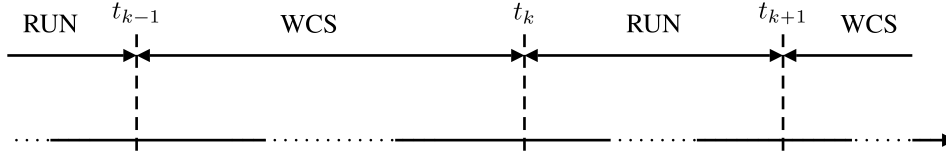


Figure C.2. Switching between WCS windows and RUN windows for a subtree. At switching instant t_k, all servers T_i for $1 \leqslant i \leqslant p$, are or become active.

C.4 X-RUN: BUDGET ESTIMATION

Although the switching idea seems simple, it requires solving the following non-trivial problem. How do we estimate the budgets of the child, grandchild servers and root server at a switching instant between a WCS window and a RUN window?

To answer this question, we begin by noting that, during a WCS window, there is no need to update the budget of child and grandparent servers of the subtree, since they are not used by the WCS policy. Hence, during a WCS window, we just need to update the execution time of each server execution at the grandchild level. As a consequence, the budget of an active grandchild server S_i can be estimated straightforwardly at a WCS-to-RUN switching instant t_k. If S_i releases a job J at time t_k *i.e.,* if $J_i.r = t_k$, then the budget of S_i at time t_k is given by $e(J_i, t_k) = \rho(S_i)(J_i.d - J_i.r)$, as defined in Section 3.4.1 . Otherwise, if $J_i.r < t_k$, then the budget of S_i at time t_k is simply the remaining execution time of S_i at time t_k, *i.e.,* $e(J_i, t_k) = \rho(S_i)(J_i.d - J_i.r) - (t_k - J_i.r)$. Note that this latter quantity can not be negative, since this would imply that S_i is not active at t_k.

C.4.1 Weighting Approach

Let us now describe our proposal to estimate the child server budgets. For this purpose, we consider a generic situation comprised of a WCS window $I_b = [t_2, t_3)$, in between two RUN windows $I_a = [t_1, t_2)$ and $I_c = [t_3, t_4)$ as shown in Figure C.3.

Let J_S be a job of a grandchild server S with release instant $J_S.r$. We denote by $W_i(S, t)$ the contribution to the budget of a dual child server T_i^* caused by job J_S at time t and we proceed using an induction reasoning over release instants during a RUN window. We first assume that all budgets are correctly estimated until t_3, inclusive at t_3, and we define the budget replenishment policy for a child server T_i during (t_3, t_4) as follows.

Replenishment at a non-switching instant

Let J_S be a job of a grandchild server S with release instant $J_S.r$ such that $t_3 < J_S.r < t_4$. That is, the release instant of J_S happens in a RUN window but is not a WCS-to-RUN switching

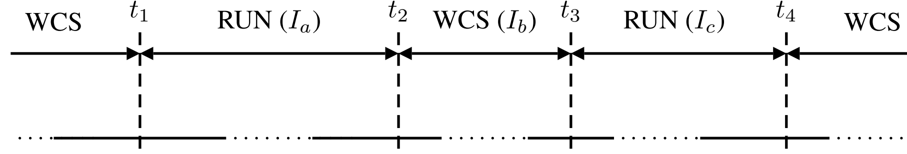


Figure C.3. WCS window I_b of length $t_3 - t_2$ in between two RUN windows I_a and I_c.

instant. If S is the single client of T_i, then we calculate the budget of T_i^* as RUN would, *i.e.,* $e(T_i^*, J_S.r) = \rho(T_i^*)(J_S.d - J_S.r)$, as seen in Section 3.4.1. That is, $W_i(S, J_S.r) = \rho(T_i^*)(J_S.d - J_S.r)$.

Now, suppose that T_i has more than one client. In such a case, the replenishment rule must be modified. Indeed, suppose that all other client of T_i are idle at the release instant of S's job. At time $J_S.r$, the workload of T_i is only generated by J_S, proportionally to $\rho(S)$. As a matter of fact, we could write

$$e(T_i, J_S.r) = \rho(T_i)\frac{\rho(S)}{\rho(T_i)}(J_S.d - J_S.r)$$

in order to represent the fact that, among the total budget of T_i, J_S contributes for a ratio $\rho(S)/\rho(T_i)$.

In a similar manner, the contribution workload $W_i(S, J_S.r)$ caused by J_S to T_i^* should also be proportional to $\rho(S)$. Hence, we estimate this contribution as the total contribution $\rho(T_i^*)(J_S.d - J_S.r)$ that would exist if S were the only client of T_i, multiplied by the ratio $\rho(S)/\rho(T_i)$. More precisely, we add to the remaining budget of T_{i^*} at time $J_S.r$ the quantity

$$W_i(S, J_S.r) = \rho(T_i^*)\frac{\rho(S)}{\rho(T_i)}(J_S.d - J_S.r) \tag{C.1}$$

Observe that if server S is the only client of T_i, then $\rho(S)/\rho(T_i) = 1$ and $W_i(S, J_S.r)$ as given by Equation C.1 precisely equals the RUN budget estimation as defined in Section 3.4.1. However, when T_i is comprised of many small rate servers, then the amount of dual budget added for each job released by a child server of T_i is proportional to its participation in the accumulated rate of T_i.

