l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Muhammad Naeem SHEHZAD

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

Discipline : Informatique
Spécialité : Automatique et Informatique Appliquée
Laboratoire : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCYN)

Soutenue le 27 March 2013

École doctorale : 503 (STIM)
Thèse n° : 000000000

Overhead control in optimal global scheduling algorithms for real-time multiprocessor systems

JURY

Rapporteurs :	**M. Michel AUGUIN**, Directeur de recherche CNRS , LEAT, Sophia Antipolis **M. Pascal RICHARD**, Professeur des Universités, Université de Poitiers(LIAS)
Examinateurs :	**M. Pierre-Emmanuel HLADIK**, Maître de Conférences, INSA de Toulouse(LAAS) **M. Mathieu JAN**, Ingénieur-Docteur, CEA Saclay Nano-INNOV **M^me^ Anne-Marie DEPLANCHE**, Maître de Conférences, Université de Nantes(IRCCYN) **M. Olivier PASQUIER**, Maître de Conférences, Université de Nantes (IETR) (invité) **M. Yvon TRINQUET**, Professeur des universités, Université de Nantes(IRCCYN)
Directeur de thèse :	**M. Yvon TRINQUET**, Professeur des universités, Université de Nantes(IRCCYN)
Co-encadrante de thèse :	**M^me^ Anne-Marie DEPLANCHE**, Maître de Conférences, Université de Nantes(IRCCYN)

Contents

1 General introduction **15**

2 Real-time multiprocessor system scheduling **21**
2.1 Introduction . . . 22
2.2 Workload . . . 23
2.3 Processing unit . . . 24
2.4 Scheduling policy . . . 26
2.4.1 Uniprocessor scheduling . . . 27
2.4.2 Multiprocessor scheduling . . . 28
2.4.2.1 Partitioned scheduling . . . 32
2.4.2.2 Global scheduling . . . 36
2.4.3 Hybrid scheduling . . . 38
2.5 Conclusion . . . 39

3 A survey of optimal global algorithms **41**
3.1 Introduction . . . 42
3.2 Proportionate Fair scheduling . . . 42
3.2.1 Principle . . . 42
3.2.2 PFair scheduling . . . 46
3.2.3 PFair algorithms . . . 48
3.2.4 PFair implementation . . . 49
3.2.5 Limitations of PFair . . . 49
3.2.6 Extensions of PFair . . . 53
3.2.6.1 Early Release Fair Model . . . 53
3.2.6.2 PFair Staggered Model . . . 54
3.2.6.3 PFair for intra-sporadic model . . . 55
3.2.6.4 Supertasking . . . 56
3.3 Deadline Partitioned Fair scheduling . . . 57
3.3.1 Principle . . . 57
3.3.2 Division of time in intervals . . . 57
3.3.3 Scheduling within an interval . . . 58

3.3.3.1 Nodal execution time computation . . . 61
3.3.3.2 Taskset dispatching . . . 63
3.3.4 Classification of DP-Fair algorithms . . . 64
3.3.5 Non-work conserving algorithms . . . 65
3.3.5.1 Real number nodal execution time computation & dynamic dispatching . . . 65
3.3.5.2 Real number nodal execution time computation & static dispatching . . . 67
3.3.5.3 Integer number nodal execution time computation & static dispatching . . . 68
3.3.5.4 Integer number nodal execution time computation and dynamic dispatching technique . . . 70
3.3.6 Work conserving algorithms . . . 71
3.3.6.1 NVNLF based on Extended TN-plane [29] . . . 71
3.3.6.2 NNLF based on TR-plane . . . 75
3.4 RUN (Reduction to a UNiprocessor) . . . 78
3.4.1 Offline scheduling . . . 78
3.4.2 Online scheduling . . . 81
3.4.3 Performances . . . 82
3.5 Conclusion . . . 82

4 Overhead control techniques **85**
4.1 Events in task execution . . . 87
4.2 Preemption . . . 87
4.3 Migration . . . 88
4.4 Overhead estimation . . . 89
4.5 Overhead control heuristics . . . 91
4.5.1 Migration control heuristic (MCH) . . . 92
4.5.2 Preemption control heuristic (PCH) . . . 92
4.5.3 Hybrid = (Migration + Preemption) control . . . 94

5 Experimental setup **95**
5.1 Taskset generator . . . 96
5.1.1 Properties of task period set . . . 99
5.1.1.1 Set 1 . . . 100

5.1.1.2 Set 2 . . . 100
5.1.1.3 Set 3 . . . 101
5.1.1.4 Set 4 . . . 101
5.1.2 Distribution of task utilization factors . . . 104
5.2 Simulator . . . 106
5.2.1 Short overview of STORM . . . 106
5.2.2 Implemented algorithms . . . 107
5.3 Experimentation process . . . 108

6 Experimental results **113**
6.1 Non-work conserving case . . . 114
6.1.1 Migration control (NWC) . . . 114
6.1.1.1 Different algorithms at U=M, U=0.75M, U=0.5M . . . 114
6.1.1.2 Hybrid-NWC algorithm at different sets of taskperiod 116
6.1.1.3 Hybrid-NWC algorithm at variable task to processor ratio . . . 116
6.1.1.4 Hybrid algorithm for U/N . . . 116
6.1.2 Preemption control (NWC) . . . 122
6.1.2.1 Different algorithms at U=M, U=0.75M, U=0.5M . . . 122
6.1.2.2 Hybrid-NWC algorithm at different sets of taskperiod 122
6.1.2.3 Hybrid-NWC at variable task to processor ratio . . . 123
6.1.2.4 Hybrid-NWC algorithm varying U/N . . . 123
6.1.3 Standard deviation . . . 129
6.2 Work conserving case . . . 129
6.2.1 Migration control (WC) . . . 130
6.2.1.1 Different algorithms at U=0.75M, U=0.5M . . . 130
6.2.1.2 Hybrid-WC algorithm at different sets of taskperiod . 130
6.2.1.3 Hybrid-WC algorithm at variable task to processor ratio 130
6.2.1.4 Hybrid algorithm for U/N . . . 134
6.2.2 Preemption control (WC) . . . 134
6.2.2.1 Different algorithms at U=0.75M, U=0.5M . . . 134
6.2.2.2 Hybrid-WC algorithm at different sets of taskperiod . 135
6.2.2.3 Hybrid-WC at variable task to processor ratio . . . 135
6.2.2.4 Hybrid-WC algorithm for U/N . . . 135
6.2.3 Standard deviation . . . 139
6.3 Conclusion . . . 140

7 General conclusion **141**

8 Introduction générale **147**

9 Ordonnancement STR multiprocesseur **151**
9.1 Introduction . . . 152
9.2 Le logiciel d'application . . . 152
9.3 L'unité de traitement . . . 153
9.4 La politique d'ordonnancement . . . 155
9.4.1 Ordonnancement monoprocesseur . . . 156
9.4.2 Ordonnancement multiprocesseur . . . 156
9.4.2.1 Ordonnancement partitionné . . . 158
9.4.2.2 Ordonnancement global . . . 161
9.4.3 Ordonnancement hybride . . . 162
9.5 Conclusion . . . 162

10 Etat de l'art des algorithmes globaux optimaux **163**
10.1 Introduction . . . 164
10.2 PFair : Proportionate Fair scheduling . . . 165
10.2.1 Principe . . . 165
10.2.2 Ordonnancement PFair . . . 167
10.2.3 Algorithmes PFair . . . 169
10.2.4 Implémentation d'un algorithme PFair . . . 169
10.2.5 Limitations de PFair . . . 169
10.2.6 Extensions de PFair . . . 172
10.2.6.1 Le modèle Early Release Fair . . . 172
10.2.6.2 Le modèle PFair Staggered . . . 172
10.2.6.3 Autres extensions . . . 173
10.3 Ordonnancement Deadline Partitioned Fair . . . 174
10.3.1 Principe . . . 174
10.3.2 Division du temps en intervalles . . . 174
10.3.3 Ordonnancement dans un intervalle . . . 175
10.3.3.1 Calcul du temps d'exécution alloué sur un intervalle . 177
10.3.3.2 Allocation des tâches aux processeurs dans un intervalle 179
10.3.4 Classification des algorithmes DP-Fair . . . 181
10.3.5 Algorithmes avec ordonnancement non-conservatif . . . 182
10.3.6 Algorithmes conservatifs . . . 182
10.3.6.1 NVNLF basé sur les TN-plane étendus . . . 183
10.3.6.2 NNLF basé sur les TR-plane . . . 184
10.4 RUN (Reduction to a UNiprocessor) . . . 185

11 Techniques de maîtrise des surcoûts **187**
11.1 Evénements durant l'exécution d'une tâche . . . 188
11.2 Preemption . . . 188
11.3 Migration . . . 189

11.4 Estimation des surcoûts . . . 190
11.5 Heuristiques de maîtrise des surcoûts . . . 192
11.5.1 Heuristique de maîtrise des migrations (*MCH*) . . . 193
11.5.2 Heuristique de maîtrise des préemptions (*PCH*) . . . 193
11.5.3 Heuristique hybride : maîtrise des migrations et des préemptions (*Hybrid*) . . . 194

12 Mise en place de l'expérimentation **195**
12.1 Introduction . . . 196
12.2 Générateur de données . . . 196
12.2.1 Propriétés des jeux de périodes . . . 198
12.2.2 Distribution des facteurs d'utilisation des tâches . . . 198
12.3 Simulateur . . . 200
12.3.1 Présentation rapide de STORM . . . 200
12.3.2 Algorithmes implémentés durant la thèse . . . 201
12.4 Processus d'expérimentation . . . 202

13 Résultats expérimentaux **205**
13.1 Résultats avec la version non-conservative des algorithmes (NWC) . . . 205
13.1.1 Maîtrise des migrations (NWC) . . . 206
13.1.2 Maîtrise des préemptions (NWC) . . . 206
13.2 Résultats avec la version conservative des algorithmes (WC) . . . 206
13.2.1 Maîtrise des migrations (WC) . . . 206
13.2.2 Maîtrise des préemptions (WC) . . . 207
13.3 Conclusion . . . 207

14 Conclusion générale **209**

Bibliography **211**

List of Figures

2.1 Task parameters . . . 24
2.2 A SMP system . . . 25
2.3 Scheduling anomaly . . . 30
2.4 Example of Dhall effect . . . 32
2.5 Partitioned scheduling . . . 33
2.6 Taskset is schedulable with migration . . . 35
2.7 Global scheduling . . . 36

3.1 Fluid versus real execution . . . 43
3.2 (a) Fluid schedule (b) Time division with $T_1.u$ = 0.4 and $T_2.u$ = 0.6 . . . 44
3.3 Execution of task in PFair . . . 45
3.4 Windows in PFair for a task with $T_i.u = 0.6$ . . . 47
3.5 An example of a PFair execution sequence . . . 50
3.6 Bus load across scheduling points in 8 processors system in PFair [37] . . . 51
3.7 Execution of task in Early Release Fair . . . 54
3.8 PFair Vs Staggered model [37] . . . 55
3.9 Bus load across scheduling points in 8 processors system in Staggered model [37] . . . 56
3.10 Defining boundary in DP-Fair . . . 58
3.11 A TN-plane . . . 59
3.12 Consecutive TN-planes. . . . 60
3.13 Scheduling in a TN plane . . . 61
3.14 A schedule in a TN-Plane . . . 64
3.15 DP-Fair classification graph . . . 65
3.16 DP-Wrap algorithm . . . 68
3.17 DP-Wrap (a) With out mirroring (b) With mirroring . . . 69
3.18 Result of McNaughton's algorithm for the taskset . . . 71
3.19 ETN-plane for a single task . . . 74
3.20 Multiple TR-planes of a task . . . 76
3.21 A TR plane . . . 77
3.22 (a) Dual schedule (b) Primal schedule . . . 79

3.23 REDUCE operation . . . 80
3.24 Server tree . . . 81

4.1 Events along a task execution . . . 87

5.1 Experimental setup . . . 96
5.2 Task generator . . . 97
5.3 Variation of intervals ϕ_1 . . . 102
5.4 Variation of intervals ϕ_2 . . . 102
5.5 Variation of intervals ϕ_3 . . . 103
5.6 Variation of intervals ϕ_4 . . . 103
5.7 Distribution of utilization factor for U/N = 0.5 . . . 105
5.8 Distribution of load for U/N = 0.25 . . . 105
5.9 Distribution of load for U/N = 0.75 . . . 105
5.10 Architecture of STORM . . . 106
5.11 An input XML file for STORM . . . 107
5.12 Observer program . . . 108
5.13 Data generation tree . . . 110

6.1 Migration control at U=M . . . 117
6.2 Migration control at U=0.75M . . . 117
6.3 Migration control at U=0.5M . . . 118
6.4 Migration control of hybrid-NWC of different period sets at U=M . . . 118
6.5 Migration control of hybrid-NWC of different period sets at U=0.75M . . . 119
6.6 Migration control of hybrid-NWC of different period sets at U=0.5M . . . 119
6.7 Migration control of hybrid-NWC varying N/M at U=M . . . 120
6.8 Migration control of hybrid-NWC varying N/M at U = 0.75M . . . 120
6.9 Migration control of hybrid-NWC varying N/M at U = 0.5M . . . 121
6.10 Migration control of hybrid-NWC at variable U/N . . . 121
6.11 Preemption control at U=M . . . 124
6.12 Preemption control at U=0.75M . . . 124
6.13 Preemption control at U=0.5M . . . 125
6.14 Preemption control of hybrid-NWC of different period sets at U=M . . . 125
6.15 Preemption control of hybrid-NWC at different period sets at U=0.75M . . . 126
6.16 Preemption control of hybrid-NWC at different period sets at U=0.5M . . . 126
6.17 Preemption control of hybrid-NWC at variable N/M at U = M . . . 127
6.18 Preemption control of hybrid-NWC at at variable N/M at U = 0.75M . . . 127
6.19 Preemption control at variable task to processor ratio U = 0.5M . . . 128
6.20 Preemption control of hybrid-NWC at variable U/N . . . 128
6.21 Migration control at U=0.75M . . . 131
6.22 Migration control at U=0.5M . . . 131

6.23 Migration control of hybrid-WC of different period sets at U=0.75M . . 132
6.24 Migration control of hybrid-WC of different period sets at U=0.5M . . . 132
6.25 Migration control of hybrid-WC at variable N/M at U=0.75M 133
6.26 Migration control of hybrid-WC at variable N/M at U=0.5M 133
6.27 Migration control of hybrid-WC at variable U/N 134
6.28 Preemption control at U=0.75M 136
6.29 Preemption control at U=0.5M 136
6.30 Preemption control of hybrid-WC of different period sets at U = 0.75M 137
6.31 Preemption control of hybrid-WC of different period sets at U = 0.5M . 137
6.32 Preemption control at variable task to processor ratio at U = 0.75M . . . 138
6.33 Preemption control at variable task to processor ratio at U = 0.5M . . . 138
6.34 Preemption control of hybrid-WC for U/N 139

9.1 Paramètres d'une tâche 153
9.2 Une architecture SMP 155
9.3 La configuration est ordonnançable avec la migration 160

10.1 Exécution fluide et exécution reéle 164
10.2 (a) Ordonnancement fluide (b) Allocation des slots temporels avec $T_1.u$ = 0.4 and $T_2.u$ = 0.6 166
10.3 Exécution d'une tâche PFair 167
10.4 Les fenêtres de l'ordonnancement PFair pour une tâche avec $T_i.u = 0.6$ 168
10.5 Un exemple d'exécution PFair d'une configuration 170
10.6 Charge d'un bus partagé par 8 processeurs aux points d'ordonnancement [37] 171
10.7 Exécution d'une tâche avec le principe *ERFair* 173
10.8 Modèle PFair et modèle PFair Staggered [37] 174
10.9 Charge d'un bus partagé par 8 processeurs aux points d'ordonnancement avec le modèle PFair Staggered [37] 175
10.10Définition des intervalles d'ordonnancement dans *DP-Fair* 176
10.11Un exemple de TN-plane 177
10.12TN-planes consécutifs. 178
10.13Ordonnancement à l'intérieur d'un TN plane} 179
10.14Exemple d'exécution de jobs dans un TN-Plane 180
10.15Classification des algorithme DP-Fair 182
10.16Les multiples TR-planes pour une tâche 185

11.1 Events along a task execution 189

12.1 Mise en place de l'expérimentation 196
12.2 Générateur de configuration de tâches 197
12.3 Distribution des facteurs d'utilization factor pour U/N = 0.5 199

12.4 Distribution des facteurs d'utilization factor pour U/N = 0.25 199
12.5 Distribution des facteurs d'utilization factor pour U/N = 0.75 199
12.6 Architecture de STORM 200
12.7 Exemple de fichier d'entrée pour STORM 201
12.8 Arbre de génération des données 203

NOTATIONS

Constants

M	Number of processors
N	Number of tasks

Sets

τ	Set of peridic tasks ={$T_i,\ i = 1, ..., N$ }
Π	Set of processors = {$P_i,\ i = 1, ..., M$ }
ϕ_j	j^{th} set of task period
Γ	Set of servers

Task parameters

T_i	A task with with index i
$T_i.o$	Offset job of task T_i
$T_i.r$	Release time of a job of task T_i
$T_i.p$	Period of task T_i
$T_i.d$	Deadline of a job of task T_i
$T_i.e$	Worst case execution time of task T_i
$T_i.u$	Utilization factor of task T_i i.e. $T_i.u = \frac{T_i.e}{T_i.p}$
$T_i.R$	Total remaining execution time of a job of taskT_i at given time
T_i^*	Dual of task T_i
U	Total utilization factor i.e. $\sum_{i=1}^{N}(T_i.u)$

Sub-task parameters

$T_{i.k}$	k^{th} subtask of task T_i
$r(T_{i.k})$	Pseudo-release of sub-task k of task T_i
$d(T_{i.k})$	Pseudo-deadline of sub-task k of task T_i
$w(T_{i.k})$	Windows of sub-task k of task T_i
$b(T_{i.k})$	Successor bit of sub-task k of task T_i
$G(T_{i.k})$	Group deadline of sub-task k of task T_i

Node parameters

b_k	Start of a node
$T_i.l$	Nodal execution time of a task T_i
$T_i.m^k$	Mandatory units allocated to a task T_i at b_k
$T_i.a$	Additional time units allocated to a task T_i at b_k
L'	Unused time units in the node

Time parameters

t	Time
t_{curr}	Current time
t_B	Time of event *B*
t_C	Time of event *C*
t_F	Time of event *F*
l	A slot in *PFair* [t, t + 1)
S	Schedule over a slot

1

General introduction

Over the last two decades, there is an immense growth in the user population of real-time embedded systems and interestingly, there is no apparent change in this trend in the near future. The life in the modern era seems incomplete without the applications of real-time systems. Telecommunications, control of production systems, military equipments, medical imaging, avionics and transport systems are few of the large scale applications. On the other hand, smart phones, tablet computers, digital cameras and multimedia systems have become an integral part of personal lives. The user needs and expectations have not only created an environment of competition and business opportunity for the manufacturers but have also suggested an area of vast exploration for the researchers.

Due to the customers demand for highly sophisticated applications with multiple functionalities, there is constant requirement for an increased processing speed. To fulfill this demand, the focus of the research in the hardware industry was to increase the speed of a single processor. However, soon it was found that the speed of a single processor may not be increased beyond certain limits, mainly because of high power consumption and too much heat dissipation. It led the researchers to find some alternatives. Multiprocessing came as one of the alternatives which describes having two or more processing units (Multiple Processors) each sharing main memory and peripherals. The purpose of multiprocessing was to enhance the effective speed to carry out the workload in more comprehensive and efficient way. In a multiprocessor system, all the processors may be treated equally or some may be reserved for special purposes. It must be noted that it is different from multitasking or multiprogramming which refers to the execution of multiple concurrent software processes in a system as opposed to a

single process at any one instant.

The rapid growth of multiprocessor systems was not possible without a strong support of hardware developers. These days multiprocessor architectures are commercially available for embedded systems because the major manufacturers of processors like AMD, Analog Devices, Intel, Freescale and IBM are producing low cost multiprocessor architectures. The advent of multicore architecture, where multiple processor cores are placed on a single chip, has proven a giant step in the growth of multiprocessors. A multicore processor implements the multiprocessing in a single physical package. In 2007, Intel announced a *Teraflops Research Chip* with 80 cores.

The hardware architecture is managed by an operating system which is known as *Real-Time Operating Systems* ($RTOS$) in the case of real-time applications. The design of a real-time system is generally based on a set of concurrent periodic tasks or processes with some timing requirements. Each task needs a processor share to get completed within its time limits. The scheduler in the $RTOS$ is responsible for managing the available processing resources. A scheduler defines a set of rules to decide the allocation of scarce resources to the tasks to achieve one or more criteria like deadline achievement, energy saving, etc. The set of rules is known as a *scheduling policy*. The objective of scheduling algorithm in a real-time system is to execute the tasks such that application meets its deadlines.

The history of research on scheduling backs to 1950s when some industries faced problem of managing the resources. Before multiprocessor scheduling, the uniprocessor scheduling was a mature field which has been studied for a long time. Few known uniprocessor algorithms like *Earliest Deadline First* (*EDF*) and *Rate Monotonic* (*RM*) are found to be optimal in their class and have been used successfully in the industry. However, these optimal uniprocessor scheduling algorithms do not sustain their optimality when used for scheduling in a multiprocessor system. Liu [27] gave a findings in 1969: " *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*". The two main types of real-time multiprocessor scheduling techniques are: partitioned scheduling and global scheduling. There is a third type, which is termed as hybrid scheduling, is a compromise between partitioned and global scheduling.

The partitioned algorithm is the relatively mature branch of multiprocessor scheduling and is mostly used in the industry. It does not allow the task migration and reduces the multiprocessor scheduling problem to a uniprocessor scheduling problem. Consequently, it uses the whole experience of uniprocessor scheduling. However partitioned

algorithms do not guarantee the complete utilization of resources because the allocation of the tasks to the processor is analogous to bin-packing problem which is *NP-hard* in the strong sense.

Global scheduling, which permits the migration of tasks among the processors, was considered inferior to non-global techniques due to a so called "Dhall effect". Therefore, the main focus of the research remained on the non-global scheduling techniques. However, later it was found on the bases of work of Philips et al. [54] in 1997 and Funk et al. [32] in 2001 that Dhall effect has a relation with high utilization factor and not with the global scheduling. After this finding the global scheduling algorithms drew a lot of attention of the researchers in the domain. An important event in the history of multiprocessor scheduling was advent of *Proportionate Fair* (*PFair*) by Baruah et al. in 1996 [56] which is the first optimal multiprocessor scheduling algorithm for periodic taskset with implicit deadlines. Afterwards, another class of optimal global scheduling algorithms was proposed which is called *Deadline Partitioning Fair (DP-Fair).* A large number of optimal global algorithms belong to this class. Both *PFair* and *DP-Fair* are based on the principle of fairness.

Although global scheduling algorithms are optimal but this optimality is achieved at the cost of frequent scheduling points, migrations and preemptions. Theoretically the costs of migrations and preemptions are considered to be zero but practically this is not the case and the overhead may jeopardize the schedulability of the configuration. On some modern multicore architectures, though the cost of migration is much lower than in past but it is still a non-zero value. Therefore, critics object that this optimality is not possible practically and these algorithms are inferior to the partitioned scheduling algorithms.

Contribution

The problem of the thesis is:

"*Overhead control in optimal global scheduling algorithms in real-time multiprocessor system*s".

It addresses to the problem of overhead confronted in the optimal global scheduling in hard real-time multiprocessor systems. This research deals with the scheduling of periodic taskset on a symmetric multiprocessor system. In this regard, both non-work conserving and work conserving cases are taken into account.

The basic objective of this work is to understand the working of *DP-Fair* schedul-

ing algorithms and to devise a few techniques to control the overhead without affecting the optimality of the scheduling algorithms. The reason for choosing *DP-Fair* model is that it is better than former optimal scheduling class *PFair* in terms of overhead. In addition, there is a space in this scheduling technique where the overhead can be easily controlled.
In this work, a classification of *DP-Fair* algorithms on the bases of independent steps involved for the completion of the scheduling process is proposed. This classification provides a degree of freedom to check different combinations of solution for these steps which is quite interesting. For the reference, a *DP-Fair* which can be implemented and which has a space of overhead control has been used.

Most of the *DP-Fair* algorithms do not describe any policy about the processor allocation, i.e. a task can be executed on any processor without keeping the history of the last processor on which it was executed. Similarly, the execution task list is built by arbitrarily choosing the tasks from the ready task list without taking into account the sequence of last running tasks. In this work, it has been tried to use these two points to control the number of migrations and preemptions without compromising the optimality of the original *DP-Fair*. Two heuristics have been used in this regard to control overhead. The first one controls the number of preemptions and the second one controls the number of migrations.

In this work, a statistical approach is used to evaluate the performance of heuristics. An extensive simulation based experiments have been performed to check out the significance of using overhead control heuristics along with classical *DP-Fair* algorithm. These experiments were performed using a well specified data generation process of our own and overhead control is checked by varying the different parameters involved.

The results obtained are very encouraging and show a significant reduction in the overhead due to both migrations and preemptions. In addition, the pattern of the results is generally consistent over varying number of processors and different sets of task periods.

Thesis Organization

The thesis is composed of six chapters other than the current one which is a general introduction.

In the chapter 2, the basic theory of real-time systems, the multiprocessing and the multiprocessor scheduling are described. The two main types of multiprocessor scheduling are presented along with the benefits and shortcomings of each. The system model is also mentioned which describes the software and hardware models which have

been used.

The chapter 3 can be regarded as bibliographic chapter of the thesis. In this chapter, the optimal global scheduling techniques are described in detail. The two main classes of optimal scheduling techniques in the literature based on the concept of fairness namely *PFair* and *DP-Fair* are discussed in detail. A classification of *DP-Fair* algorithms is proposed which is based on the scheduling steps involved in it. The recently proposed optimal technique RUN, which is based on a light form of fairness, is also described briefly.

In the chapter 4, the concept of overhead is uncovered. A migration and a preemption is defined in terms of events involved and the overhead incurred. Our overhead control heuristics are also presented along with their algorithms and complexities.

In the chapter 5, the whole experimental setup is described. It includes the data generator, the simulator STORM and the data analyzing tool based on MALAB. The whole data generation process and different aspects of the generated data are presented. It also mentions the extent of experimentation process to show the reliability of results.

In the chapter 6, the results of the simulation process are presented along with brief reasoning. The results are presented for non-work conserving case followed by work conserving one. In each case, the results are further divided into migration control and preemption control. The results are presented varying different dataset parameters so as to highlight important aspects of overhead control heuristics.

In the chapter 7, a general conclusion of this thesis is given.

Remark: *A very simplified version of the thesis in french language is also presented in the end of this report. This is to comply with the requirement of «Ecole doctorale STIM» of the University of Nantes.*

2

Real-time multiprocessor system scheduling

Overview

In this chapter, the basic concepts of real-time systems, multiprocessing and multiprocessor scheduling are presented. The two main types of multiprocessor scheduling are explained. The system model along with basic definitions of the related terms are also discussed.

2.1 Introduction

A real-time system differs from the ordinary computing systems in the sense that it also exhibits some timing constraints in addition to their correct logic. A basic run time activity or a *task* in a real-time system has either some specific start time or some end time or even both in some cases. The end time, which is technically termed as *deadline,* is the most significant timing constraint and is mostly studied in real-time systems. A task in real-time system loses its significance if it is completed after its deadline.

The real-time systems are categorized into *hard real-time systems* and *soft real-time systems* on the bases of their tolerance to miss the timing constraints. In *hard real-time systems,* the missing of a timing constraint is unacceptable. It can result in the total failure of the system and have fatal consequences. Critical applications like the anti-missile defense system or the anti-lock braking system in the automobiles are examples of hard real-time systems. A little delay can be fatal in both of these applications. In *soft real-time systems,* timing faults can be accepted upto some extent. However the usefulness of a result degrades after the timing constraints are missed. In this category, only the precedence and sequence for the task operations are defined, interrupt latencies and context switching latencies are small but there can be few deviations between expected latencies of the tasks and observed time constraints and a few deadline misses are accepted. Live video streaming, where the pictures or video quality can be compromised to some extent is an example of soft real-time system.

Generally, a real-time system can be considered as built upon following three components:

1. Workload: A workload comprises the programs which implement the funtionalities of the real-time application. These programs are termed as tasks and are processed over the processing unit. A real-time system is generally based on a set of concurrent recurring tasks with some timing requirements and characterized by some timing parameters.

2. Processing unit: Some hardware resources are required to execute the workload. A basic processing units includes a processor, cache memory, external memory and connection buses. A hardware platform is a uniprocessor if there is only one processor in the system. If the number of processors are more than one, it is known as multiprocessor system.

3. Scheduling policy: It is responsible for the execution of the workload over the processing unit such that each task meets the timing constraint. A scheduler implements a scheduling policy. It defines that at any given time that which task will execute and on which processor if there are more than one processor in the system.

In the next sections, we will discuss each of these components in detail.

2.2 Workload

The workload is usually considered through a task model which is an abstraction of its programs. A task may be periodic, aperiodic or sporadic. A periodic task is repeated after a fix interval of time. An aperiodic task is activated after an irregular interval of time. A sporadic task is just like an aperiodic task but with a minimum fix interval between any two activations.

The classical model [44] considers that τ is a static set of N periodically arriving real-time tasks $\tau = \{T_i,\ i = 1, 2...N\}$. The task T_i is characterized by a release time $T_i.o$, time period $T_i.p$, worst case execution time $T_i.e$ and a prescribed deadline $T_i.d$. Each task in such a system is invoked or released every $T_i.p$. One such complete invocation is called a *job* of the task. For *arbitrary deadline* systems, there are no constraints between the deadline of a task and its time period, in particular the deadline may be greater than the task period. For *implicit deadline* systems, it is required that for all tasks the deadlines are equal to the periods i.e. $T_i.d = T_i.p$. For *constrained deadline* systems, the deadlines are always less than or equal to the task period i.e. $T_i.d \leq T_i.p$. The *offset* of a task T_i, denoted by $T_i.o$, is the release time of its first job. A taskset in which the offset of all the tasks are equal is called *synchronous*. The taskset is called *asynchronous* if the tasks have different offset values.

A periodic task T_i, with release time $T_i.o$ is shown in figure 2.1. It shows the first two jobs of the task T_i. The utilization factor $T_i.u$ of a task T_i is defined as $T_i.e/T_i.p$. It represents the fraction of time for which T_i needs a processing unit. The total utilization factor U of the system is the sum of the utilization factors of all the tasks in the taskset.

$$U = \sum_{i=1}^{N} (\frac{T_i.e}{T_i.p}) \tag{2.1}$$

System utilization is another parameter which represents the fraction of time for which the processors remain busy. It is defined as

$$U_S = \frac{U}{M} \tag{2.2}$$

If the value of system utilization is 1, it means that all the processors will remain busy for all the times.

The model that will be considered consists of synchronous periodic tasksets with implicit deadlines. We assume that all the tasks are independent, i.e. they do not share any common resource and do not have any precedence for each other.

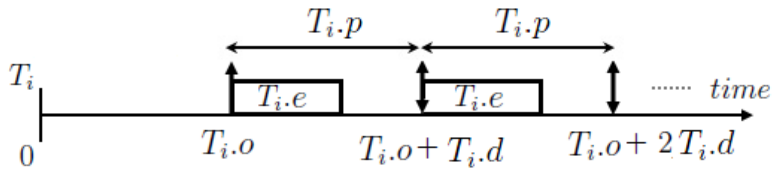


Figure 2.1: Task parameters

2.3 Processing unit

A processor is the basic hardware unit of the computing system. It can be regarded as the brain of the system. The physical structure of a simple processor consists of a single core on a die. The processor core is an independent execution unit. When there are more than one core on a single die, it is known as multicore processor. A dual-core processor has two cores and a quad-core processor contains four cores. In 2007, Intel announced a processor named Teraflops Research Chip with 80 cores.

When there are more than one processor in the system it is known as multiprocessor system. If the processors are placed on the same chip of *Integrated Circuits* (ICs), it is termed as *System on Chip* (SoC). If the processors belong to the different chips of ICs, it is termed as *Multiple Chip Module* (MCM). The system is called *multicore multiprocessor* if each processor in the multiprocessor system has more than one core.

When we use the term *multiprocessor system* in our work, it means a system with multiple independent processors which share the memory and can execute the tasks in parallel. It has following advantages :

1. With an increased number of processors, more work is done in less time and hence we get better performance (higher throughput and shorter response time). However, when a task is executed on more than one processors, an overhead is established. An additional loss also comes due to the contention of the shared resources and consequently we do not get the expected gain;
2. Multiprocessor systems are more reliable. In a well designed system , the failure of a single processor will not halt the whole system. The working processor(s) automatically take(s) over the system workload until repairs are made;
3. Multiprocessor systems are economically better than mono-processor systems because they share peripherals, power supplies and memories.

The multiprocessor systems can be classified into loosely coupled and tightly coupled systems on the bases of their physical structure.

In a *loosely coupled multiprocessor* system, processors are connected through high speed links such as ethernet. Each processor has its own memory and I/O channels. There are functionally specialized channels each doing different things.

Manufacturers	Products	Number of cores
Freescale Semiconductor	MPC8640D	2
Texas Instruments	ARM Cortex-A9 MPCore	4
Intel	Intel® Xeon® Processor E3	4

Table 2.1: Examples of multiprocessor architectures

In a *tightly coupled multiprocessor* system, the processors are connected at the bus level. The processors may share some central memory. This is also known as chip level multiprocessing. To get effectively the benefits of parallelism, shared memory must not become a performance bottleneck.

Recently "many core architecture" became available for high power computing system. This work does not address the many core architecture where there is a grid of cores on a single chip with fast communication links among them.

Table 2.1 gives the names of some vendors of multiprocessor components with the name of their products available in the market.

A classification is usually made in scheduling research on the bases of similarities and differences among the processors.

Homogeneous: In this model, all the processors are identical and share the same main memory. Normally this model is used to run the same type of tasks. Any task may be executed on any processor. It is also known as *Symmetric MultiProcessor* (SMP). A simple SMP is shown in the figure 2.2.

Uniform: In this model, there are processors with different speeds and the rate of task execution of task depends on the speed. If the speed of processor 1 is double than that of processor 2, the processor 1 takes half time to run the same task as that of processor 2.

Heterogeneous: The processors are not all identical. They are classified on the basis

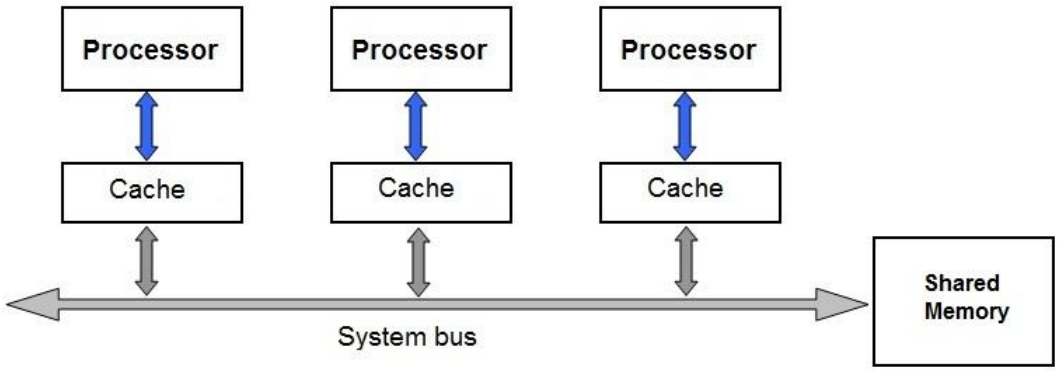


Figure 2.2: A SMP system

of work, e.g. GPU (Graphics Processing Unit) and DSP (Digital Signal Processor). All the tasks may not be executed on all the processors. This is also known as Asymmetric MultiProcessor (AMP).

Most of the real-time scheduling research is focused on homogeneous or symmetric multiprocessor platforms because the scheduling problem is much simpler in this case. We also work on the very same model where M is the number of identical processors.

2.4 Scheduling policy

The objective of a scheduling policy is to find a feasible schedule of the taskset on the processing unit. A taskset is *feasible* if a schedule exists that meets the deadlines. A taskset is *schedulable* with respect to a scheduling algorithm if the algorithm generates a feasible schedule. An *optimal algorithm* for a task model is the one which can generate a feasible schedule for all the tasksets of that model that are schedulable by any other algorithm [17]. Different objectives can be set for a scheduling policy. The objective may be meeting the deadlines, providing higher utilization of processing unit, minor overheads or short response time etc. In scheduling of hard real-time systems, meeting the deadlines is the most important objective.

Scheduling policy defines the relative priority of each task in the taskset which decides the dispatching order of the tasks to the processing unit. The task *priority* is the urgency level allocated to the task and a scheduler always chooses a task of higher priority over one with a lower priority. A task is said to be preempted when a task is stopped or interrupted, without requiring its cooperation, with an intention of resuming it later on. A scheduling algorithm may or may not allow task preemptions. In a pure *non-preemptive scheduling*, a scheduler never interrupts a running task and waits the task to end. In *preemptive scheduling,* a task is forced to release the processor on the arrival of a task with higher priority or due to non-work conserving behavior of the scheduling algorithm. [1]. Preemption causes a context switching. A *context switching* is the switching of the processor from one task to another. A context is the contents of a processor's registers and program counter at any given time. When a task is preempted, its contexts are saved and are retrieved when the task execution is resumed.

Both preemption and context switch cause some overheads in the scheduling process. Switching from one task to another requires a certain amount of time for doing the administration, saving and loading registers and memory maps, updating various tables and list.

To simplify the scheduling the cost of preemption, context switching and run-time operations of the scheduler is assumed to be zero.

1. A scheduling algorithm is said to be work conserving if and only if it never leaves any processor idle when there exists at least one active task awaiting for the execution in the system. The system is *non-work conserving* otherwise.

In hard real-time systems, guarantees are required to verify that all the tasks will meet their deadlines. Therefore some conditions for testing the schedulability are proposed. Utilization bound is a useful performance metric in this regard. The maximum *utilization bound* U_A of an algorithm A is the maximum utilization of any taskset that is schedulable according to that algorithm. It can be regarded as sufficient (but not necessary) schedulability test.

A scheduler is said to be *offline* if all the scheduling decisions are made before running the application. The actual run time scheduling is done using a precomputed table where execution decisions are recorded. In *dynamic* scheduling the task priorities and assignment decisions are taken at run time.

Scheduling is classified into uniprocessor and multiprocessor scheduling depending on the number of processors in the system.

2.4.1 Uniprocessor scheduling

In a uniprocessor or monoprocessor system, a single processing unit is shared by all the tasks. Therefore, a scheduler is required which decides the timing allocation of the processor to the taskset. Least Laxity First (LLF) [50], Earliest Deadline First (EDF) [44], Rate monotonic (RM) [44], Round Robin (RR) [14], First-Come-First-Serve $(FCFS)$ and Shortest Process First (SPF) are few examples of uniprocessor scheduling algorithms. LLF [50] and EDF [24, 44] are optimal uniprocessor scheduling algorithms for periodic taskset with implicit deadlines.

Task priorities are either fixed task, fixed job or dynamic ones.

Fixed Task priority: The priority of a task is defined at the start of execution and it will always remain constant. RM [44] is an example of fixed or static task priority assignment. The shorter the task period duration is, the higher is the task priority. A set of N periodic tasks with implicit deadlines is schedulable on a uniprocessor by the RM algorithm if the processor utilization satisfies the following condition [44]:

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq N(2^{1/N} - 1) \tag{2.3}$$

Fixed Job priority: Each job of the task is assigned a priority that remains the same until it is finished. EDF is an example of fixed job priority assignment. In EDF, at any time, the task with the earliest next deadline gets the highest priority (with ties broken arbitrarily). A periodic taskset with implicit deadlines is schedulable if and only if following condition is satisfied [44] :

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq 1 \tag{2.4}$$

Dynamic priority: The priority assigned to a job varies with the time even within a single period. LLF [11] scheduling is an example of dynamic priority assignment. The laxity of a job is defined as the maximum amount of time that the job can wait while still meeting its deadline. In LLF the task with the smallest laxity is given the highest priority. A periodic taskset with implicit deadlines is schedulable if and only if the following condition is satisfied :

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq 1 \tag{2.5}$$

2.4.2 Multiprocessor scheduling

Scheduling on multiprocessors is defined as: "How can a set of tasks τ, can be executed on a set of M processors subject to time constraints of the tasks". It can be divided into two sub-problems:

- The allocation problem: On which processor a task will execute ?
- The priority allocation problem: What is the relative order of task execution on a processor ?

A multiprocessor scheduler is different from a uniprocessor scheduler in the sense that it has to specify the processor (spatial dimension) in addition to the timing allocation (temporal dimension) of the tasks. Unfortunately, it is not a simple extension of the uniprocessor scheduling problem. The optimal uniprocessor scheduling techniques like EDF [44, 24] and LLF [50] do not show the same efficiency when used for multiprocessor scheduling and result in low processor utilization [27]. Liu [46] noted in 1969 that multiprocessor real-time scheduling is much more difficult than uniprocessor scheduling. "*Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors*."

Scheduling anomaly is another important issue which commonly occurs in multiprocessor scheduling opposite to that of uniprocessor scheduling. A *scheduling anomaly* occurs when any change in the scheduling configuration becomes counter productive and the total processor utilization increases or system may be unschedulable. For example, increasing a task period while keeping the other parameters constant normally decreases the processor utilization but in some cases it may result in unschedulability and cause some of the already guaranteed tasks to miss their deadlines. Similarly deadlines originally met can be missed on decreasing execution times of tasks.

Task	$T_i.p = T_i.d$	$T_i.e$
T_1	5	1
T_2	6	3
T_3	8	5
T_4	6	4

Table 2.2: Periodic synchronous taskset with implicit deadlines

According to the Graham's theorem *[35], " If a task set with fixed priorities, execution times, and precedence constraints is scheduled according to priorities on a fixed number of processors, increasing the number of processors, reducing the computation times, or weakening the precedence constraints can increase the schedule length".*

To understand such an anomaly consider the one time schedule of the taskset given in table 2.2 over a multiprocessor system comprising two processors using theEDF algorithm. The schedule is shown in figure 2.3 (a) where all the tasks meet their deadlines. If we increase the period of T_1 from 5 to 7, the total utilization factor decreases. The resulting schedule is shown in the figure 2.3 (b), where T_3 misses the deadline.

Following general rules are observed in multiprocessor scheduling:

- A processor can not execute more than one task at any given time;
- A single task cannot be executed on more than one processor at any given time.

Sometimes a task is preempted on a processor and, after a pause, resumes on a different processor. This is known as *migration*. Multiprocessor scheduling is divided into following three categories on the base of freedom in task migration.

1. The migration is not allowed. The task is allocated to a processor and it cannot migrate to any other processor.
2. A limited migration of the task is allowed. A task can migrate after completing one whole job on one processor.
3. The migration is allowed at the job level. A task may migrate to a different processor even during execution of a job.

The scheduling algorithms which do not allow the migration are called *partitioned algorithms*. On the other hand, the algorithms which allow the task migration are known as *global scheduling*.

The very first work in multiprocessor scheduling theory came as an extension of uniprocessor scheduling. The tasks were distributed to the processors in this technique

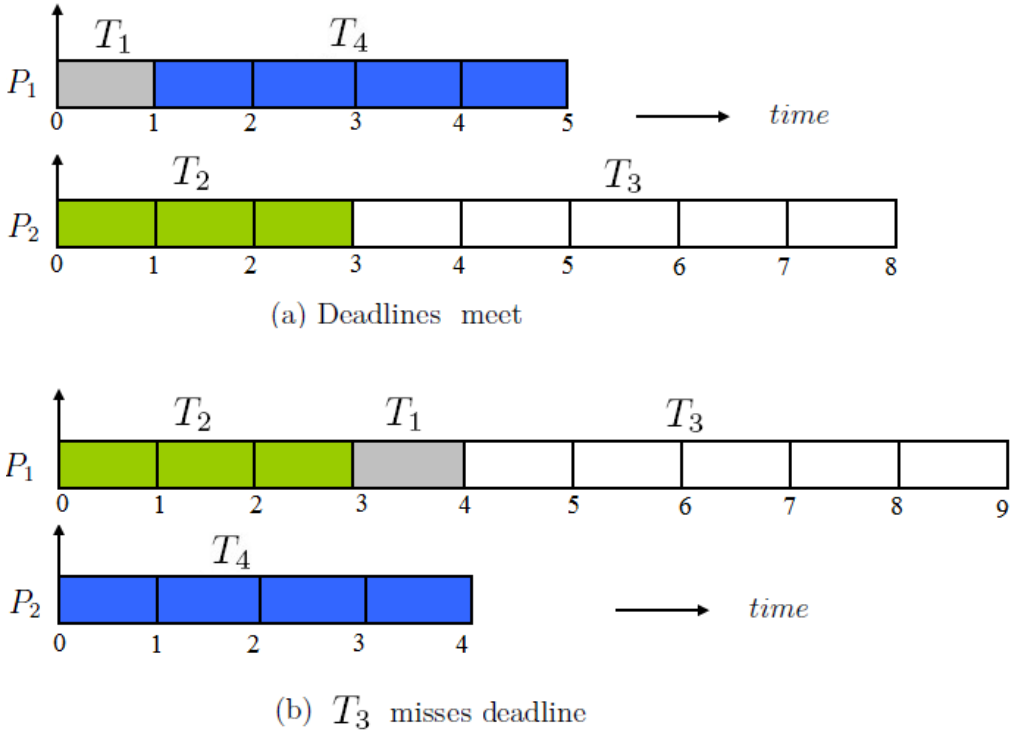


Figure 2.3: Scheduling anomaly

and then efficient uniprocessor scheduling technique were used to schedule tasks on that processor (partitioned technique).

Another important event, which influenced the direction of research in multiprocessor scheduling, was the the publication of results of Dhall and Liu [27] in 1978. They used EDF for scheduling periodic taskset with implicit deadlines on M processors using global scheduling. The taskset consisted of M tasks with small periods and infinitely small utilization factor and a single task with utilization factors that approaches to 1. They showed that the utilization bound for this scheduling algorithm is 1+ϵ, for an arbitrary small value of ϵ. The taskset was not schedulable even though the total utilization factor was much less than the number of processors. This was named the *Dhall effect*. It says that some tasksets may be unschedulable even though they have low utilization (much less than the number of processors).

To understand the Dhall effect, consider the scheduling of the taskset given in table 2.3. We use EDF for scheduling of a taskset with three tasks on a multiprocessor system comprising two processors. Suppose ϵ is a number with a very small value close to zero. The total utilization factor of the taskset is very close to 1 as given by following equation.

Task	$T_i.p = T_i.d$	$T_i.e$	$T_i.u$
T_1	1	2ϵ	2ϵ
T_2	1	2ϵ	2ϵ
T_3	1+ϵ	1	$\frac{1}{1+\epsilon}$

Table 2.3: Taskset for example of Dhall effect

$$M * 2\epsilon + \frac{1}{1+\epsilon} \underset{\epsilon \to 0}{\to} 1 \tag{2.6}$$

Although the total utilization factor is much less than 2, this taskset is not schedulable using EDF. The schedule is shown in the figure 2.4. Tasks T_1 and T_2 are given higher priorities than T_3 as they have early deadlines. T_3 which starts after the completion of $T_1$and T_2 misses its deadline.

After the Dhall effect was found, an impression was built that global algorithms suffer from this effect and are inferior to the partitioned algorithms. Consequently, the research remained focus mainly on the partitioned algorithms. However, later on it was found on the bases of work of Philips et al. in 1997 [54] and Funk et al. in 2001 [32] that Dhall effect was more a problem of high task utilization. Thereafter, global scheduling algorithms got the attention of researchers.