Replenishment at a WCS-to-RUN switching instant

We define now the budget replenishment policy for a child server T_i at the WCS-to-RUN switching instant t_3.

First, consider a server S of T_i which releases a job J_S before t_3 with deadline before t_3. The workload contribution caused by J_S to T_i^*'s budget has deadline $J_S.d < t_3$, hence it

should not contribute to T_i^*'s budget a time t_3.

Hence, we calculate $e(T_i^*, t_3)$ considering only the contributions of T_i's clients with deadline after t_3. Let J_S be a job of a grandchild server S with deadline $J_S.d$ such that $J_S.d > t3$. We distinguish three different cases according to the release instant of $J_S.d$.

***Case 1**:* $J_S.r = t_3$

Since we assume that Equation C.1 is used for any instant arbitrarily close to and greater than t_3, $W_i(S, J_S.r)$ with $J_S.r = t_3$ must tend to $W_i(S, J_S.r)$ when $J_S.r$ is strictly greater than t_3 and tends to t_3. Hence, we also use Equation C.1 when $J_S.r = t_3$, for the sake of continuity of $W_i(S, t)$ as a function of $J_S.r$.

***Case 2**:* $t_2 \leqslant J_S.r < t_3$

Here, we observe that, when $J_S.d > t_3$ tends to t_3, then $W_i(S, J_S.r)$ must tend to zero, since the contribution of a job with deadline not greater than t_3 is zero. Thus, we can think of $W_i(S, J_S.r)$ proportional to $J_S.d - t_3$. Moreover, for the sake of continuity of $W_i(S, t_3)$ as a function of $J_S.r$, we propose the following estimation

$$W_i(S, t_3) = \rho(T_i^*) \frac{\rho(S)}{\rho(T_i)} (J_S.d - t_3) \tag{C.2}$$

since it tends to the estimation given by Equation C.1 when $J_S.r$ tends to t_3.

***Case 3**:* $J_S.r \leqslant t_2$

In this later case, we must consider the remaining budget of T_i^* at time t_2. To convince ourselves of the pertinence of this point, we use again a continuity argument. If window I_b in Figure C.3 becomes arbitrarily short, then, the budget of T_i^* at time t_3 must tend to its budget at time t_2. That is, $e(T_i^*, t_3)$ must tend to $e(T_i^*, t_2)$ when $t_3 - t_2$ tends to zero.

Let $A(t_2)$ be the set of all client of T_i which release jobs before t_2 with deadlines after t_3, *i.e.,* $A(t_2) = \{S \in \text{cli}(T_i), S \text{ releases a job } J_S \text{ with } J_S.r < t_2 \text{ and } J_S.d > t_3\}$.

For $t_3 > t_2$, the remaining budget $e(T_i^*, t_2)$ should have been consumed during I_b for an amount equal to $\rho(A(t_2))(t_3 - t_2)$. Since this amount is possibly greater than $e(T_i^*, t_2)$, we deduce that the contribution of the client jobs of servers in $A(t_2)$ released before t_2 to the dual workload $e(T_i^*, t_3)$ of T_i^* at time t_3 equals $max\{0, e(T_i^*, t_2) - \rho(A(t_2))(t_3 - t_2)\}$.

Let $B(t_3)$ be the set of all clients of T_i which release jobs before t_2 with deadlines after t_3, *i.e.,* $B(t_3) = \{S \in \text{cli}(T_i), S \text{ releases a job } J_S \text{ with } t_2 \leqslant J_S.r < t_3 \text{ and } J_S.d > t_3\}$. We finally

obtain the following proposal for the estimation of T_i^*'s budget at the WCS-to-RUN switching instant t_3:

$$\begin{aligned} e(T_i^*, t_3) &= max\{e(T_i^*, t_2) - \rho(A(t_2))(t_3 - t_2)\} + \sum_{S \in B(t_3)} W_i(S, t_3) \\ &= max\{e(T_i^*, t_2) - \rho(A(t_2))(t_3 - t_2)\} + \sum_{S \in B(t_3)} \rho(T_i^*)\frac{\rho(S)}{\rho(T_i)}(J_S.d - t_3) \end{aligned}$$

C.4.2 Horizon Approach

In order to complete the picture of our proposal for the X-RUN algorithm, we must establish the replenishment policy for the root server of a subtree.

For this purpose, we define the horizon $h(U^*, t)$ of a root server U^* of a subtree as the earliest possible deadline among the jobs already active at time t or yet to be released after t. Indeed, a grandchild server S, idle at time t, can release a job J_S at any time after t. Then, at time $J_S.r$, the deadline $J_S.d$ would become the earliest deadline in the system. Thus, if the budget of the root server U^* had been estimated at time t using only the earliest inherited deadline from the active servers at time t, then, at time $J_S.r$, the earlier deadline $J_S.d$ would decrease the U^*'s budget. Moreover, the budget estimated at time t could have been already consumed at time $J_S.r$, resulting in a possibly negative budget of U^*.

We prevent such event to happen by only replenishing the budget of root server U^* until its horizon, *i.e.,*, at a replenishment instant of U^*, we estimate its budget

$$e(U^*, t) = \rho(U^*)(h(U^*, t) - t)$$

Also, the next replenishment instants of U^* after t is the earliest instant between $h(U^*, t)$ and the next release instant of a U^* 's grandchild job.

This last equation completes what we think that RUN must look like in order to cope with the STID model. As mentioned before, this piece of work must still be and implemented and proved correct.