Proportionate fair, or simply *PFair,* was presented by Baruah et al. in 1996 [56]. *PFair* is a global scheduling algorithm which is shown optimal for periodic taskset with implicit deadlines. It is based on the concept of fairness which will be explained later in 2.4.2.2. After *PFair* a number of other optimal scheduling algorithms were also proposed as a consequence of other research work.

In a multiprocessor system, periodic taskset with implicit deadlines is schedulable on M processors if and only if the following two conditions are true:

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq M \tag{2.7}$$

$$\forall i \;\; T_i.u_{(max)} \leq 1 \tag{2.8}$$

Classification of multiprocessor scheduling algorithms As already discussed, the main classification of multiprocessor scheduling is based on the extent of migration allowed. Multiprocessor scheduling can be classified into three broad categories.

- Partitioned scheduling

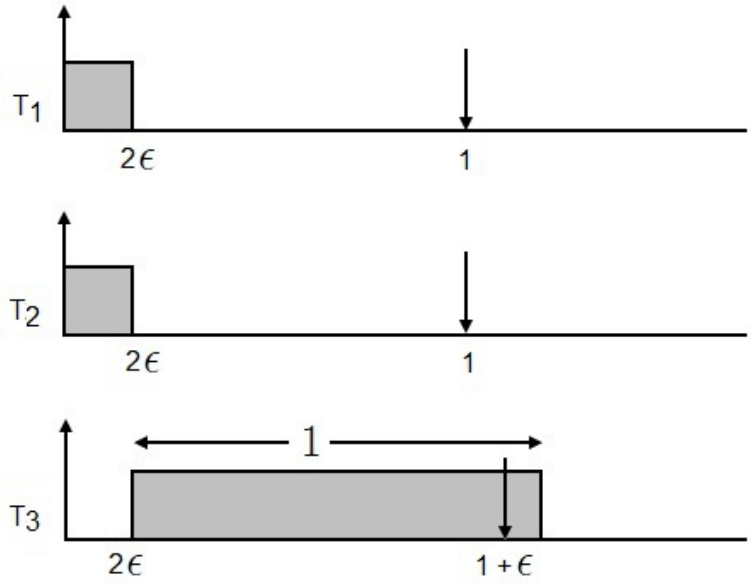


Figure 2.4: Example of Dhall effect

- Global scheduling
- Hybrid scheduling

Partitioned and global scheduling have been already defined. Hybrid lies in between of the Partitioned and global scheduling algorithms. All of these scheduling techniques will be discussed in detail in the following sections.

2.4.2.1 Partitioned scheduling

Partitioned scheduling is the most mature type of multiprocessor scheduling algorithms and is being used very successfully after the advent of the multiprocessor scheduling. According to the principle of partitioned scheduling, taskset is partitioned offline into M subsets and each subset is allocated to a single processor. Once the tasks are assigned to a particular processor, their migration to a different processor is not allowed. Hence, each processor has an independent local queue and a scheduler to schedule tasks assigned to it. In short, the partitioned scheduling is divided into two steps.

- Offline partitioning
- Uniprocessor scheduling

The two steps are shown by rectangles in the figure 2.5, the upper rectangle covers the partitioning while the lower highlights the scheduling.

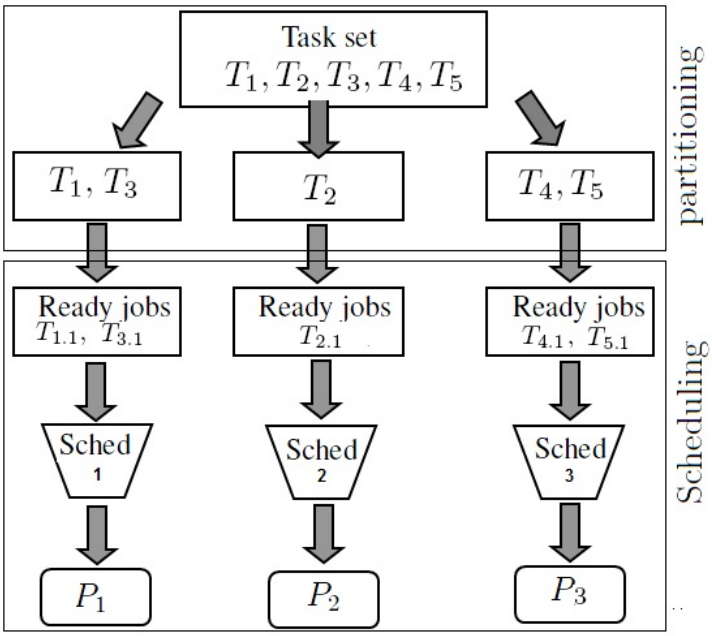


Figure 2.5: Partitioned scheduling

The offline partitioning is analogous to a bin packing problem where a set of items are adjusted into a number of bins such that the total volume allocated to each bin does not exceed its capacity. Bin-packing problem is *NP-hard* [33] in strong sense. In our context, the capacity of bin is related to the schedulability criterion. The processors represent the bins while task utilization factors represent the items. Several bin-packing heuristics have been used to complete the taskset allocation. *First Fit (FF)*, *Next Fit (NF)*, *Best Fit (BF)* and *First Fit with Decreasing Utilization (FFDU)* etc. are few of the commonly used bin-packing solving heuristics. A brief description of two of them is given below. For all of them, the capacity of processors is exceeded when the set of tasks becomes unschedulable. The limit depends on the schedulability algorithm which is used and is specified thanks to schedulability test.

First-Fit (FF): This algorithm allocates a new item to a non-empty bin with the lowest index, such that the size of the new item along with the total capacity of the items allocated to that bin, do not exceed the capacity of the bin.

Best-Fit (BF): This algorithm allocates the task to a free bin if the task cannot be allocated to a non-empty bin. Otherwise, BF will allocate the item to the non-empty bin with smallest capacity available, in which the item can be allocated. If there is more than one bin with the same capacity, then BF will choose the bin with smallest index.

The early research on the partitioned scheduling was done by Dhall and Liu. [27] in late 80s. It was then followed by some significant work of Davari and Dhall [22] in 1986 and Oh and Son in 1993 [51] and 1995 [52]. Burchard et al. [16] used some bin

packing heuristics *Best Fit (BF), First Fit (FF), Next Fit (NF)* etc. along with efficient uniprocessor scheduling algorithms. Few of them are briefly discussed in the following. Their efficiency is mentioned by their approximation ratio.The *Approximation ratio* is a ratio of processor numbers which compares the performance of a specific algorithm when compared with an optimal algorithm. The approximation ratio R_A of an algorithm A is defined as:

$$R_A = \underset{M_0 \to \infty}{max} \left(\underset{\forall \Gamma}{max} \left(\frac{M_A}{M_0} \right) \right) \tag{2.9}$$

where M_0 is the number of processors required by an optimal algorithm and M_A is the number of processors required by algorithm A. The minimum value of approximation ratio can be 1 which means that algorithm is optimal.

Following are given few partitioned algorithms for periodic tasksets with implicit deadlines. The algorithms are referred by abbreviated names. For example *RM-NF* algorithm uses the rules of Rate Monotonic for processor scheduling while using the technique of Next Fit heuristics for task allocation to the processors.

RM-FF [51] It stands for *Rate Monotonic First Fit algorithm*. The approximation ratio is 2.33. The maximum utilization bound is $M(2^{1/2} - 1)$. The computational complexity of *RM-FF* is *O(NlogN)*.

RM-NF [27] It stands for *Rate Monotonic Next Fit algorithm*. The approximation ratio is 2.67. The maximum utilization bound is $\frac{M+1}{2}$. The computational complexity of *RM-NF* is *O(NlogN)*.

RM-FFDU [52] It stands for *Rate Monotonic First Fit with Decreasing Utilization*. The approximation ratio is 1.66. Its maximum utilization bound is $M(2^{1/2} - 1)$. The computational complexity of *RM-FFDU* is *O(NlogN)*.

RM-GT [16] It stands for *Rate-Monotonic General Task* algorithm. It gives the best approximation ratio value of 2.33. The maximum utilization bound for RM-GT is $0.5(M - 1.42)$. The computational complexity of *RM-GT* is *O(NlogN)*.

EDF-FF [33] It stands for Earliest Deadline First and First Fit algorithm. The approximation ratio is 1.7. The computational complexity of *EDF-FF* is *O(NlogN)*.

EDF-BF [33] It stands for *Earliest Deadline First Best Fit* algorithm. The approximation ratio is 1.7. The computational complexity of *EDF-BF* is *O(NlogN)*.

There are many advantages of partitioned scheduling. It is easy to implement as it reduces the multiprocessor problem to a uniprocessor problem. Once the taskset is partitioned among the processors, the problem is reduced to uniprocessor scheduling and thereafter it uses the advantages of uni-processor algorithms [40, 39, 33]. The well studied uniprocessor scheduling algorithms like EDF, RM, DM etc. are thus equally applicable to the partitioned scheduling. Since there is no migration of tasks between the processor there is no migration overhead in partitioned scheduling and it gives improved

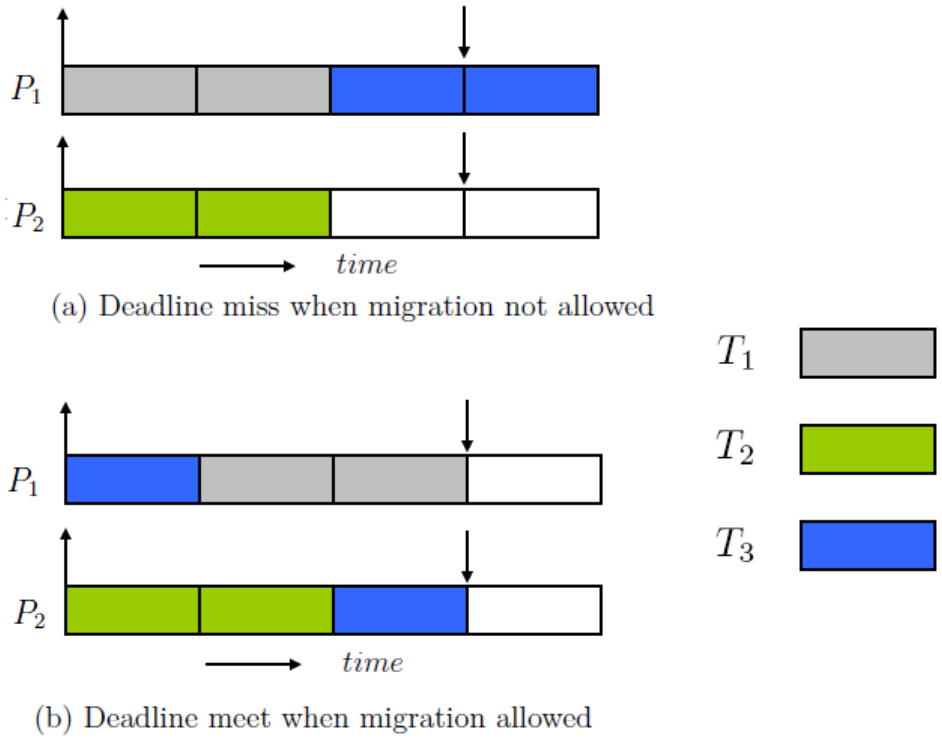


Figure 2.6: Taskset is schedulable with migration

cache performance. Each processor has an independent scheduler which schedules the ready jobs on that processor. If a task overruns its worst case execution time, then it only affects the other tasks on the same processor.

The main disadvantage of partitioned scheduling is that it is inherently sub-optimal and therefore does not guarantee the complete utilization of the resources. The maximum utilization bound for any partitioning algorithm for periodic tasks with implicit deadlines is [12]:

$$U_P = \frac{M+1}{2} \tag{2.10}$$

The partitioned algorithm is not an ideal choice for the system where there are frequent arrivals and exits of new tasks because with the addition of new tasks all the system has to be repartitioned which will result in some run time overhead.

In the worst case scenarios partitioned scheduling algorithms do not show good performance. There are some tasksets which are schedulable only when they are not parti-

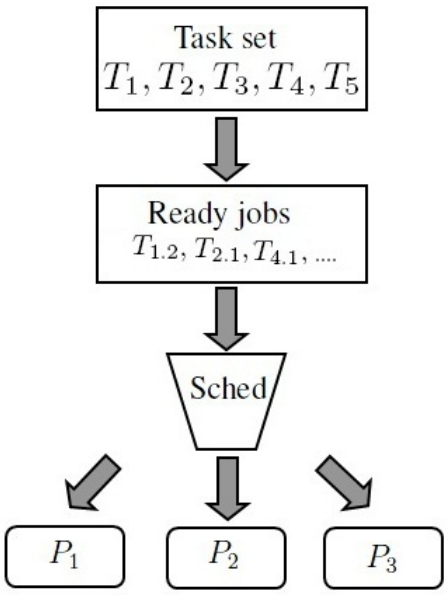


Figure 2.7: Global scheduling

tioned. Consider a taskset consisting of three tasks each with an execution requirement of two units of time and relative deadline of three units of time, it cannot be scheduled on two processors without task migration. This is shown in the figure 2.6. In the figure 2.6 (a), deadlines are missed because the task migration is not allowed. In the figure 2.6 (b), the deadlines are achieved since task migration between the processors is achieved.

2.4.2.2 Global scheduling

In global scheduling, there is a single queue of ready tasks for all the processors and there is no pre allocation of tasks to the processors. A single scheduler is responsible for scheduling all the taskset. The migration of tasks between the processors is allowed. A task can execute on one processor and, after a preemption, can be resumed on a different processor. Some global scheduling algorithms allow only task level migration while the others allow both task as well as job level migration. The global scheduling is shown in figure 2.7.

The global scheduling can be classified in optimal and non-optimal scheduling algorithms.

Non-optimal global scheduling:

This type includes the fixed task priority and fixed job priority scheduling.

B. Anderson et al. [9] carried out a research for periodic tasksets with implicit deadlines and showed that the maximum utilization bound of any fixed job priority

global scheduling algorithm is given by following equation:

$$U_{FJP} = \frac{(M+1)}{2} \tag{2.11}$$

This is equal to the maximum utilization bound for a partitioned algorithm.

Besides global RM, global DM and global EDF some specified global scheduling algorithms have been proposed.

B. Andersson and Jonsson [7] presented a TkC policy in 2000 which suggests an assignment technique to prevent Dhall effect. The algorithm assigned a new priority based on parameter k which is a function of the number of processors. They showed that it is an effective priority assignment policy for periodic tasks with implicit deadlines.

In 2003, B. Andersson et al. [9] proposed *RM-US*(ς) algorithm where the tasks with utilization factor higher than ς are given highest priorities and ties are broken arbitrarily. The scheduling is performed using rate monotonic algorithm. B. Andersson et al. showed that *RM-US*($\frac{M}{3M-2}$) has maximum utilization bound of $\frac{M^2}{3M-2}$.

In 2008 B. Andersson and Jonsson proposed a slack monotonic algorithm *SM-US* [8], where the priorities of the tasks were assigned according to the order of slack i.e. $T_i.p - T_i.e$. They showed that *SM-US* [$\frac{2}{3+\sqrt{5}}$] has a at most utilization bound o f $0.382M$.

EDF-US(ς) was proposed for fixed job priority by Srinivasan and Baruah [58] in 2002. Tasks with a higher utilization factor value than threshold ς are given highest priority and ties are broken arbitrarily. In 2005, Baker [13] showed that if this threshold is fixed to be 1/2, then the fixed job priority algorithms give the maximum utilization bound is equal to U_{FJP}.

$EDF(k)$ is another algorithm presented by J. Goosens et al. [34] in 2003. In this technique, k tasks with highest utilization factor are given priority. They showed a sufficient condition for the schedulability of $EDF(k)$:

$$M \geq (k-1) + \left\lceil \frac{U - T_k.u}{1 - T_k.u} \right\rceil \tag{2.12}$$

where $T_k.u$ is the utilization factor of k_{th} task, sorted in order of decreasing utilization factor. Baker [13] also proposed a modified model of $EDF(k)$ which he named $EDF(k_{min})$ where k_{min} is the value for which the sufficient condition of schedulability is valid. Baker showed that the maximum utilization bound of $EDF(k_{min})$ is $(M+1)/2$.

Optimal global scheduling:

The algorithms discussed upto now are based on work conserving techniques. However, the global techniques are based on both non-work conserving and work conserving algorithms. There are two main branches of optimal global scheduling algorithms for periodic tasks sets with implicit deadlines, $PFair$ [56] and Deadline Partitioning Fair

which is known as *DP-Fair* [43]. Both categories are based on the principle of fairness but they differ on how much fairness is required.

In a fair task scheduling, tasks are required to make progress at steady rates. PF, PD and PD^2 are three $PFair$ algorithms which are proven to be optimal [56]. *ER-Fair* [5] which is a work-conserving technique is an extended form of $PFair$.

The well known non-work conserving algorithms following the principle of *DP-Fair* are *DP-Wrap* [43], Boundary fair [61], $LLREF$ [36] and *LRE-TL*[30]. $TRPA$ and $ETNPA$ are two work conserving optimal scheduling techniques. These optimal scheduling algorithms are explained in detail in next chapter.

Global algorithms have some advantages and disadvantages.

Migration increases the schedulability which allows better utilization of resources. The global scheduling can perform load sharing between processors. Load sharing is the transfer or migration of a task to a processor which is idle or relatively less busy [20]. An important advantage is that a number of global scheduling algorithms are found to be optimal for periodic tasks with implicit deadlines. Global scheduling is better for dynamic systems where there is frequent arrival and leaving of tasks.

The main disadvantage of the global scheduling is the migration overhead. For implementing a migration, the system preempts the task, saves the data, transfers the saved data to a new processor, and restarts the task on the new processor. Hence, an overhead is involved in migration of a running task [20]. With an increase in the number of migrations, a lot of traffic is generated that causes an extra load on the buses, the cache and causes delays in the overall system. As a consequence, in the programming of schedulers, special attention is paid to avoid unnecessary migrations.

One of the objections on the global scheduling is that most of the advantages are based only on the theoretical results or some simulation work. Rarely can be founded some experimental results [15] based on some practical work. One of the experimental results was presented by Brandenburg et al.[15] in 2008. In this research, several global and partitioned scheduling algorithms were evaluated on Sun Niagara multicore platform with 32 logical CPUs.

2.4.3 Hybrid scheduling

The partitioned scheduling algorithms have low overheads but do not guarantee high utilization bounds. On the other hand, the global scheduling algorithms give better utilization bound at the cost of high amount of overheads. Therefore, the hybrid scheduling is a compromise between partitioned scheduling and global scheduling while retaining the qualities of both algorithms. It gives better utilization bound than partitioned while lowers the overhead as compare to the global techniques. This is because hybrid algorithms allow some types of migration while prohibiting others to improve the utilization

bound. It includes some semi-partitioned and clustering techniques.

EKG [10] is a semi partitioned technique presented by B. Anderson and Tovar for periodic tasks with implicit deadlines. It stands for EDF with task splitting and k processors in a Group. The processor set is divided into $\lfloor \frac{M}{k} \rfloor$ clusters with k processors in each cluster. The taskset is also divided into $\lfloor \frac{M}{k} \rfloor$ subsets with the total utilization factor of each subset smaller than k. This division is done using a bin-packing algorithm. Some tasks are fixed to the processors and are scheduled using EDF. The others are allowed to execute on two different processors of the same cluster. Maximum utilization bound of EKG depends on the parameter k which controls the division of tasks.

$$U_{EKG} \leq \begin{cases} \frac{k}{k+1}M \ \forall & k < M \\ M & k = M \end{cases} \tag{2.13}$$

For $k = M$, *EKG* is optimal. For k = 2, the maximum utilization bound becomes $\frac{2}{3}M$.

Clustering is another hybrid scheduling technique. In this technique the processors are divided into M° clusters where $1 \leq$ M° $\leq$ M. The tasks are statically assigned to the clusters. The tasks in each cluster are scheduled using a global technique. If the number of processors in each cluster is 1, clustering becomes same as partitioned schedule. On the other hand if there is only one cluster of all the processors, it becomes the same as global scheduling. Clustering has better maximum utilization bound than partitioned scheduling. At the very same time, a fewer number of processors in each cluster decreases the length of global queue and has a potential to reduce migration overheads depending on the architecture. For example, the processors in the same cluster may share the same common cache thus reducing the miss penalty. One of the clustering techniques, named Earliest Deadline Deferrable Portion (EDDP), was proposed by Shin et al. [41] in 2008 where the scheduling within each cluster is performed using EDF. According to this algorithm some tasks are fixed to execute on dedicated processor and are not allowed to migrate while others are allowed to migrate.

2.5 Conclusion

Partitioned scheduling and global scheduling are two main branches of multiprocessor scheduling while the hybrid scheduling is a compromise between the two. The global algorithms theoretically guarantees higher utilization factor than both partitioned scheduling and hybrid scheduling and dominate them. Further in global scheduling, there are few techniques which are optimal. Though theoretically optimal, questions are raised about their practical significance.

In our research work, we use the optimal global scheduling to schedule the periodic taskset with implicit deadlines on symmetric multiprocessor system.

The main objection on the optimal global scheduling is that optimality is achieved at the cost of frequent scheduling points, migrations and preemptions. Migrations cause cache miss and excessive load on the data buses between the processors while preemptions lead towards context switches. These overheads in the system results in an increase in worst case execution times of the tasks which results in the missing of deadlines. Theoretically the cost of overheads due to preemptions and migrations are considered to be negligible but practically it cannot be neglected in the system. On some modern multicore architectures, the cost of migration is much lower than in past but still a nonzero value. Hence these overheads need to be managed to fully exploit the benefits of global scheduling.

The objective of our work is to devise the techniques in global scheduling to minimize these overhead to a possible minimum level while sustaining the optimality. We have used some simple heuristics which achieve this objective with out increasing the complexity of the original algorithms.

3

A survey of optimal global algorithms

Overview

In this chapter, a detailed overview of the optimal global scheduling algorithms is given. The working, advantages and disadvantages of each of them are discussed. It also presents a comparison between them while highlighting important characteristics. It gives a classification of optimal scheduling algorithms called *DP-Fair*. It also discusses the recently proposed optimal algorithm called RUN which uses a very weak version of fairness.

3.1 Introduction

All optimal scheduling algorithms for real-time multiprocessor systems belong to the global scheduling category using dynamic job priority. As discussed in the previous chapter, the global scheduling is the one where there is a single queue of ready tasks for all the processors and there is no pre-allocation of tasks to the processors. A single scheduler is responsible for scheduling the set of all ready tasks and it allows migration of tasks between the processors. We recall that a scheduling algorithm is said to be optimal for a class of taskset and hardware model if it provides a correct schedule for any feasible taskset belonging to that class.

Until recently, the two main classes of optimal global scheduling algorithms for periodic tasksets with implicit deadlines in symmetric multiprocessor system were *PFair* [56] and *DP-Fair* [43]. However another scheduling algorithm *RUN* [55] has been proposed in the end of 2011 which has proven to be optimal. The major portion of the chapter deals with *PFair* [56] and *DP-Fair* [43] while a minor part deals with RUN algorithm.

Both *PFair* and *DP-Fair* achieve the optimality by following the core idea of tracking the fluid schedule. The basic concept of fluid and practical schedules is shown in the figure 3.1 which shows the execution of a job of the task T_i. The job has to finish its execution time $T_i.e$ before its deadline $T_i.d$. The fluid schedule is shown by dotted line with a constant slope of $-T_i.u$. The practical execution, represented by the solid line, has a slope -1 when a job is running and a slope 0 when a job is waiting.

Both the *PFair* and *DP-Fair* are interval-based scheduling strategies. However, these two differ in :

1. The interval length;
2. The precision with which fluid schedule is tracked.

Both *PFair* and *DP-Fair* are explained in detail with the principles, benefits and short comings of each in the next sections.

3.2 Proportionate Fair scheduling

3.2.1 Principle

Proportionate fair, or simply *PFair*, was presented by Baruah et al. in 1996 [56]. *PFair* is an optimal global scheduling algorithm for periodic tasksets with implicit deadlines. It is based on the concept of *fairness*. Different to that of an equally distributed

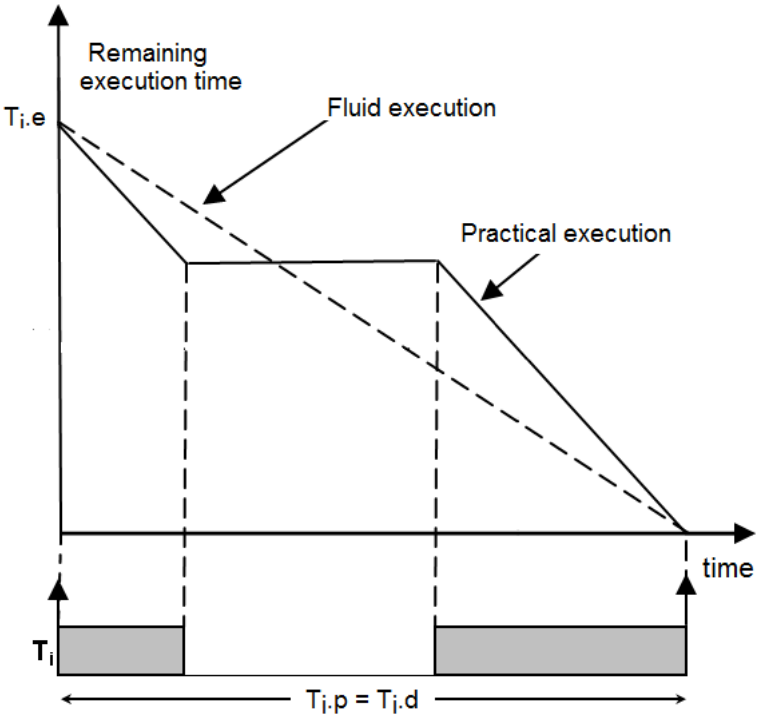


Figure 3.1: Fluid versus real execution

system where the processor is equally shared among all the tasks, each task gets a processor share proportional to its utilization factor.

The concept of fairness is given in the figure 3.2. It shows the execution of two tasks T_1 with $T_1.u = 0.6$ and T_2 with $T_1.u = 0.4$ on a processor P. Ideally each task should get a share of the processor capacity proportional to its utilization factor at any instant of its execution and strictly follows its fluid schedule as shown in figure 3.2 (a) . At time 5, T_1 should (5×0.6) = 3 units of time and T_2 gets (5×0.4) = 2 units of processor. However, practically a processor can execute only one task at any given time with its entire capacity. A way to make the real execution as close as the fluid one is to divide the task execution into small time intervals as shown in the figure 3.2 (b).

In *PFair* schedule, the time is divided into small intervals of equal lengths called *quanta*. The length of each quantum is equal to the smallest unit of time. The time interval $[t,\, t+1)$ where $t \geq 0$, is called a *slot* t. So as to measure the difference between ideal and practical executions the lag function is introduced which is defined for each task. The lag for a task T_i at time t is defined as :

$$lag(T_i,\, t) = T_i.u * t - \sum_{l=0}^{t-1} S(T_i,\, l) \tag{3.1}$$

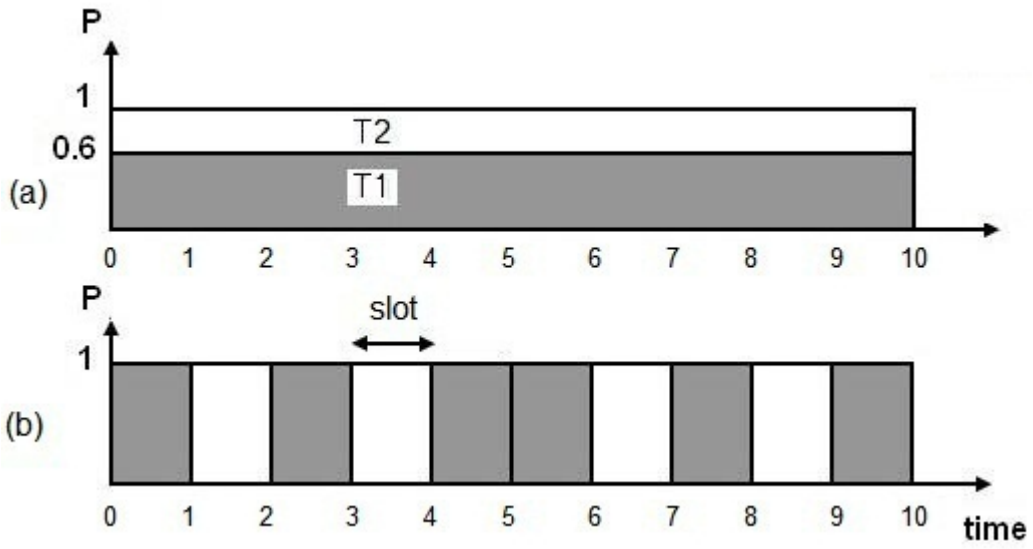


Figure 3.2: (a) Fluid schedule (b) Time division with $T_1.u$ = 0.4 and $T_2.u$ = 0.6

where $T_i.u * t$ represents the total executed time of task T_i at time t in an ideal system. S is the real schedule function defined for each slot of time. $S(T_i, l) = 1$ means that task is scheduled in the slot l while $S(T_i, l) = 0$ shows that task is not scheduled over the slot. $\sum_{l=0}^{t-1} S(T_i, l)$ is the total executed time of task T_i at time t in the real sequence. In figure 3.2 $\sum_{l=0}^{t-1} S(T_1, l)$ is 1 at t = 2 and 4 if t = 6. In the figure 3.2, the lag of task T_1 and T_2 at time 8 can be calculated as follow :

$$lag(T_1, 8) = (0.6 \times 8) - 5 = -0.2$$
$$lag(T_2, 8) = (0.4 \times 8) - 3 = 0.2$$

A taskset execution is *PFair* if it fulfills the following condition on lag :

$$-1 < lag(T_i, t) < 1 \quad \forall\, t > 0 \; and \; \forall\, i \tag{3.2}$$

The *PFair* performs the scheduling of tasks on the processors such that at any given time t, accumulated time allocated to a task T_i with utilization factor $T_i.u$ is either $\lfloor t * T_i.u \rfloor$ or $\lceil t * T_i.u \rceil$. It means that in *PFair* a task may not be more than a quantum length away from its ideal execution in the fluid system at any time.

If the condition in equation 3.2 is met, all the deadlines will be met and taskset will be schedulable. It can be easily demonstrated :

At $t = T_i.p$, when $T_i.u * T_i.p = T_i.e$, lag function can be rewritten as :

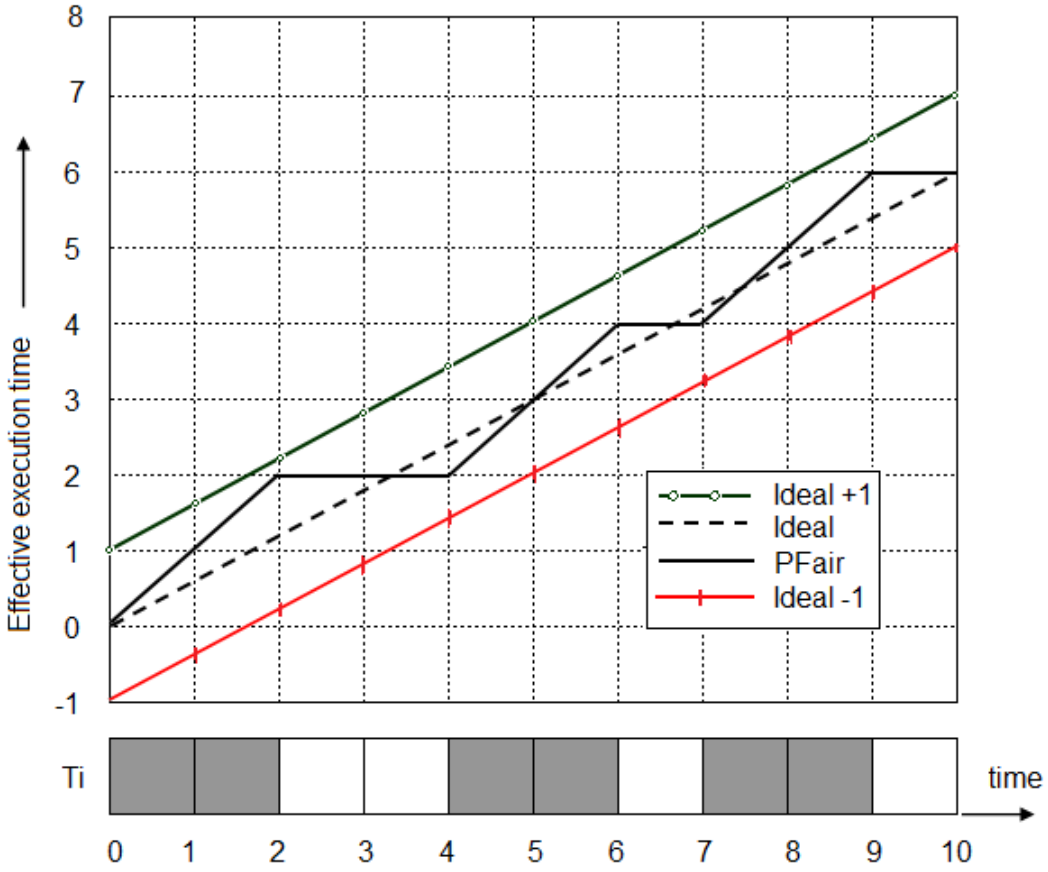


Figure 3.3: Execution of task in PFair

$$lag(T_i, T_i.p) = T_i.e - \sum_{l=0}^{T_i.p-1} S(T_i, l)$$

We know that both $T_i.e$ and $\sum_{l=0}^{T_i.p-1} S(T_i, l)$ are integers. The only possibility to satisfy the lag condition given in equation 3.2 is that both have the same value and their difference is zero. In other words, both the practical and fluid schedules converge on the same point. Since we are considering implicit deadline tasks therefore the deadlines are also met.

The figure 3.3 shows the execution of a task T_i with utilization factor $T_i.u$=0.6 that is *PFair*. This execution representation is different from the one mentioned in figure 3.1 in the sense that it has effective execution time on the vertical axis rather than remaining execution time. The straight dotted line in the middle with uniform slope shows the fluid schedule of the task. Two parallel lines across both sides of fluid schedule show the limitations defined by condition in equation 3.2. *PFair* schedule of the task is represented by dotted line.

Baruah proposed the following simple test to check out if a *PFair* schedule is possible or not.

Theorem [56] A *PFair* schedule on M processors exists if and only if the total utilization factor of the taskset is less than or equal to the number of processors. i.e. :

$$U \leq M \tag{3.3}$$

3.2.2 PFair scheduling

After division of the time into quanta, *PFair* performs the scheduling of tasks in each quantum. All the tasks are scheduled at the start of each quantum. For scheduling, each task T_i is divided into sub-tasks and each sub-task needs a quantum length of a processor to complete its execution. $T_{i.k}$ represents the k_{th} sub-task of task T_i where $k \geq 1$.

Practically to build a *PFair* execution, a pseudo-release $r(T_{i.k})$ and pseudo-deadline $d(T_{i.k})$ can be associated to each sub-task that will conduct the scheduling. Mathematically, they can be defined as :

$$r(T_{i.k}) = \left\lfloor \frac{k-1}{T_i.u} \right\rfloor \tag{3.4}$$

$$d(T_{i.k}) = \left\lceil \frac{k}{T_i.u} \right\rceil \tag{3.5}$$

where k is the index of the sub-task. Sub-task execution must respect the precedence order: k_{th} sub-task must execute before $(k+1)_{th}$ sub-task. The time between the pseudo-release and the pseudo-deadline of a sub-task is known as its *window*. The window for the execution of k^{th} sub-task of task T_i is given by following equation.

$$w(T_{i.k}) = [r(T_{i.k}), d(T_{i.k})) \tag{3.6}$$

Each sub-task must execute within its specified window to assure *PFair* execution. The length of the windows depends on the utilization factor of the task. Higher is the utilization factor, smaller is the window length and vice versa. The minimum length of the window of a sub-task is 1 when the utilization factor of the task is 1.

The task cannot be started before the start of its pseudo-release time and it cannot continue execution after its pseudo-deadline. In case of execution of sub-task after the pseudo-deadline, schedule is no more a *PFair* schedule. If a sub-task is executed in the earlier slots of its window then the next sub-task will not become ready before the start of its windows [4, 5]. This causes *non-work conserving* behavior of *PFair* scheduling.

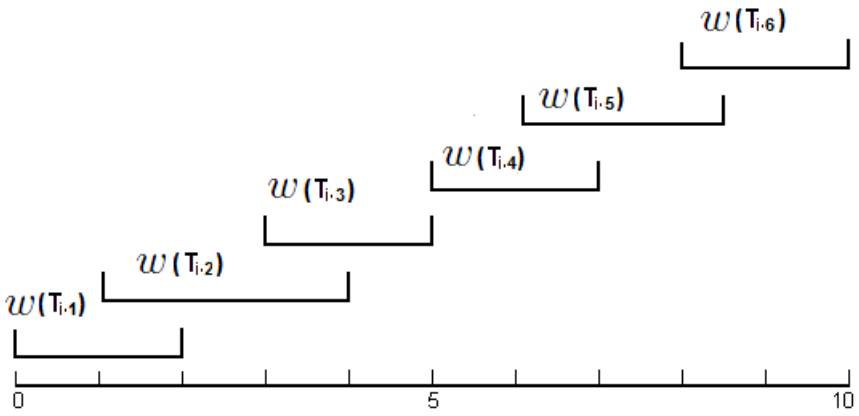


Figure 3.4: Windows in PFair for a task with $T_i.u = 0.6$

Suppose we have a task with utilization factor $T_i.u = 0.6$. Windowing of some sub-tasks of this task is shown in the figure 3.4 as calculated by equation 3.4 and 3.5. For example, the window for sub-task $T_{i.3}$ is calculated as follow:

$$r(T_{i.3}) = \left\lfloor \frac{3-1}{6/10} \right\rfloor = 3$$
$$d(T_{i.3}) = \left\lceil \frac{3}{6/10} \right\rceil = 5$$

The scheduling decisions in *PFair* are taken at the start of each quantum. At this point, M ready sub-tasks with highest priority are executed on M processors. A sub-task $T_{i.k}$ is said to be ready at time t if it satisfies the following two conditions:

1. The previous sub-task $T_{i.(k-1)}$ is already completed. This condition is not required for the first sub-task of each job;

2. The time t must be equal to or more than the pseudo-release of the sub-task i.e. $r(T_{i.k})$.

The priority of sub-tasks are defined by a set of rules. Since all of these rules are applied at same time for all the taskset, *PFair* is also known as synchronized tick scheduling or quantum based scheduling.

Like most of the other scheduling algorithms, *PFair* scheduling does not give any information about the processor allocation to the task. A task can execute on any processor and after the preemption it may resume on a different processor.

3.2.3 PFair algorithms

PF, PD and PD^2 are three *PFair* algorithms which are proven to be optimal [56]. All PF, PD and PD^2 use early pseudo-deadline first $(EPDF)$ algorithm to find out the highest priority sub-tasks among ready sub-tasks but then they use different tie breaking rules.

Though optimal, PF is inefficient due to recursion involved in calculation of tie breaking rules which are different to that of PD^2. In PD the group-deadline calculation is replaced by three additional rules and hence the calculation is complex.

The PD^2 is considered to be very efficient and it uses three simple rules to find out the sub-tasks with highest priority. The complexity of PD^2 schedule on M processors is $O(MlogN)$ [56].

The priority rules of PD^2 algorithm are defined for two ready sub-tasks $T_{i.a}$ and $T_{j.b}$ of two different tasks in the following way:

- The sub-task with the earlier value of pseudo-deadline is given priority. $T_{i.a} \succ T_{j.b}$ i.e. $T_{i.a}$ has higher priority than $T_{j.b}$ if :

$$d(T_{i.a}) < d(T_{j.b}) \tag{3.7}$$

- If the two sub-tasks have same value of pseudo-deadline, the sub-task with higher value of successor bit is given priority. $T_{i.a} \succ T_{j.b}$ if :

$$d(T_{i.a}) = d(T_{j.b}) \;\&\&\; b(T_{i.a}) > b(T_{j.b}) \tag{3.8}$$

  where $b(T_{i.a})$ is the successor bit of sub-task $T_{i.a}$ and is defined lateron.

- Finally, if the two sub-tasks have same value of pseudo-deadline and successor bit, priority is given to the sub-task with smaller value of group deadline. $T_{i.a} \succ T_{j.b}$ if :

$$d(T_{i.a}) = d(T_{j.b}) \;\&\&\; b(T_{i.a}) = b(T_{j.b}) \;\&\&\; G(T_{i.a}) < G(T_{j.b}) \tag{3.9}$$

  where $G(T_{j.b})$ is the group deadline of the sub-task $T_{j.b}$.

- The ties are broken arbitrarily if the group deadlines are same.

Successor bit of a sub-task $T_{i.k}$ is defined as [56]:

$$b(T_{i.k}) = \left\lceil \frac{k}{T_i.u} \right\rceil - \left\lfloor \frac{k}{T_i.u} \right\rfloor = d(T_{i.k}) - r(T_{i.k+1}) \tag{3.10}$$

Informally $b(T_{i.k})$ denotes the number of time units by which window of sub-task $T_{i.k}$ overlaps the window of sub-task $T_{i.k+1}$. The value of the successor bit is either 0 or 1. The value of successor bit for sub-task $T_{i.1}$ is 1 and 0 for sub-task $T_{i.3}$ in figure 3.4.

The value of successor bit equal to 1 means that if the sub-task happens to be executed in the last slot of its window, the next sub-task will have "smaller space" in its window. Therefore, this sub-task is given higher priority compare to the one with smaller value of successor bit.

$G(T_{i.k})$ is the group deadline of the sub-task $T_{i.k}$. Group deadline is taken in account only for heavy tasks i.e. the ones for which the utilization factor is more than half. Otherwise group deadline is taken as zero. The group deadline $G(T_{i.j})$ for a sub-task $T_{i.j}$ is the earliest time t where $t > d(T_{i.j})$ such that either:

$$\forall\ k > j \left\{ \begin{array}{c} t = d(T_{i.k})\ when\ b(T_{i.k}) = 0 \\ t+1 = d(T_{i.k})\ when\ |w(T_{i.k})| = 3 \quad for\ any\ subtask \end{array} \right\}$$

$|w(T_{i.k})|$is the length of the window of that subtask.

In the figure 3.4, the group deadline of sub-task $T_{i.2}$ is 5 because this is the deadline of first following sub-task with zero successor bit. The group deadline for sub-task $T_{i.3}$ is 8 because the next sub-task with window length 3 is $T_{i.5}$ and its deadline is 9.

3.2.4 PFair implementation

Division of tasks in sub-tasks is not always necessary for implementing *PFair*. It can be implemented by directly manipulating with the task. It can avoid the calculation and storing of parameters involved with each sub-task.

We implemented *PFair* PD^2 for the STORM [1], a simulation tool [2, 60]. We use some work done by Chandra et al. [19] in this regard.

Figure 3.5 shows the execution of a taskset with total utilization factor equal to the number of processors which is 2. The task configuration is given in the table 3.1. The figure shows only first 50 time units. The *PFair* successfully schedules the set of tasks with all deadlines met. The first two rows show the execution of tasks on processors while the last three rows show the execution of single task. The simulator used for this experiment does not take in account the system delays due to task migration and scheduling overhead.

3.2.5 Limitations of PFair

Few of the objections on the *PFair* scheduling are:

1. Though theoretically *PFair* is optimal, it achieves this optimality at a very huge runtime overhead due to frequent switching, preemptions and migrations [57].

1. See chapter 5 for further details about STORM.

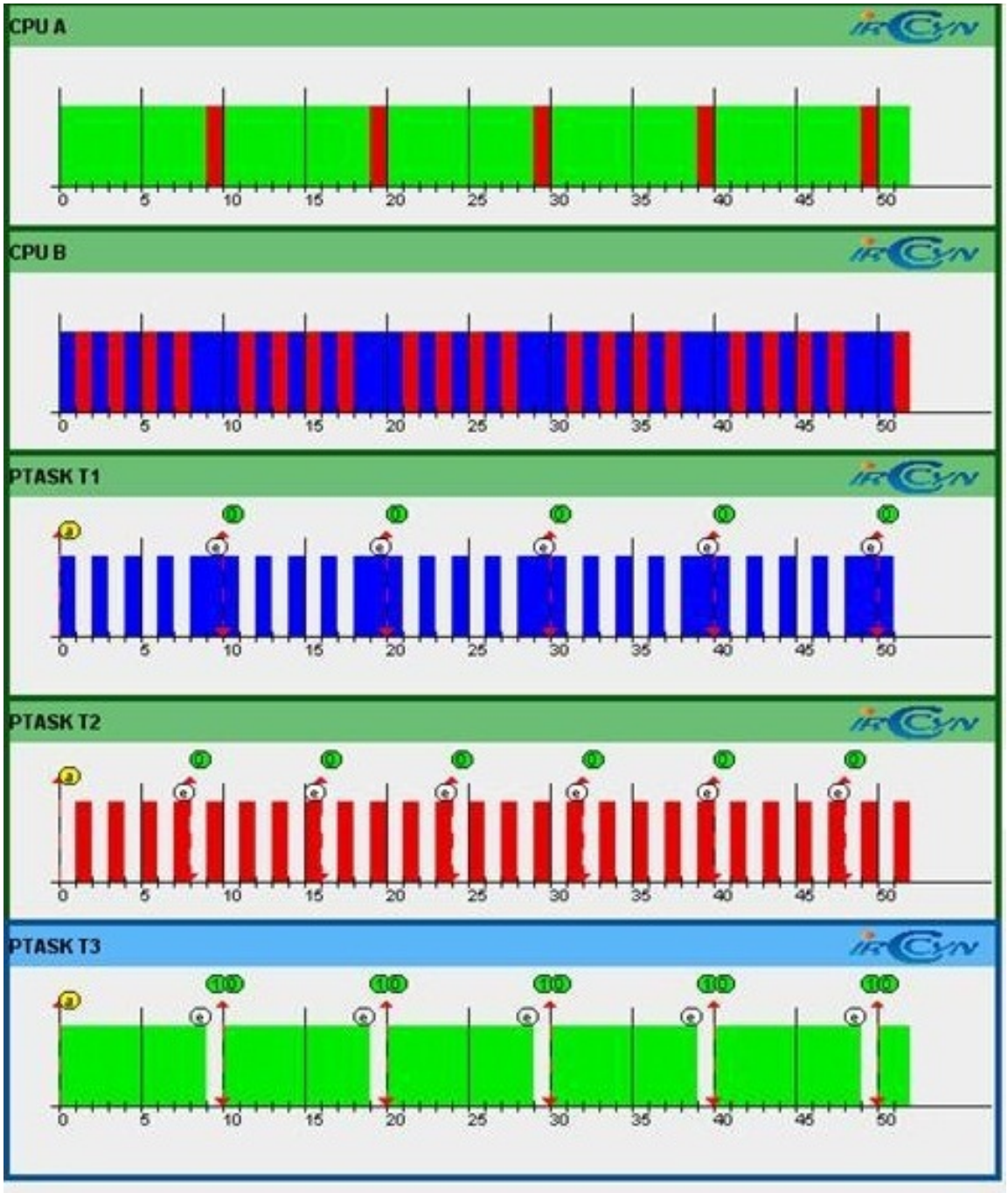


Figure 3.5: An example of a PFair execution sequence

Task	$T_i.p = T_i.d$	$T_i.e$
T_1	10	6
T_2	8	4
T_3	10	9

Table 3.1: Taskset configuration for figure 3.5

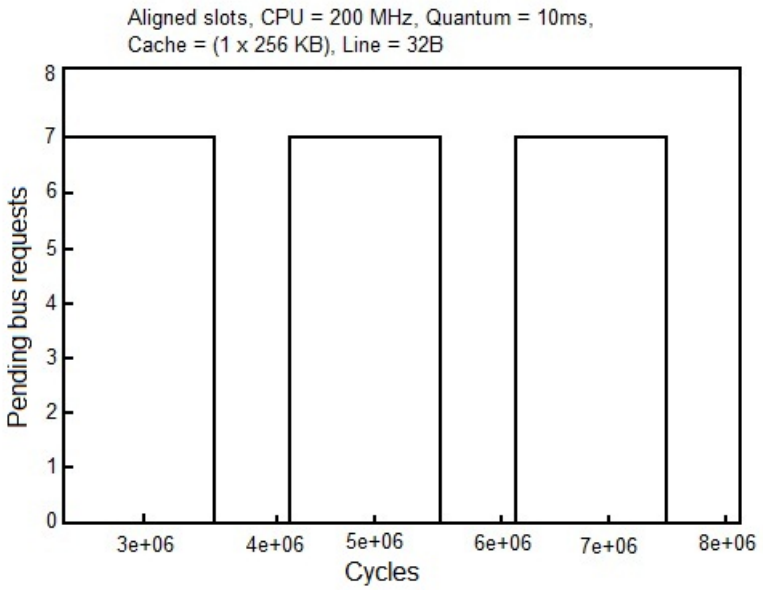


Figure 3.6: Bus load across scheduling points in 8 processors system in PFair [37]

In the worst case, a task may be preempted or migrated each time when it is scheduled. Practically the cost of overhead in *PFair* is unbearable and it results in poor cache performance. Consequently, *PFair* scheduling is often dismissed as an impractical approach;

2. *PFair* scheduling is based on synchronized tick. Scheduling of all the tasks occurs periodically at the start of each quantum. At this point, all the processors are simultaneously scheduled. In worst case *PFair* scheduling do not favor shared-bus architecture. The figure 3.6 [37] shows the result of an experiment where 8 processors were scheduled using *PFair*. The simulated environment consisted of 200 MHz i86 processors using a 10 millisecond tick size and a simple blocking, direct-mapped cache with 256 KB capacity and 32-byte lines. The bus contention was measured by pending bus requests over an interval containing three scheduling points. The results show a significant load on the buses at the start of each quantum. Thus heavy contention follows each scheduling point;

3. To ensure optimality, *PFair* allocates processor time units in fixed-size quantum. These are *synchronized quantum with fixed size* (SFQ) across all the processors. It may lead sometimes towards wastage of quantum. There are various reasons to address this limitation [25]. Firstly, the worst case execution time estimates or WCETs are generally pessimistic, many jobs will execute for less than their $WCETs$. When a job completes before the next quantum boundary, the rest of that quantum (on the associated processor) is wasted. Secondly *PFair* uses periodic timer interrupts that points out quanta to be synchronized across all the processors. Drift in the timing of interrupts on any processor is propagated to other processors as well. Thirdly, the scheduling decision is made only by one processor while the rest of processors remain idle.

But the above discussed behavior does not always exist because of the following reasons:

1. The empirical evidences exist now which show that *PFair* is practical alternative to other approaches. Calandrino et al. [18] conducted an experiment on four processor Linux testbed evaluating multiprocessor real-time scheduling algorithms. The tested algorithms were compared based on both raw performance and schedulability (with real overheads considered). The results showed that *PFair* scheduling is either competitive with or outperforms other approaches including partitioning and global scheduling based *non-PFair* algorithms;
2. The newly design processors architecture favors design that can considerably mitigate overheads and hence are very well suited for *PFair* scheduling [26]. Advent of multicore architecture is most significant in this regard. It has multiple cores on a single chip and use various levels of on and off caches of considerable capacity. The tasks do not significantly lose cache affinity when they migrate hence overhead due to migrations are of less concern;
3. To avoid deadline miss, tie breaking rules are used in addition to the $EPDF$ in *PFair* optimal algorithms. Srinivason and Anderson observed that overhead due to tie breaking rules is unnecessary for the many soft real-time tasks systems [36]. The computation of finding successor bit and group deadline may be done in constant time but there exist many scenarios where eliminating them is preferable;
4. As described earlier, all the scheduling decisions are taken at the start of fixed size quantum in *PFair* scheduling. A study was carried out by U. C.Devi et al. to find out the effects of relaxing this requirement [25]. They proposed a *desynchronized and variable sized quantum* (DVQ) model to address this problem. They found out that relaxing this limitation, under an otherwise optimal algorithm, results in missing of deadline by at most one quantum only which is sufficient to provide soft real time guarantees;

5. Length of a quantum is an important issue in *PFair*. Smaller is the length of quantum, more uniform is the execution of task but higher is the frequency of scheduling and scheduling overhead. The maximum possible value of quantum is better for the system efficiency. It may be highest common factor (HCF) of all the worst case execution times (WCET) and periods of the tasks. If we have a taskset of three tasks i.e. $T_1(30,\ 20)$, $T_2(25,\ 10)$ and $T_3(40,\ 10)$ following a model $T_i(T_i.p, T_i.e)$ then 5 is the HCF (highest common factor) and may be used as the quantum length. But since the quantum length is considered as the unit, a new configuration will be emerged using this technique. The above discussed configuration will become $T_1(6,\ 4)$, $T_2(5,\ 2)$ and $T_3(8\ ,2)$.

3.2.6 Extensions of PFair

3.2.6.1 Early Release Fair Model

PFair has been extended to work conserving and this is known as *Early Release Fair* or *ER-Fair*.

According to the *ER-Fair* scheduling, other than the first sub-task of a task which has a pseudo-release, any other sub-task is valid to execute after the completion of its previous sub-task. There is no pseudo-release for the sub-task while the pseudo-deadline of the sub-task remains the same as in *PFair*. Figure 3.7 shows the schedule of a task with a utilization factor of 0.6 according to *ER-Fair*.

The mathematical constraint of lag for the Early Release Fair may be written as :

$$lag(T_i, t) < 1 \quad \forall\ t > 0\ and\ \forall\, i \tag{3.11}$$

The processor does not remain idle in the presence of any active task and hence it achieves work conserving behavior.

For the task execution in the figure 3.7 lag function can be defined as:

$lag(T_i, 5)$ = (0.6 $\times$ 5) - 5 = -2

$lag(T_i, 9)$ = (0.6 $\times$ 9) - 5 = 0.4

ER-Fair schedule has a lower number of preemptions as compare to the non-work conserving model. This is due to the fact that when a sub-task is completed, the next sub-task can be started immediately without any preemption provided that there is no waiting task with higher priority.

A hybrid approach can also be established where some tasks are scheduled according to *PFair* while the rest are scheduled by *ER-Fair* [23]. In this case a sub-task may be released earlier but only up to certain slots.

It must be noticed that any *PFair* schedule is an *ER-Fair* but not vice versa.

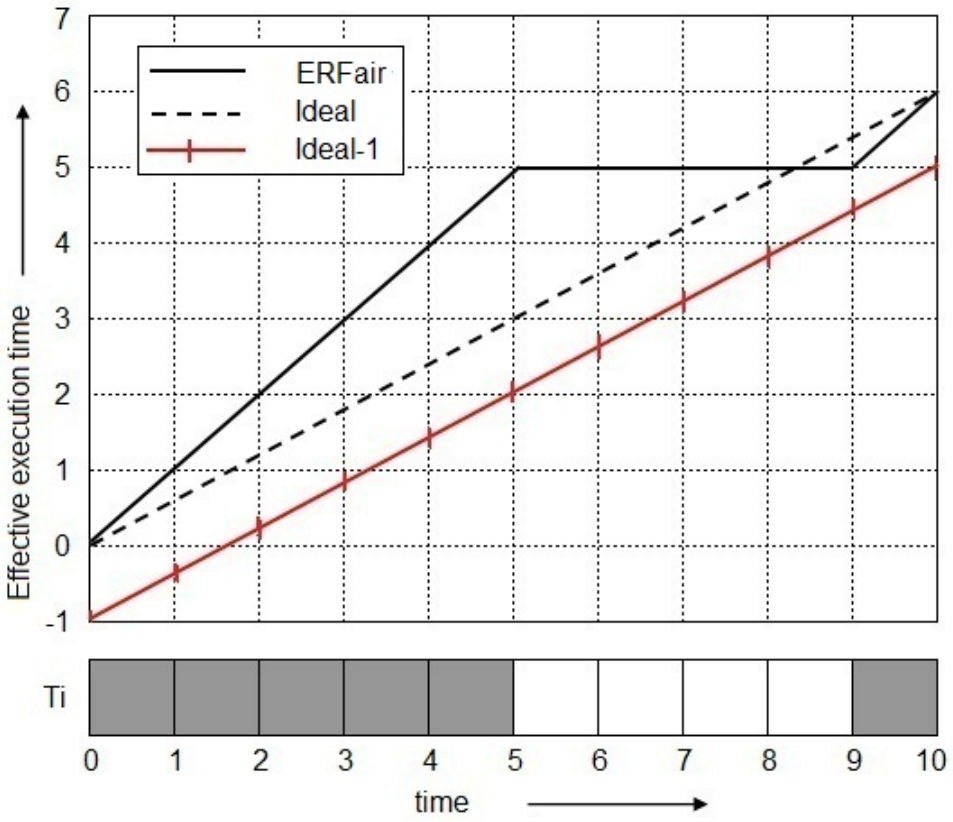


Figure 3.7: Execution of task in Early Release Fair

PF, PD and PD^2 can be easily adapted to allow early releases. *ER-PD* has been proved to be more efficient in case of runtime overhead as that of *PFair-PD* and the job response time is also better. The *ER-PD* is not only correct for the periodic tasks but also for the sporadic and intra-sporadic system [36].

3.2.6.2 PFair Staggered Model

To resolve the problems of bus contention and low cache performance due to scheduling of all the processors at the same time (SFQ model), the staggered model was proposed by Holman and Anderson for implementing *PFair* scheduling on bus based symmetric multiprocessor (SMP) [37, 53].

Opposite to *PFair* aligned model where scheduling of all the processors occurs at the same point (see figure 3.8 (a)), in *PFair* staggered model, scheduling points of processors are distributed across the time (see figure 3.8 (b)). Contrary to *PFair* model, each processor can take its own scheduling decision under staggered model, without the need for explicit synchronization, as shown in figure 3.8 (b). Because processor stalls

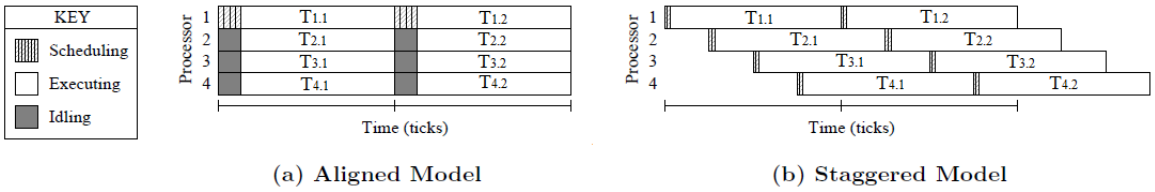


Figure 3.8: PFair Vs Staggered model [37]

are avoided when making scheduling decisions, scheduling overhead is reduced. The experiments show that staggered model significantly reduces the traffic across the buses and hence improves the cache performance [37]. Figure 3.9 shows the load across the processors for staggered model repeating the same experimental setup as discussed for *PFair* in figure 3.6.

3.2.6.3 PFair for intra-sporadic model

Intra-sporadic model was proposed as an extension of the periodic and sporadic model [6, 57]. We know that a sporadic task is just like a periodic task but with a minimum fix inter-arrival time. When a sporadic model is applied to a periodic model the jobs are released late i.e. the separation between two consecutive job releases of any task is more than a task's period.

The intra-sporadic model applies the very same concept on sub-tasks by allowing the sub-tasks to be released late i.e. the intra-sporadic separation between release time of two sub-task $T_{i.j}$ and $T_{i.j+1}$ is allowed to be more than $(\left\lfloor \frac{j}{T_i.u} \right\rfloor - \left\lfloor \frac{j-1}{T_i.u} \right\rfloor)$. This distance can be considered as the period of the sub-task $T_{i.j}$, if it was a periodic task (or the distance between releases of two consecutive sub-tasks).

Each sub-task has an offset that gives the value of right shift. The pseudo-release time and pseudo-deadline are achieved by addition of offset. If $\triangle_{i.k}$ is the offset for sub-task $T_{i.k}$, then the pseudo-release pseudo-deadline time can be calculated as :

$$r(T_{i.k}) = \triangle_{i.k} + \left\lfloor \frac{k-1}{T_i.u} \right\rfloor \tag{3.12}$$

$$d(T_{i.k}) = \triangle_{i.k} + \left\lceil \frac{k}{T_i.u} \right\rceil \tag{3.13}$$

$[r(T_{i.k}), d(T_{i.k}))$ is *PFair* window for IS model.

The offsets are constrained that keep the separation between a pair of sub-tasks to a minimum level of difference of their release time, if the two were periodic. The offset

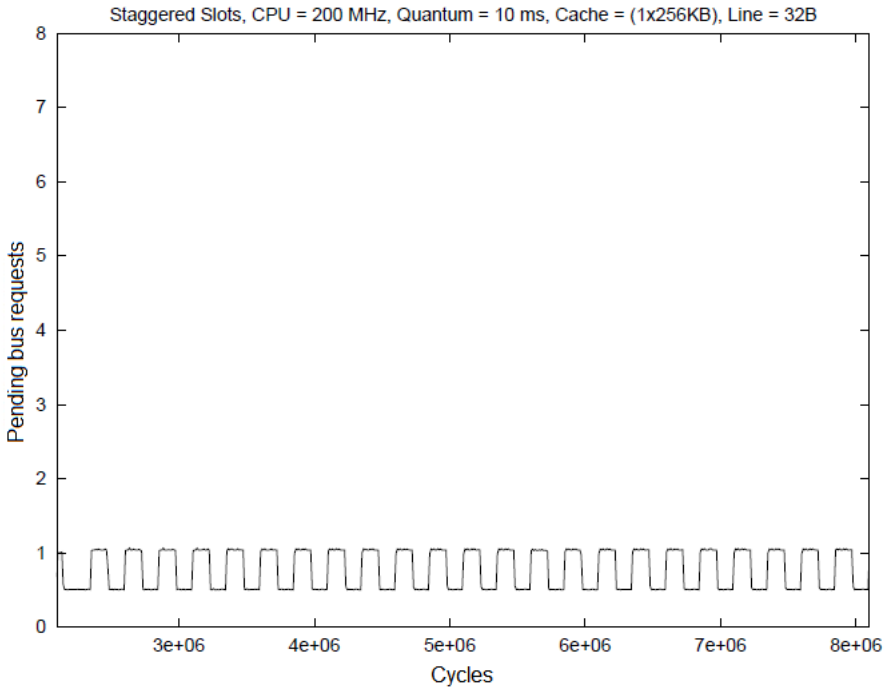


Figure 3.9: Bus load across scheduling points in 8 processors system in Staggered model [37]

must satisfy :

$$k \geq j \Rightarrow \triangle_{i.k} \geq \triangle_{i.j} \tag{3.14}$$

Under IS model each sub-task is allowed to execute before the start of *PFair* windows, similar to that of *ER-Fair*. However, for execution before the pseudo-release time, a sub-task must satisfy an eligibility criteria similar to that of *ER-Fair*.

IS model allows to differ the instantaneous rate of sub-task release from the average rate specified by the task utilization factor. IS taskset is feasible on M processor if and only if :

$$U \leq M \tag{3.15}$$

In [57], Srinivason and Anderson showed that *PFair* PD^2 is optimal for scheduling an IS taskset.

3.2.6.4 Supertasking

We know that *PFair* scheduling requires each task to execute at a constant rate. Hence scheduling decisions are made such that each task receives approximately its

designated share of processor time. But this is done at the cost of frequent switching which is not good for cache performance. In [49], Moir and Ramamurthy observed this problem that certain tasks need to be executed on specific processors. They noted that overhead can be reduced by statically binding the tasks to processors. To support these non-migratory tasks they proposed the concept of supertasking. In supertasking, these non-migratory tasks constitute a group that is scheduled instead of a single task. Each member of supertask is called component task. The total utilization factor of each super task is equal to the sum of utilization factor of its component tasks. Supertasking effectively relaxes the strictness of *PFair* scheduling i.e. each super task is required to make progress at a steady rate rather than individual tasks [38]. Whenever a super task is scheduled one of its component tasks is executed according to an internal algorithm.

3.3 Deadline Partitioned Fair scheduling

3.3.1 Principle

In the recent years, a scheduling model was proposed which combines the notion of fluid scheduling with the one of deadline partitioning to still guarantee the optimality while improving performance of *PFair*. In [43] such approaches are qualified as *Deadline Partitioned Fair* or simply *DP-Fair* scheduling. In this technique, the scheduling intervals are defined by *task deadlines* and scheduling of each interval is done by following the principle of *fairness*. *DP-Fair* techniques lower the number of scheduling points as compare to the *PFair* technique. For a set of 100 tasks, Zhu showed that $BFair$ (a *DP-Fair* technique) has 48 % scheduling points as compare to PD (*PFair* technique) [61]. A relatively relaxed requirement of fairness and a possibility of using varied techniques in scheduling makes it more attractive than *PFair*. In [43], Levin et al. showed that any *DP-Fair* scheduling algorithm for periodic task sets with implicit deadlines is optimal. *LRE-TL* [30], a *DP-Fair* algorithm, also accommodates the sporadic tasks in addition to the periodic tasks.

3.3.2 Division of time in intervals

We know that algorithms like LLF and EDF fail in multiprocessor scheduling because usually they lack some global knowledge about other tasks in the system where each task is struggling for achieving its own deadline. This knowledge can be improved by dividing the time into smaller intervals where all the tasks have a common deadline.

DP-Fair scheduling chooses to divide time into slices where all the tasks have the same deadline which may be different from their original deadlines. The extreme points of these slices are the deadlines (same as arrivals due to the implicit deadline assumption) of tasks and are called *boundaries.* These boundaries are abbreviated as b_0, b_1, etc.

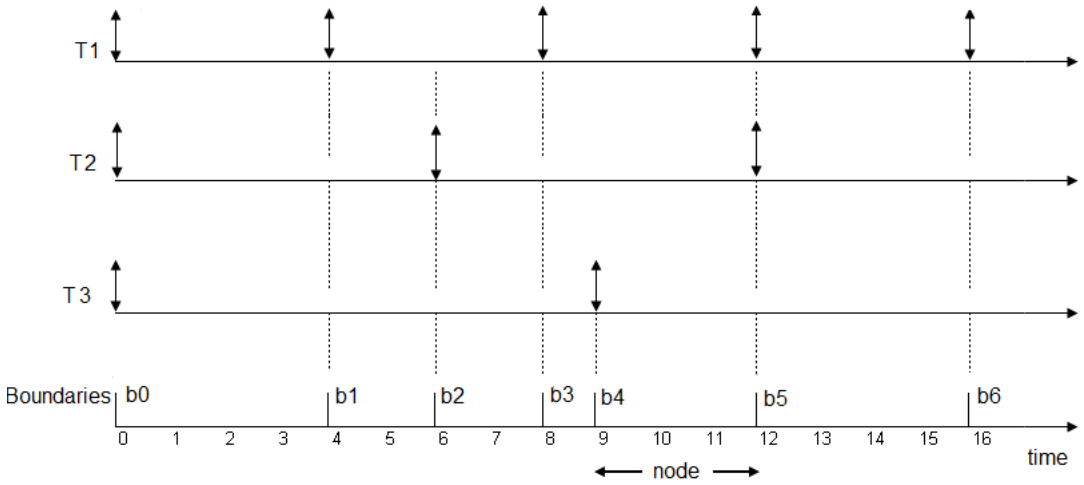


Figure 3.10: Defining boundary in DP-Fair

The boundaries b_0, b_1,... are defined in the figure 3.10 for a system of three tasks T_1, T_2 and T_3 define the boundaries. One of the difference of *DP-Fair* from the *PFair* is that the length of intervals in *DP-Fair* is not constant. It varies and the variation depends on the deadlines of the tasks in the taskset.

The interval between any two consecutive boundaries is also known as a $node$. Thus the problem is to define scheduling rules inside one interval so that the resulting global schedule is valid.

3.3.3 Scheduling within an interval

The scheduling inside each node is based on the fluid scheduling model, as the algorithm tracks task fluid path through task execution. Within each node, all jobs are allocated some execution time units for the node and these portions of jobs share the same deadline.

There are two aspects that need to be considered for scheduling inside a node :

- Nodal execution times computation
- Taskset dispatching concerning the time and processors

We will use an abstraction which is named Time and Nodal remaining execution time plane *(TN-plane)*. A *TN-plane* is a visual abstraction to represent execution of tasks on a multiprocessor system which allows insightful and analytical understanding. It was introduced by Funaoka et al. [28].

The figure 3.11 shows the basic idea behind a *TN-plane*. In this figure, time is represented on horizontal axis and nodal remaining execution time on vertical axis. The

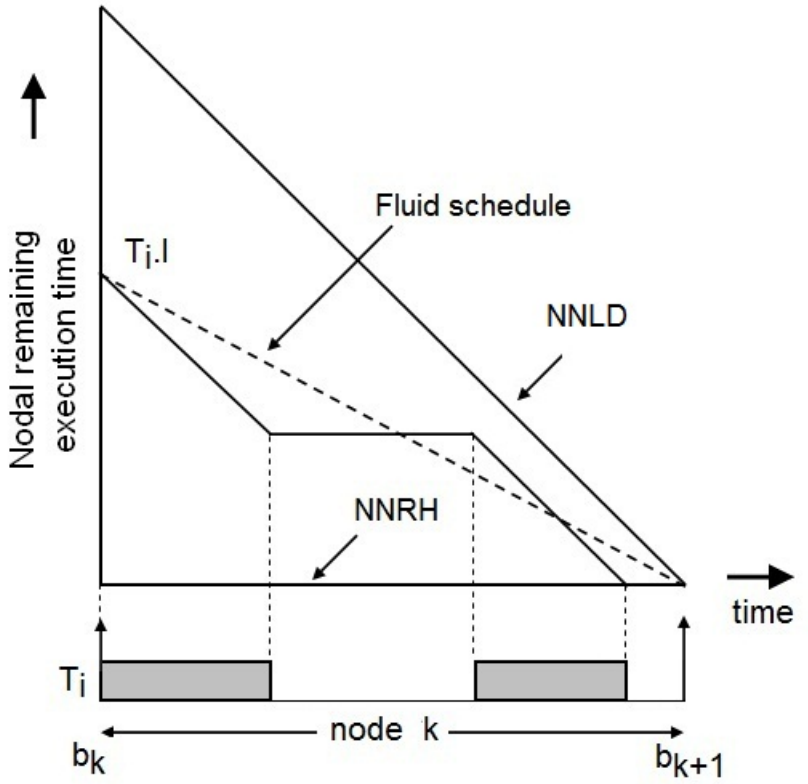


Figure 3.11: A TN-plane

nodal execution time is the minimal duration for which the task needs to be executed during the node. The nodal execution time of a task T_i is denoted by $T_i.l$ and is different from actual remaining execution time of the task. The nodal execution time $T_i.l$ allocated to task T_i at b_k must be finished before end of the node i.e. b_{k+1}. The end of the node is called *nodal deadline*. The zero nodal remaining execution time axis is also called *No Nodal Remaining execution time Horizon (NNRH)*. The oblique side of the *TN-plane* is called No Nodal Laxity Diagonal $(NNLD)$. A task on this diagonal must be executed immediately to achieve its nodal deadline. It is further explained later on. The task is said to be *nodally feasible* if it finishes its nodal execution time before the end of the node.

When N tasks are considered in the system, their fluid schedules are made as shown in the figure 3.12. A right angled isosceles triangle can then be considered between any two consecutive boundaries for each task. The right most vertex of the triangle coincides with the fluid schedule. Since triangles for all the tasks are of the same size in a node, task execution domains for all the tasks may be represented as a single overlapped isosceles triangle that constitutes a *TN-plane*. The size of one *TN-plane* may be differ-

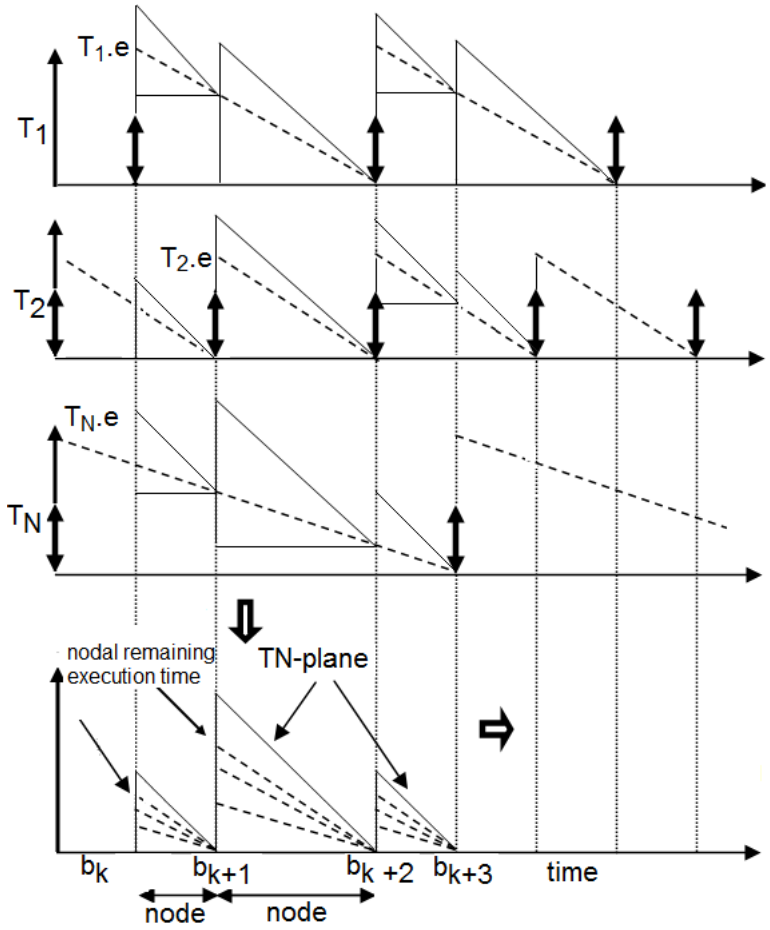


Figure 3.12: Consecutive TN-planes.

ent to the size of other *TN-plane* as shown in the figure 3.12.

A single *TN-plane* between boundaries b_k and b_{k+1} is shown in figure 3.13. The execution of tasks are represented by the movement of tokens over time. We define the *nodal laxity* of a task that represents the time which a task can wait without missing the nodal deadline. For a task T_i, at any time t, it can be defined as :

$$T_i.laxity(t) = b_{k+1} - t - T_i.l(t) \tag{3.16}$$

where $T_i.l(t)$ is the nodal remaining execution time at t.

At b_k, the value of nodal laxity is maximum i.e. $b_{k+1} - b_k - T_i.l$. This is shown in the figure 3.13. When a token hits the *NNLD* side, it implies that the task does not have any nodal laxity. The task with zero nodal laxity must be executed immediately

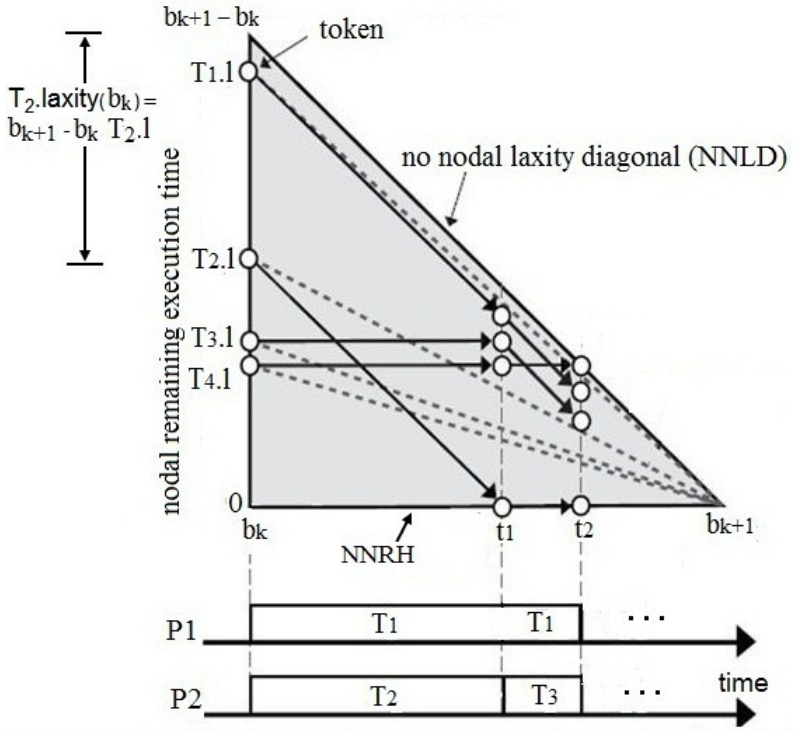


Figure 3.13: Scheduling in a TN plane

otherwise it cannot satisfy the scheduling objective of nodal feasibility.

The dispatching process of the taskset can be considered as controlling the movements of tokens which are shown by trajectories. Recall that the slope of a trajectory is 0 when the task represented by token is waiting and -1 when it is running. In this process, we try to find a trajectory so that all the tasks finish their nodal execution times before the nodal deadline.

The *TN-plane* makes it possible to envision the entire scheduling activity over time as scheduling in repeated *TN-planes* of various sizes, so that feasibly scheduling on a single *TN-plane* results in a feasible schedule for all *TN-planes* across time [12, 36, 43].

3.3.3.1 Nodal execution time computation

To achieve a nodally feasible schedule, the following conditions must be satisfied for the computation of nodal execution times in a node :

1. Nodal execution time of each task is less than or equal to the length of the node,

i.e. $T_i.l \leq (b_{k+1} - b_k) \; \forall \, i \; 1 \leq i \leq N$;

2. The sum of the nodal execution times of all the tasks is less than or equal to total capacity of the processors for the node, i.e. $\sum_{i=1}^{N}(T_i.l) \leq M * (b_{k+1} - b_k)$;

3. For any task T_i, the sum of its all nodal execution times $T_i.l$ allocated during its time period is equal to its worst case execution time. Suppose $T_i.j$ is the j^{th} job of the task T_i, b_k is its activation time and b_q is its deadline, then the following condition must be satisfied:

$$\sum_{x=k}^{q-1}(T_i.l^x) = T_i.e \;\; \forall \, i \tag{3.17}$$

where $T_i.l^x$ is the nodal execution time computed at b_x. If a taskset is nodally feasible and it also satisfies the condition in equation 3.17, then the deadline constraints of the taskset will be met.

In [48], Megel et al. used three rules to compute the nodal execution time over hyper period using a linear programming approach. Without considering fairness, they used the schedulability conditions (equation 2.7, 2.8) along with equation 3.17 to set a utilization factors such that each task is feasible. They used proposed an *Incremental scheduling with Zero Laxity (IZL)* algorithm for dynamically scheduling of jobs with identical start times and deadlines i.e. in an interval. They use a linear programming formulation and a local scheduler which exhibits low complexity and produces few task preemptions and migrations.

In most of the studied algorithms, the nodal execution time of each task is computed using the principle of fairness. The nodal execution time is computed such that it remains close to the fluid execution value. In case of non work-conserving scheduling is considered, following two possibilities exist :

1. The nodal execution time for each task can be computed exactly proportional to the task utilization factor. Hence a task T_i with utilization factor $T_i.u$ gets $T_i.u(b_{k+1} - b_k)$ units for its $T_i.l$. The resulting value may be a real number which causes a practical problem (due to the hardware characteristics of processors, execution time unit numbers should be integral multiples of the highest precision timer).

2. The second option counters the problem of real number and computes a nodal execution time which is always an integer. For a task T_i with utilization factor $T_i.u$, nodal execution time $T_i.l$ between b_k and b_{k+1} is either $\lfloor (b_{k+1} - b_k) * T_i.u \rfloor$ or $\lceil (b_{k+1} - b_k) * T_i.u \rceil$.

$Task$	$T_i.l$
T_1	5
T_2	4
T_3	7
T_4	3

Table 3.2: Taskset configuration with $M = 2$ and $(b_{k+1} - b_k) = 10$ for the figure 3.14

3.3.3.2 Taskset dispatching

The taskset dispatching is the process to find the path for the tokens in a *TN-plane* such that all the tokens successfully reach at *NNRH* without crossing *NNLD*. We consider the dynamic dispatching where the dispatching decisions are taken at run time i.e. at the beginning of the node which is called *primary scheduling* point as well as at some *secondary scheduling* points within a node. The other type of dispatching technique is static where all the decisions are precomputed and are just executed at the runtime. The following general rules are applied while dispatching taskset between two boundaries b_k and b_{k+1}:

1. At most M tasks can be executed at any given time ;
2. A task with zero nodal laxity is given maximum priority. It must be executed immediately, otherwise it will miss its nodal deadline. If all the processors are busy, the task will preempt a running task with non-zero nodal laxity. This situation is shortly termed as $event\ C$ shown in figure 3.14;
3. When a task has executed its nodal remaining execution time, we need to reallocate the corresponding processor. Therefore, the task needs to be stopped. This is shortly termed as $event\ B$ as shown in figure 3.14;

   The points where event C and event B occur are called secondary scheduling points.

In the process of taskset dispatching, no processor remains idle if there is a ready task with non-zero nodal remaining execution time. There may exist various trajectories in the plane which can be adopted for feasible schedule in taskset dispatching. The best is the one which causes lowest overhead in terms of scheduling decisions, preemptions and migrations.

Figure 3.14 shows a dispatching of a taskset on two processors inside a *TN-plane* of length 10. The configuration is given in the table 3.2. It is easy to check that conditions of nodal feasibility are met. The task execution is represented by the movement of tokens. It shows that tasks T_1 and T_2 start their execution at the beginning of the node.

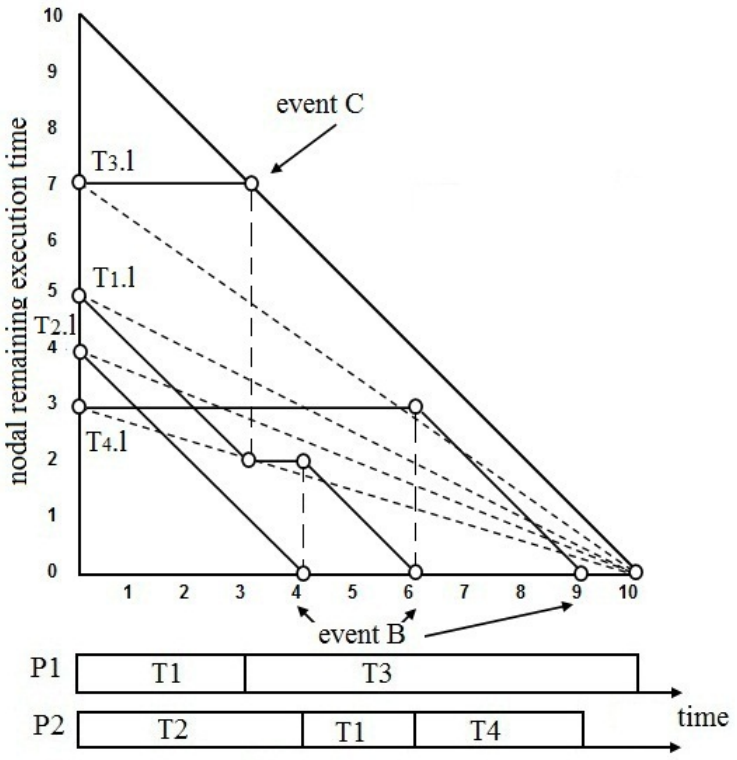


Figure 3.14: A schedule in a TN-Plane

At time 3, event C occurs because T_3 reaches $NNLD$. The task T_1 is preempted and is replaced by T_3. Event B occur when a token reaches *NNRH* i.e. at time 4 for T_2 and 9 for T_4.

The above mentioned scheduling conditions are concerned with a non-work conserving scheduling. The dispatching inside a work-conserving scheduling model are discussed later on in section 3.3.6.

3.3.4 Classification of DP-Fair algorithms

DP-Fair algorithms can be classified according to the way they behave. The classification is based on whether:

1. The schedule is non-work conserving or work conserving;
2. The nodal execution time is a real number or an integer;
3. The dispatching technique is dynamic or static.

The classification graph of *DP-Fair* is given in figure 3.15.

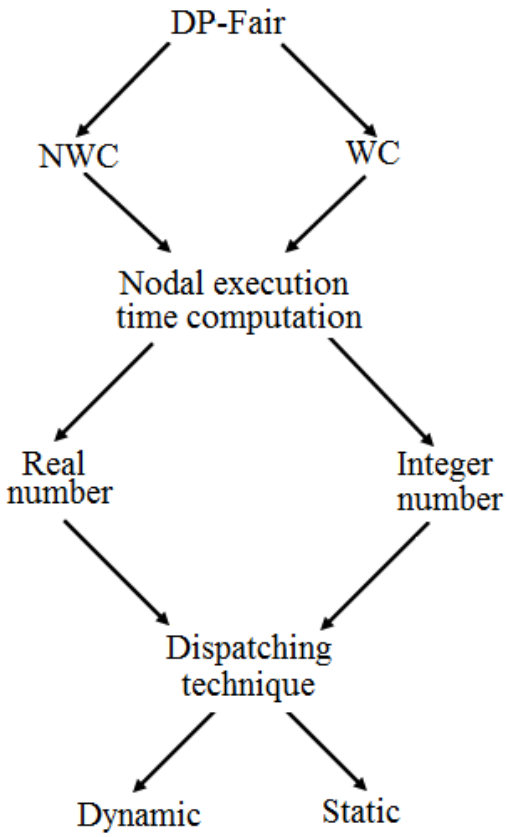


Figure 3.15: DP-Fair classification graph

The *DP-Fair* algorithms presented in the literature belong to one of these categories. This classification helps to divide a *DP-Fair* algorithm into sub-problems and allows to concentrate on these sub-problems independently. Different combinations of solution to these sub-problems can result in the finding of some hybrid algorithms with improved characteristics as compare to original ones. Table 3.3 gives the names of some known algorithms with their position according to this classification.

In the following sections these *DP-Fair* algorithms are discussed.

3.3.5 Non-work conserving algorithms

3.3.5.1 Real number nodal execution time computation & dynamic dispatching

In this category the computed nodal execution time is a real number while the tasks are dynamically dispatched to the processors. *LLREF [36]* and *LRE-TL [30]* are examples of this type of *DP-Fair* scheduling.

Name of algorithm	WC/ NWC	$T_i.l$ (Real / integer)	Dispatching type
BFair [Zhu et al. 2003]	NWC	Integer	Static
LLREF [Ravindran et al. 2006]	NWC	Real	Dynamic
LRE-TL [Funk et al.2009]	NWC	Real	Dynamic
DP-Wrap [G. Levin et al. 2010]	NWC	Real	Static
BFair/LRE-TL[Cho et al. 2010]	NWC	Integer	Dynamic
E-TNPA [Funakao et al.1 2008]	WC	Real	Dynamic
TRPA [Funakao et al.2 2008]	WC	Real	Dynamic

Table 3.3: Classification of DP-Fair algorithms

LLREF

LLREF stands for *Largest Local Remaining Execution time First* [36]. The term ''local'' is equivalent to the term ''nodal'' .

The nodal execution time for each task is exactly proportional to the task utilization factor.

$$T_i.l = T_i.u * (b_{k+1} - b_k) \ \forall i,\ k \tag{3.18}$$

At the primary scheduling point, all the ready tasks are sorted in order of largest nodal remaining execution time first and then at most first M tasks are dispatched to the processors.

At the secondary scheduling points in *LLREF*, all the running tasks are preempted and all the ready tasks sorted in the order of largest nodal remaining execution time first, then the first at most M tasks are executed on M available processors. In worst case a task may change its processor after every scheduling point.

Srinivason et al. [56] showed that the number of task preemptions can be bounded in *LLREF* by consecutively scheduling a task on the same processor. This reduces the number of context switches and possibility of cache misses.

LRE-TL

LRE-TL stands for *Local Remaining Execution TL-plane.* It was proposed by Funk et al. [30, 31] and is optimal for periodic as well as sporadic task sets. It suggests a modification in *LLREF* [36] to lower the scheduling overhead.

The computation of execution time for each task is same as that of *LLREF*.

The dispatching technique in *LRE-TL* is different from that of *LLREF*. In *LRE-TL* the scheduling process at primary scheduling point is exactly same as that of *LLREF* but it differs at secondary scheduling points. At secondary scheduling point, the task causing the scheduling is replaced while the other tasks continue to execute without any preemption. For example, when an event B occurs, all the tasks will continue except

	LLREF	LRE-TL
Running time		
TN-plane initialization	$O\,(N)$	$O\,(N)$
B & C event (per TN-plane)	$O\,(N^2)$	$O\,(N\,\log(N))$
Other overhead		
Max. preemptions	$O\,(M)$	$O\,(1)$
Max. migrations	$O\,(MN)$	$O\,(M)$

Table 3.4: Computational complexity of LLREF vs. LRE-TL [30]

the tasks causing the event B. It will be replaced by another task from the ready list. Hence event B does not cause any unnecessary preemption. On the other hand, event C causes a preemption. The task with zero laxity preempts a running task with non-zero nodal laxity and occupies the processor without any effect on the other tasks.

Table 3.4 gives a comparison between *LLREF* and *LRE-TL* in terms of computational complexity. It shows that *LRE-TL* shows a better performance than *LLREF*. In *LRE-TL,* sporadic tasks can also be taken into account in some cases.

3.3.5.2 Real number nodal execution time computation & static dispatching

In this category, the nodal remaining execution time is exactly proportional to the utilization factor while the dispatching is done using offline static technique.

DP-Wrap

DP-Wrap [43] is the simplest *DP-Fair* scheduler but it was proposed after other ones.

To schedule the taskset inside a node, *DP-Wrap* makes a block for each task with length equal to its utilization factor. These blocks are lined along with the number line in any order, starting at zero as shown in 3.16 (a). The maximum length of this line is M. The time line is split into lengths of 1 chunks at $1, 2, ..., M-1$. Each of the chunk is assigned to a processor and it represents the scheduling of tasks on that processor;. The tasks which are split will execute on two processors as shown on 3.16 (b). Task T_3 is split between processor P_1 and P_2.

Once the configuration of a chunk is defined, it always remains the same. The nodal execution time for any task is computed just by multiplying its portion in a chunk with the node length. For example, for a node length of 4, nodal execution time of T_1 on processor P_1 will be 0.25*4 = 1 unit. If the node length becomes 12, the nodal execution time will become 0.25*12 = 3.

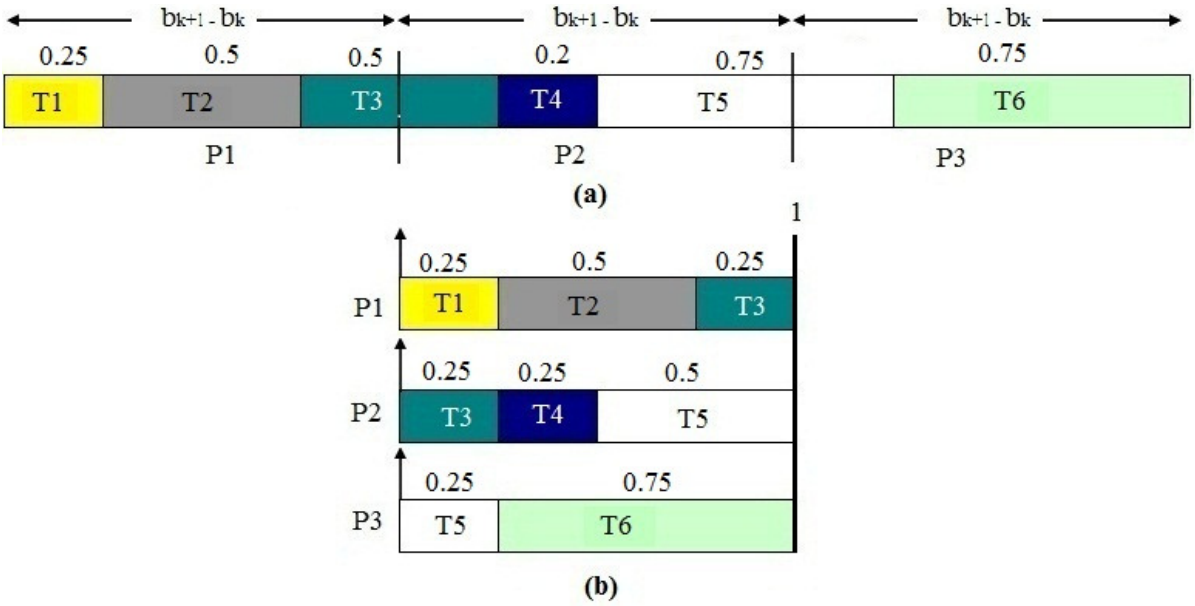


Figure 3.16: DP-Wrap algorithm

In *DP-Wrap* scheduling , if we continue with predetermined ordering for tasks in each chunk then the $M - 1$ tasks can be split and each of these may migrate twice per node: once in the middle, and again at the end, when it moves back to its starting processor. The number of migrations can be decreased simply by reversing the ordering of tasks on each processor in the next node. This is shown in the figure 3.17. It shows the execution of tasks on two consecutive nodes. The 3.17 (a) shows the simple *DP-Wrap* scheduling while 3.17 (b) shows the scheduling using *DP-Wrap* with above discussed migration controlling technique. In two nodes, task T_3 migrates three times between processors indicated by arrows 3.17 (a). Now if we reverse the tasks order in the next node, the migration of taskT_3 will decrease as shown in 3.17(b). This strategy is also known as mirroring.

The scheduling complexity of *DP-Wrap* per *TN-plane* is $O(N)$.

3.3.5.3 Integer number nodal execution time computation & static dispatching

Although *DP-Fair* is theoretically optimal with the real value of nodal execution time, a practical problem may face during implementation (execution times should be an integral multiple of the highest timer precision). This is because the real world scheduler always take their decisions on integer value. Therefore, to practically achieve the optimality nodal execution time must be an integer. This problem is solved in *BFair*.

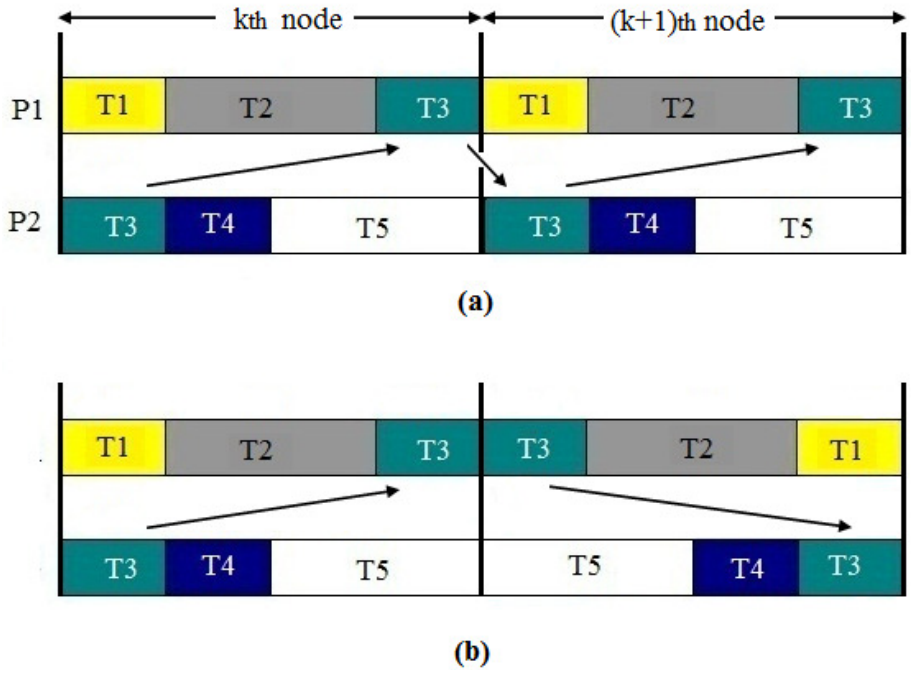


Figure 3.17: DP-Wrap (a) With out mirroring (b) With mirroring

BFair [61]

Boundary fair or *BFair* is a technique proposed by Zhu et al. [61] based on an integer time model. *BFair* states that optimality can be achieved even if the nodal execution time is not an exact multiple of task utilization factor and the node length. In *BFair*, for a task T_i with utilization factor $T_i.u$, nodal execution time for any node between b_k and b_{k+1} is either $\lfloor (b_{k+1} - b_k) * T_i.u \rfloor$ or $\lceil (b_{k+1} - b_k) * T_i.u \rceil$. This condition can also be expressed in terms of lag (defined in equation 3.1). For a *BFair* schedule, at any boundary b_k, *lag* must obey the following condition :

$$-1 < lag(T_i, b_k) < 1 \; \forall i, \forall k \tag{3.19}$$

BFair computes the nodal execution time by dividing the process in two following steps:

- Every task is allocated some mandatory units, which a task must execute so that the difference between the fluid schedule and practical schedule is less than 1. $T_i.m^{k+1}$ is the number of mandatory units of task T_i calculated at b_k for the interval $[b_k \; b_{k+1})$. The mandatory units of a task is the lower round off value of sum of its real fluid value of present node and the remaining work of previous node.

Remaining work is the difference of fluid share and computed nodal execution time of a node. After the computation of mandatory units, the condition of lag can be expressed as $0 < lag(T_i, b_{k+1}) < 1$.

- The unused remaining units in the node after the allocation of the mandatory units to the tasks in taskset are $RU = M * (b_{k+1} - b_k) - \sum_{i=1}^{N}(T_i.m^{k+1})$. These remaining units are distributed among the highest priority tasks such that no task gets more than one unit. The tasks with the higher values of remainig work are given priority. Some more rules are applied in case of a tie. This single unit is called optional unit.

Hence, the nodal execution time of a task T_i is the sum of mandatory units and optional unit.

$BFair$ uses McNaughton's algorithm [47] for dispatching the tasks in a static way. It is a simple bin packing problem the possibility of dividing the task between two processors. It is a function that calculates a schedule sequence of fixed number of jobs on any number of machines. In other words, McNaughton's algorithm solves the problem (number Of Processors, duration Of Jobs). It is used to schedule independent jobs on identical processors in order to minimize schedule length. Schedule length S_l is computed by using the following equation:

$$S_l = max\{max(l_i), \frac{\sum_{i=0}^{k-1}(l_i)}{M}\} \tag{3.20}$$

Where k is the number of ready tasks in the list.

Suppose we have a taskset with execution times {5, 8, 4, 6, 2, 5} and we have a system with three processors with node length 12. The node length is calculated as follow:

$$S_l = max\{(8), \tfrac{30}{3}\} = max\{8, 10\} = 10.$$

The output schedule using McNaughton's algorithm is given in figure 3.18. The scheduling complexity of *BFair* per node is $O(N)$.

3.3.5.4 Integer number nodal execution time computation and dynamic dispatching technique

Nodal remaining execution time computed is an integer value and then a dynamic dispatching technique is used thereafter.

In [21], Cho et al. used the *BFair* for computation of nodal execution time and used a dynamic dispatching technique of *LRE-TL*. Though McNaughton's technique gives the minimal schedule length, but at the same time it may cause unnecessary migration

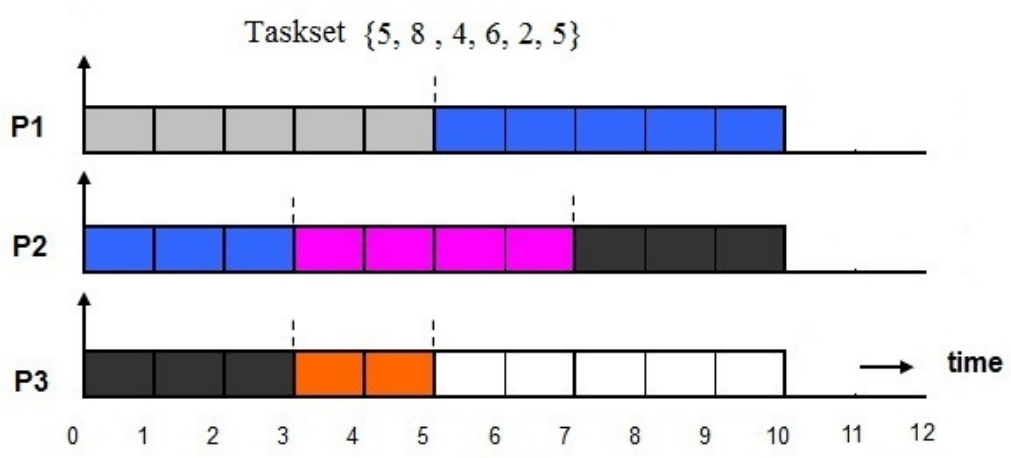


Figure 3.18: Result of McNaughton's algorithm for the taskset

for some cases. Different to McNaughton's technique, *LRE-TL* is a dynamic dispatching technique with the possibility of using some additional heuristics to decrease the migration and preemptions.

3.3.6 Work conserving algorithms

As discussed earlier, a scheduling algorithm is said to be *work conserving* if and only if it never leaves any processor idle when there exists at least one ready task.

There is no difference between non-work conserving and work conserving scheduling when the total utilization factor of the taskset is equal to the capacity of processors i.e. $U = M$. However when $U < M$, a difference comes between the two scheduling models. So we discuss the work conserving algorithms with an assumption that $U < M$. Similar to that of non-work conserving *DP-Fair* techniques, the work conserving ones also perform the scheduling process into two steps:

1. The computation of nodal execution time;
2. The dispatching of taskset to processors.

There is relatively less work available on work conserving algorithms than non-work conserving ones. The two known work conserving algorithms are based on real nodal execution time computation. They are explained in the following.

3.3.6.1 NVNLF based on Extended TN-plane [29]

NVNLF stands for *''No Virtual Nodal Laxity Diagonal First''*. We explain the algorithm into two parts:

Algorithm 1 NVNLF based on ETNPA [29]

– **Input** : ζ^k -The queue of ready tasks. Size of $\zeta^k \leq N$,
– **Output** : $T_i.l$ for all the tasks to be dispatched on processors,
– **Notation**
  – $T_i.R$ is total remaining execution time of task at b_k, to be updated before the execution of algorithm
  – $T_i.l^*$ is the initially computed nodal execution time
  – $T_i.a$ is additional nodal time for task T_i
  – L' is the sum of free time units
  @ (b_k)

1. $L' = (M - U) * (b_{k+1} - b_k)$
2. **for** $(T_i \epsilon \zeta^k)$ // Retrieve extra units from tasks
3. $T_i.l^*$ = $T_i.u * (b_{k+1} - b_k)$ // Basic nodal execution time
4. **if** $(T_i.R \leq T_i.l^*)$
5. // retrieve additional nodal time $(T_i.a \leq 0)$
6. $T_i.a = T_i.R - T_i.l^*$
7. $T_i.l = T_i.l^* + T_i.a$
8. $L' = L' - T_i.a$
9. **else**
10. $T_i.a$= 0
11. **end if /else**
12. **end for**
13. **for**$(T_i \epsilon \zeta^k)$ // Distributes extra units from tasks
14. **if** $(T_i.a == 0)$
15. // apportion additional nodal time $(T_i.a \geq 0)$
16. **if** $(T_i.R \leq b_{k+1}$-$b_k)$
17. $T_i.a = min\{T_i.R - T_i.l^*, L'\}$
18. **else**
19. $T_i.a = min\{(b_{k+1}$-$b_k) - T_i.l^*, L'\}$
20. **end if /else**
21. $T_i.l = T_i.l^* + T_i.a$
22. $L' = L' - T.a$
23. **end if**
24. **end for**

Computation of nodal execution time: In this technique, in addition to the ''conventional'' nodal execution time computation, the time units left unused in the node are distributed among the tasks at the start of each node. These units are distributed such that the resulting nodal execution time of any task is not more than the length of the node or its total remaining execution time. If $T_i.l^*$ is the initial value of computed nodal execution time and $T_i.a$ is the additional time units allocated to it, then the final value of computed nodal execution time in a node between any two boundaries b_k and b_{k+1} is calculated as:

$$T_i.l = T_i.l^* + T_i.a \tag{3.21}$$

The minimum unused processor time L' between any two boundaries b_k and b_{k+1} is calculated as:

$$L' = (M - U) * (b_{k+1} - b_k) \tag{3.22}$$

Recall that M is the number of processors and U is the total utilization factor of all the tasks in the task set.

The distribution of these units to the tasks is given in the algorithm 1.

The first for loop in the algorithm checks the case where a task has previously executed more units than its fluid share. In this case the task does not need all the nodal execution time units initially computed. The difference between fluid share and already executed units is added in the free unused units.

The second for loop arbitrarily distributes the free units among the ready tasks. The unused units are allocated such that maximum value of final nodal execution time of a task is either equal to the length of node or the total remaining execution time of the task.

Extended Time and Nodal execution time Plane Abstraction or *ETNPA* is an abstraction to visualize the *NVNLF*.

The work conserving scheduling needs few modifications in the original *TN-plane* to explain it. *ETN-plane* is shown in figure 3.19 [29]. Since there are some extra time units $T_i.a$ are allocated to a task other than the initially calculated nodal execution units $T_i.l^*$, few changes are done in *TN-plane* and this results in *ETN-plane*.

The coordinates of the original *TN-plane* (shown by lined triangles) are $(b_k,\ T_i.a)$, $(b_{k+1},\ T_i.a)$ and $(b_k,\ b_{k+1} - b_k + T_i.a)$ where the $T_i.a$ has positive value for 3.19 (a) and has negative value for 3.19 (b). A line called No Virtual Nodal Laxity Diagonal $(NVNLD)$ is drawn from $(b_k, b_{k+1} - b_k)$ to $(b_{k+1}, 0)$ which refines the diagonal of zero nodal laxity. The line which represents that tasks have no nodal remaining execution time is called No Nodal Remaining Horizon $(NNRH)$. The $NNRH$ is same as that in *TN-plane* and is reference action of $T_i.l$. The shaded triangles in the figure 3.19 represents the *ETN-plane*.

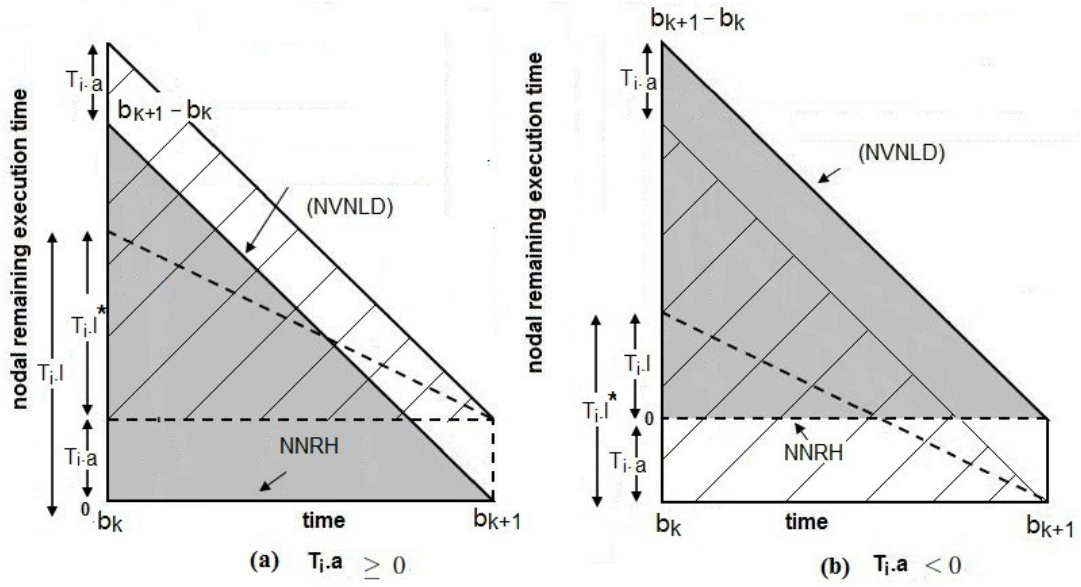


Figure 3.19: ETN-plane for a single task

T_i	$T_i.u$	$T_i.e$	$T_i.l^*$	$T_i.a$	$T_i.l$
T_1	$\frac{2}{5}$	2	2	0	2
T_2	$\frac{4}{9}$	4	2.22	1.78	4
T_3	$\frac{3}{12}$	3	1.25	1.75	3

Table 3.5: Computation of $T_i.l$ in a node length of 5

The computation of $T_i.l$ is elaborated with the help of an example. The summary of computation of $T_i.l$ is given in table 3.5 Suppose we have a taskset of three tasks with $T_1.u = 2/5, T_2.u = 4/9$ and $T_3.u = 3/12$. The $T_i.e$ value for tasks are 2, 4 and 3 respectively while there are two processors i.e. $M = 2$. The total utilization factor of taskset is $U = 1.09$. For the nodal length 5, $T_1.l^* = 2$, $T_2.l^* = 2.22$ and $T_3.l^* = 1.25$. Hence the idle time for processors five units is L'= 4.53. Using the *NVNLF* algorithm we can use this time for the tasks which have still some units to execute and normally which wait until next *node*. The first task needs no additional time. Second and third task will get $T_2.a$= 1.78 and $T_3.a$= 1.75. Nodal execution time for T_2 and T_3 becomes 4 and 3 respectively.

Scheduling rules: Once the nodal execution times of the task are finalized, the tasks are dynamically dispatched to the processors as that in a non-work conserving techniques. The main scheduling rules of *NVNLF* inside an *ETN-plane* are same as that of scheduling rules in a *TN-Plane.* The 2 main ones are:

- When the token of a task reaches the *NVNLD*, it preempts a running task with non-zero virtual nodal laxity and starts execution. This is called event *C*;
- When the token of a task reaches the *NNRH*, it is preempted because it has finished its nodal execution time allocated to it for the current node. This is also called event *B*;

3.3.6.2 NNLF based on TR-plane

NNLF [42] stands for *''No Nodal Laxity First''*.

The nodal execution time is statically computed at the start of a *node* and then a dynamic dispatching technique is adopted to realize the work conserving behavior. The running tasks continue to execute beyond their nodal execution time while they do not compromise the nodal feasibility of the taskset.

Computation of nodal execution time: Each task is allocated a nodal execution time at the start of each node. This time is allocated such that the practical schedule coincides with the fluid schedule by the end of the node while keeping in mind the previous execution of the task.

To understand the computation of nodal execution time, the status of a task in this algorithm can be considered either as unsafe or safe. The task is said to be *unsafe* if the task has not completed its nodal execution time. The task is said to be *safe* once it completes its nodal execution time. We term the boundary from where the status of task T_i is changed from unsafe to safe task as $T_i.s$. It is the final or the lower most value of fluid schedule in the current node. The value of $T_i.s$ is calculated at the start of each node and remains unchanged by the end of that node. It can be calculated by following equation at the start of node b_k:

$$T_i.s = T_i.e - T_i.u * (b_{k+1} - T_i.r) \tag{3.23}$$

where $T_i.r$ is the release time of current job of task T_i. Three different values of $T_i.s$ for three consecutive nodes are shown in the figure 3.20.

Once the value of $T_i.s$ is calculated at b_k, the nodal execution time of a task T_i is computed by following equation:

$$T_i.l = max\{T_i.R - T_i.s, 0\} \tag{3.24}$$

where $T_i.R$ is the total remaining execution time of the task at that b_k.

Scheduling rules: After the computation of nodal execution time units for all the ready tasks, at most M arbitrary tasks are chosen for starting execution. Each task continues to execute unless:

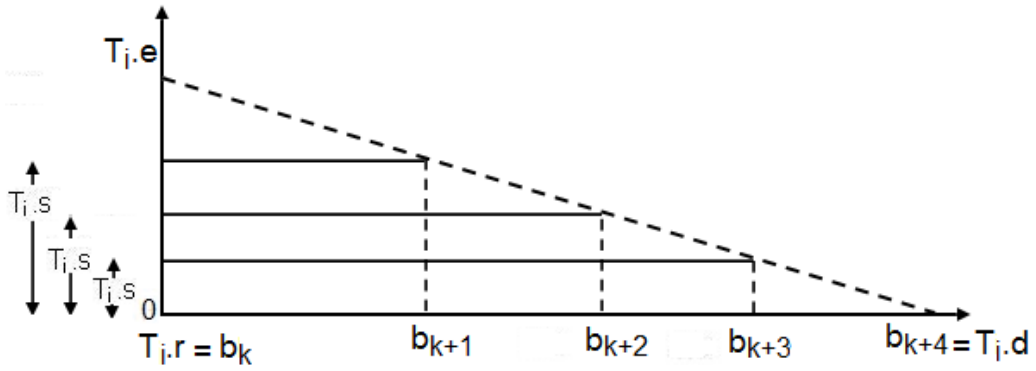


Figure 3.20: Multiple TR-planes of a task

1. The task finishes its total remaining execution time;
2. An event C occurs and the task causing this event replaces it;
3. The sum of nodal remaining execution time of all the unsafe tasks at time t becomes equal to the remaining capacity of the processors at that time. This is called event F.
   Suppose that t_F is the time when event F occurs and U_F is the total nodal remaining utilization factor of all the unsafe ready tasks at this time. Also suppose that M is the remaining capacity of the processors at t_F, then the event F occurs as soon as the following equation becomes true:
   $$U_F \geq M \tag{3.25}$$

A flag F is also attached with event F. The flag is in *''off''* state by default at the start of a node. When event F occurs the flag becomes *''on''* from its default state *''off''*.

Once the event F occurs, all the tasks in safe tasks are stopped and only unsafe tasks are executed.

NNLF scheduling can also be visualized by using a *TR-plane* which is just an abstraction. A *TR-plane* is a trapezium as shown in figure 3.21**.** The basic concept of a *TR-plane* is similar to that of *TN-plane*. The upper right diagonal of both *TR-plane* and *TN-plane* is No Nodal Laxity Diagonal *(NNLD)*. However there are few differences between the two planes:

1. *TR-plane* is a trapezium opposite to that of a *TN-plane* which is a triangle;
2. The vertical axis of a *TR-plane* represents the total remaining execution time opposite to that of a *TN-plane* which represents the nodal remaining execution time;
3. The *TR-planes* are not congruent even in the same node as opposite to a *TN-plane* where all the planes are congruent in the same node.

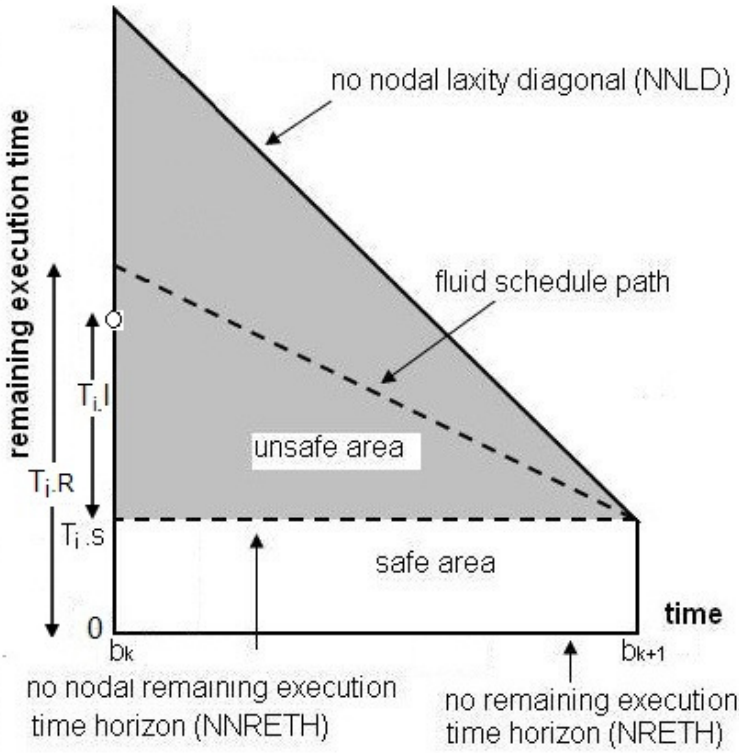


Figure 3.21: A TR plane

	Flag F off	Flag F on
Event B	NRETH	NNRETH
Event C	NNLD	NNLD

Table 3.6: NNLF based on TRPA

The position of the token as shown in the figure 3.21 depends on the execution of task in the previous node.

The conditions of event *B* and event *C* inside a *TR-plane* are given in table 3.6. It shows that when the flag is *off* the conditions of occurrences of event *B* and event *C* remains the same as in *ETN-plane.* When the flag is *on* the conditions of a simple *TN-plane* are applied.

3.4 RUN (Reduction to a UNiprocessor)

RUN is an optimal global scheduling algorithm for tasks with fixed rates [2]. It was proposed by Regnier et al.[55] in 2011. Contrary to already existing optimal algorithms which are based on proportional fairness, it uses a very weak version of proportional fairness. As the name suggests *RUN* implements the scheduling by reducing the problem into a uniprocessor problem. The advantage of using *RUN* is a significant reduction in the number of preemptions per job regardless of number of processors, tasks or jobs [55].

RUN performs the following three steps for implementing scheduling:

1. It reduces the real-time multiprocessor scheduling problem to one or more easily solved uniprocessor problems. This is done in two operations which are named as DUAL and PACK;
2. The uniprocessor problem is solved by well known techniques;
3. The solution is transformed back to original multiprocessor problem.

The whole process is divided into offline reduction and online scheduling.

In offline reduction the original taskset is partitioned into subsets of accumulated utilization not greater than 1. A virtual server is associated to each packed subset and it will be incharge of scheduling the tasks of its subset which are also called client tasks.

In online scheduling the *RUN* algorithm schedules those servers at steady rate proportional to their utilization factor between any two deadlines of their task clients (the utilization factor of a server is precisely equal to the sum of utilization factors of its clients) leading to a form of ‘‘partitioned proportionate fair’’. However the servers are not required to schedule their clients at a steady rate; they simply apply *EDF* rules.

3.4.1 Offline scheduling

The following operations are important to understand the offline reduction process:

DUAL operation: The DUAL operation exists both for a task and a server. Dual of a task T_i is denoted by T_i^*. The execution rate of dual is complement of execution rate $T_i.u$ of original task i.e. $T_i^*.u = 1 - T_i.u$. The deadline of dual is same as that of T_i. The same is true about a server.

2. A task with fixed rate is the one which executes with a fixed execution rate in any time interval. However we explain the *RUN* algorithm considering a periodic model where utilization factor corresponds to the rate

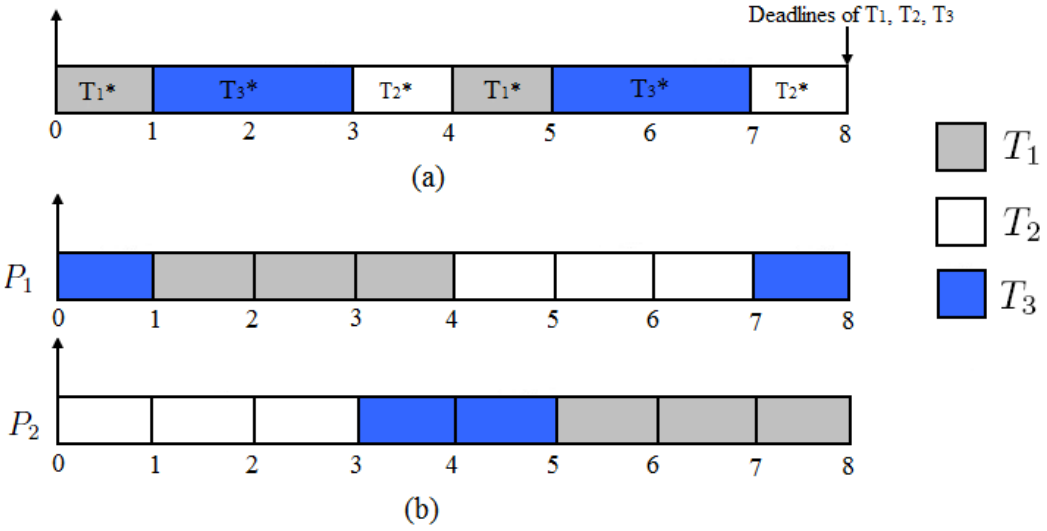


Figure 3.22: (a) Dual schedule (b) Primal schedule

In case of a periodic taskset with implicit deadlines the execution time of dual is $T_i^*.e = T_i.p - T_i.e$. The dual task represents the idle time of the task T_i. The schedulability rules associated to this duality relation is that a task and its dual task cannot execute at the same time.

The DUAL operation is used when $(N - M) < M$. For a system with $U = M$, the condition shows that most of the tasks have high values of utilization factors tasks because *U/M > 0.5*. In this situation the DUAL operation will generate a dual taskset with reduced total utilization factor which can be scheduled on less number of processors. If the primal tasks are schedulable on M processors then the dual tasks will be schedulable on $(N - M)$ processors. The schedule of dual a taskset is called dual schedule and the processor on which it is made is called dual processor. Once we come across a feasible dual schedule, the schedule of original taskset can be derived from the dual schedule. The schedule of the original taskset is obtained by blocking the schedule of a task on the original processor when its dual task is running on the dual processor.

Lets consider the dual schedule of the configuration given in the table 3.7. We have

Primal task T_i	$T_i.d = T_i.p$	$T_i.e$	Dual of task T_i	$T_i * .d$	$T_i^*.e$
T_1	4	3	$T_1{}^*$	4	1
T_2	8	6	$T_2{}^*$	8	2
T_3	4	2	$T_3{}^*$	4	2

Table 3.7: Original taskset and its dual

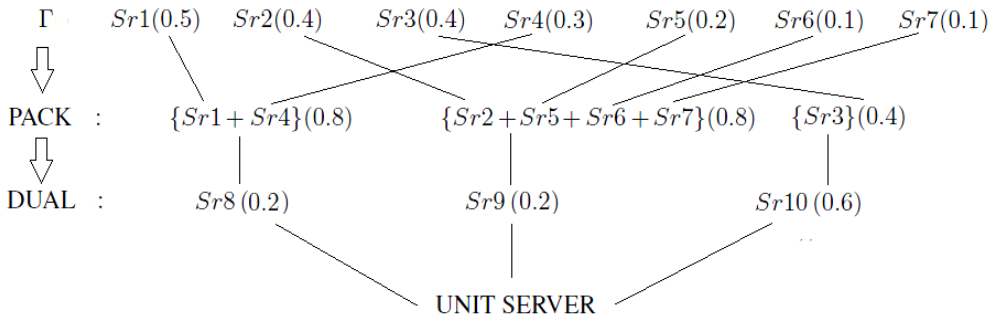


Figure 3.23: REDUCE operation

a periodic taskset with three tasks to be scheduled on two processors. The table shows the parameters of original or primal taskset and its dual taskset. We have scheduled the tasks upto the *LCM* of the periods which is 8. The dual of three tasks can be scheduled on a single processor (dual processor) using *EDF*. When the dual task $T_1{}^*$ is active between 0 and 1, the primal task cannot be scheduled on the original processor. The dual and primal schedules of the taskset given in the table 3.7 are given in the figure 3.22.

For the above discussed example, the conversion to a uniprocessor problem is done in a single step. But if this is not the case the *RUN* performs a sequence of transformations iteratively unless a single or multiple uniprocessor systems are achieved. Since the primal task and its corresponding dual task cannot be executed at the same time, each schedule in the resulting hierarchy controls the resulting scheduling of processors in the next level up.

PACK operation: The PACK operation also exists both for tasks and servers. It reduces the number of tasks or servers by aggregating them. The PACK process is used when $(N - M) \geq M$. It means that tasks have lower utilization factors and they can be grouped. The tasks are grouped and are associated to a server. The servers can also be grouped to make a set of servers in a hierarchical way. A set of servers is called a unit set if the sum of utilization factors of its servers is 1.

REDUCE operation: The DUAL and PACK processes constitutes the REDUCE process. The REDUCE operation continues on a taskset unless only unit servers are achieved. Once the sequence of REDUCE is done, the schedule of multiprocessor can be deduced from virtual schedules of the derived uniprocessor systems. The REDUCE operation on a set of servers Γ is shown in figure 3.23. The notation $Sr1(0.5)$ means server 1 with utilization factor 0.5. Since the $(N - M) \geq M$ condition is satisfied, we start with PACK operation. It is followed by a DUAL operation and it results in a unit server.

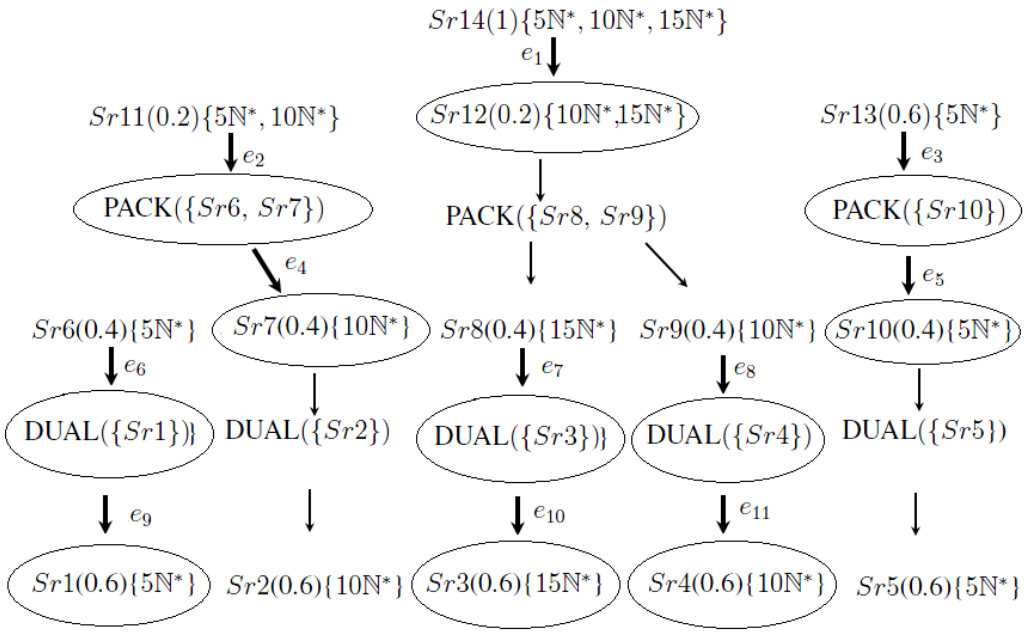


Figure 3.24: Server tree

3.4.2 Online scheduling

Once the offline process is completed using the REDUCE process, a server tree is created using inversion process. The root is a unit server which represents the top level virtual uniprocessor system. The children are packed or dual servers and it continues down to the leaves which are the original tasks.

The inversion process is shown in the figure 3.24 [55] with the help of a server tree. It explains how a schedule of original taskset is retrieved from a unit server. The notation $SrX(0.8)\{10\mathbb{N}^*, 15\mathbb{N}^*\}$ represents a server with name X with accumulated utilization factor of 0.8 and assigned deadlines of clients servers $10\mathbb{N}^*$and $15\mathbb{N}^*$. The unit server is the root and it represents the top level virtual uniprocessor. The root's children are the unit server's client which are scheduled by *EDF*. In the figure 3.24 the servers executing at each level at time a given time t are circled.

At each time t the M tasks that should be executing can be determined by applying 2 following rules [55]:

1. For *EDF* servers, if a packed server is executing, execute the child node with the earliest deadline among those children with work remaining. In the opposite case where the packed server is not executing , execute none of its children.
2. For dual servers, execute the child (packed server) of a dual server if and only if the dual server is not executing.

The rule 1 is seen in the edges $\{e_1, e_4, e_5, e_9, e_{10}, e_{11}\}$ and rule 2 is seen in the edges at $\{e_2, e_3, e_6, e_7, e_8\}$ in figure 3.24.

3.4.3 Performances

RUN shows better performance than existing optimal algorithms with an upper bound of $O(\log M)$ average preemptions per job on M processors. The most important thing is that preemptions per job do not depend on the number of processors. In [55], Regnier et al. mentioned that *RUN* gave less than 3 preemptions per job for all the task set on which simulation was performed. They also showed that *RUN* is better than *LLREF*, *DP-Wrap* and *EKG* in terms of preemptions.

3.5 Conclusion

In this chapter the two main classes of optimal global scheduling algorithms namely *PFair* and *BFair* are discussed. In the first part, the basic concept of *PFair* is discussed along with scheduling steps, *PFair* algorithms, limitations and extensions. In the second part, it presents *DP-Fair* scheduling, the principle and the scheduling steps. Thereafter it proposes a classification of *DP-Fair* algorithms on the bases of difference of either in computation of nodal execution time or dispatching technique. All the optimal standard *DP-Fair* algorithms proposed in the literature belong lie some where in this classification. However, certain possible combinations of the classification are still unused. For example, the use of *BFair* algorithm for computation of nodal execution time in *NNLF* results in an algorithm which can be practically implemented.

The *DP-Fair* is better than *PFair* as it has lower number of scheduling points. Out of different *DP-Fair* algorithms, we found that one which computes the integer value of nodal execution time and a dynamic dispatching is quite interesting. The reason is that it is practically possible to implement it. The following two algorithms belongs to this category:

- In the non-work conserving case, we used $BFair$[61] technique for the computation of nodal execution time while a dynamic technique of *LRE-TL* [30] for the task dispatching.
- In the work conserving model, we used again $BFair$ [61] technique for the computation of nodal execution time units and the dynamic dispatching rules of *NNLF* [29].

Although above discussed algorithms are optimal but these scheduling rules are only concerned with establishing an execution order for the tasks so that deadline may be met without specifying their execution on any particular processor. This is the reason why most of the scheduling algorithms give no explicit prescription about assigning the tasks to the processors or sorting the ready tasks.

RUN [55] is a recently proposed optimal algorithm. It is briefly introduced in the chapter but we do not have the time to include it in our study of reducing overhead.

4

Overhead control techniques

Overview

In this chapter, the precise definitions of the overheads and overhead control heuristics are presented. At first, the preemption and the migration are defined and then the heuristics which are meant to control the number of migrations and preemptions are described. Heuristics are given along with their algorithms and their complexities.

A significant amount of overhead in multiprocessor global scheduling is associated with the preemption and migration of tasks and therefore affects the efficiency of the system. It is sometimes quite high and is unbearable for real-time systems because these overheads give rise to timing penalties which may result in missing the deadlines.

To study the overhead due to migration and preemption, we consider a symmetric multiprocessor platform comprising a cache memory L1 on each processor and an external common cache L2 or main memory.

Cache memory is added to the processor to reduce the average time to access memory. It is a buffer memory, smaller and faster, which stores copies of the most recently used data. Later on when the processor requires to read or write the required data in main memory, it will first check the cache memory. A cache may be an instruction cache to speed up executable instruction fetch or a data cache to speed up data fetch and store. If processor finds the data in cache, it will read from or write to this cache memory rather than doing it on the main memory. Reading or writing data to the cache is much faster than reading or writing it to the main memory. Finding the data in the cache is called *cache hit* and is called *cache miss* otherwise.

The Average Memory Access Time *(AMAT)* is the average time to access the required data and is given by the following equation:

$$AMAT = HitRate * CacheAccessTime + MissRate * MissPenalty$$

where *CacheAccessTime* is the time needed to search the cache for required data. The *HitRate* is ratio of cache hits to total number of attempts to access cache. Similarly, the *MissRate* is the ratio of number of cache misses to the total number of attempts to access the cache. *MissPenalty* represents the total time required to access the data outside the cache after a cache miss from next level of cache or external memory.

Suppose we have a cache with access time of 10 nanoseconds and an external memory with access time of 100 nano seconds. We suppose that cache miss rate is 0.2 or we find the required date in cache 80 % of the time. Now the average memory access time for cache can be calculated as:

0.8 * 10 + 0.2 * (100 + 10) = 30 nanoseconds. Hence it results in great improvement relative to the situation without cache i.e. 100 nanoseconds.

If the size of a cache memory is big, better are the chances of finding the data in cache, but the time to search the cache increases. On the other hand, smaller is the cache memory lower is the possibility to have a cache hit but latency is improved. To find a compromise in this issue, multiple levels of cache are defined. L1 cache, built on the processor, is the fastest and most expensive cache in the computer. If this cache is missed, the next level largest cache L2 is checked.

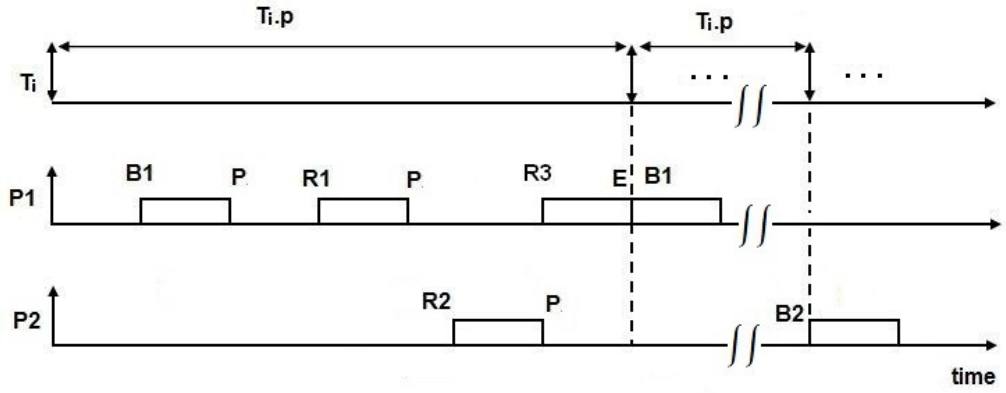


Figure 4.1: Events along a task execution

4.1 Events in task execution

To study the preemption and migration, consider the figure 4.1. It shows the execution of a task T_i. Various events may occur in the life of a task which may be distinguished as follows. Suppose :

- B is the event which corresponds to the start of a job execution. Two situations can be distinguished:
  - $B1$ is the case where the job starts on the same processor on which its previous job was executed;
  - $B2$ is the case where the job starts on a processor different from its previous one.
- E is the event which corresponds to the completion of job;
- P is the interrupt of a job due to preemption or suspension;
- R is an event which represents the resumption of job;
  - $R1$ is the case where a job is resumed on the very same processor on which it was executed last time;
  - $R2$ is the case where a job is resumed on a processor different from the one on which it was executed last time;
  - $R3$ is the case where a job is continued on a different processor without any waiting time.

4.2 Preemption

A task is preempted when it is temporarily interrupted by the task scheduler without requiring its cooperation, with an intention of resuming it later on. A running task is preempted when :

- A task of higher priority comes into the system and replaces it;
- Its execution is stopped before its completion due to the non-work conserving behavior of scheduling algorithm;
- The time slot alloted to the task in a round robin scheduling has elapsed.

Schedulers used in the most of real-time operating systems are fully preemptive. Some of the reasons behind this are :

- Preemption increases the schedulability;
- Preemption decreases the response time of critical tasks.

The cost of a preemption mainly comprises context switching. Indeed when a task is preempted a context switching takes place. As discussed in chapter 2, a context switching is the switching of the processor from one task to another. A context is the contents of a processor registers and program counter at any given time. When a task is preempted its context is saved and is retrieved when the task execution is resumed. These operations cause a delay.

The events involved in the preemption are mentioned in the table 4.1 along with other events.

4.3 Migration

A migration occurs when a task is executed on a processor which is different from the one on which it was executed last time. A migration is either at task level or job level (already discussed in section 2.4.2). A job migration is relatively costly as compare to a task migration where one complete job is executed on one processor. This is because a job migration causes a preemption of task in addition to the private cache miss which is not the case in task migration.

Migration of task has following advantages:

- It increases the schedulabilty. There are certain tasksets which are schedulable only when the tasks are allowed to migrate;
- Migration of the tasks allows the load balancing between the processors;
- Migration improves the response time of the system.

A migration is a consequence of a scheduling decision.

4.4 Overhead estimation

To identify the costs of overheads we use the events mentioned in the table 4.1. The table gives the name of the event, a brief comment about it with characteristics associated to each of them. We divide the total cost as the cost of *RTOS* services and the cost of scheduling algorithm. Micro architecture cost is the actual hardware delays due to each event.

It is quite difficult to calculate the exact value of hardware delays because the real cost depends on interaction between jobs and so on scheduling. This is a hot research topic in the field of "Static WCET analysis" which tries to take into account Cache Related Preemption Delay (CRPD) [3]. Some research work is also done in our team using a simulation approach (binary code simulation including OS and application) [1]. However, in this thesis we are not able to determine these delays, so they are not taken into account. Since we do not consider CRPD, events R2 and R3 can be merged.

Our work is concerned with a *time triggered system* using variable intervals. So the costs of the RTOS services are the ones invoked at the beginning of an interval and inside an interval. At each beginning of an interval we need to:

- Manage the jobs through some computations (RTOS service cost + a large scheduling cost);
- Preempt some jobs (event P, save context cost);
- Resume some jobs (event R, restore context cost)

Inside an interval we need to:

- Manage the jobs (RTOS service cost + a small scheduling cost);
- Preempt some jobs (event P, save context cost);
- Resume some jobs (event R, restore context cost)

We use the notation cost*(# Ev)* which denotes the overhead due to the number of occurrences *(#)* of the event *Ev*. So the overhead over a *LCM* time period can be computed as :

$$Overhead = cost((LCM/T_i.p) * (cost(B) + cost(E)) + cost(\#P) + cost(\#R)) \quad (4.1)$$

The number of *B* and *E* costs can easily be determined using the period of the task and the *LCM* of the configuration and for all the tasks in taskset. For the preemption and migration costs we need to count them during the execution of the configuration, the time of simulation being the *LCM* of the configuration.

Event type	Comment	Specific cost	RTOS cost	Micro architecture cost
B1	Release of a job on the same processor as for the previous release	Initialize context	Executed service (activateTask or Time Triggered) Scheduling	Variable (depend on the history of caches)
B2	Release of a job on a different processor as for the previous release	Initialize context	Executed service (activateTask or Time Triggered) Scheduling	Update at least first level cache for the job
E	End of a job	No cost	Executed service (terminateTask) Scheduling	No cost
P	Preemption of a job	Save context	No cost of service (preemption is the consequence of another event)	No cost
R1	Resume type 1	Restore context	No cost of service (resumption is the consequence of another event)	Variable (depend on the history of caches since the preemption of the job)
R2	Resume type 2 which is a migration	Restore context	No cost of service (resumption is the consequence of another event)	Update at least first level cache for the job
R3	Resume type 3 which is a migration	Restore context	No cost of service (resumption is the consequence of another event)	Update first level cache for the job (no misses on second level cache)

Table 4.1: Task events

It is obvious that the number of B and E events is constant. The only mean to reduce the overhead is to reduce the B_2 events (B_2 is more costly than B_1) using allocation decisions.

The number of preemptions and migrations depends on the scheduling decisions. So it is important to reduce them and more prior the number of migrations using allocation decision. Hence, it is more important to lower the number of migrations due to its negative impact on the cache performance (not taken into account here).

Therefore, we propose heuristics to do that.

4.5 Overhead control heuristics

As discussed in the conclusion of chapter 3, our *DP-Fair* model uses:

- *BFair* technique [61] for computation of integer value of nodal execution time;
- Dynamic dispatching technique of *LRE-TL* [30] for dispatching of taskset to the processors.

We name our model *BFair/LRE-TL*.

Once the nodal execution times for the taskset are computed and the simple scheduling rules are established, one can see that there is still room to design complementary dispatching strategies so as to reduce the number of task preemptions and migrations. Our intention is to explore this space and to decrease the overhead due to migration and preemption. Our proposal has been guided by few observations and gives rise to the related heuristics. These heuristics are used with *BFair/LRE-TL*.

The first heuristic results from the standard assumption of instantaneous preemptions and migrations. Then in theory it does not matter which processor is hosting a given task, but only which tasks are running at a given time. That is why most algorithms give no explicit description about how to assign tasks to processors. Thus, this heuristic deals with the task to processor assignment criterion.

The second one comes from the fact where it has been shown that the order in which the M tasks are selected for execution is not important, provided that they have non-zero nodal remaining execution times [30]. Thus, this heuristic is concerned with the running task selection criterion.

The two heuristics are not specific only to *BFair/LRE-TL* but can be used to any type of *DP-Fair* global scheduling to control the overhead due to migrations and preemptions.

4.5.1 Migration control heuristic (MCH)

This heuristic is meant to control the migrations. We also name it as Migration Control Heuristic (*MCH*).

Usually, running tasks are assigned to the available processors without considering their previous histories. According to *MCH*, a task keeps the record of the processor on which it was executed last time and then an affinity relation exists between task and processor. *MCH* takes into account this relation and if possible, tries to assign a newly running task to the same processor on which it was scheduled the last time. This heuristic is applied at main scheduling points i.e. those that coincide with time boundaries (or start of a *node*) as well as at secondary scheduling events (event *B* or event *C* occurrences inside a *node*). After the decisions of next running tasks has been taken, the computational complexity of *MCH* per plane is O(M). The steps are explained in algorithm 2.

Algorithm 2 Migration control heuristic *(MCH)*

Suppose
- H_B is the list of running tasks. Maximum size of H_B is M
- $getProc()$ returns the processor on which task was executed last time
- $setProc$(*X*) set X as the processor allocated to the task
- *isRunning()* returns true if the task was running at $t - \epsilon$ where ϵ is a number which is infinitely small
- P is an object that represents a processor
- $isIdle()$ returns true if the processor is free
- $searchIdleProc()$ returns an idle processor

Inputs H_B

@ b_k, *event C, event B*

1. **for**($T \epsilon H_B$)
2. **if** (!$T.isRunning()$)
3. $P = T.getProc()$;
4. **if** (!$P.isIdle()$)
5. $P = searchIdleProc()$;
6. $T.setProc$(*P*);
7. **end for**

4.5.2 Preemption control heuristic (PCH)

This heuristic basically controls the preemption and thus we also call it as Preemption Control Heuristic (*PCH*).

At a primary scheduling point, all the ready tasks have equal priority and atmost M of them can be chosen arbitrarily for execution. *PCH* attempts to control the preemptions at these scheduling points. According to this technique, the tasks executing on a processor just before the scheduling point are given priority to re-execute provided they still are ready. By continuing such executions, some unnecessary preemptions are avoided. This heuristic is applied only at main scheduling points.

Algorithm 3 Preemption Control heuristic)

Suppose
- *size()* returns the size of the list
- *add*(X) adds the object X to the list
- *getFirst()* returns the first object of the list
- *isRunning()* returns true if the task was running at $t - \epsilon$ where ϵ is a number which is infinitely small
- H_B is the list of running tasks. Maximum size of H_B is M
- $ReadyList$ is the list containing unsorted ready tasks

Inputs $ReadyList$
Outputs H_B

@b_k

1. **for** ($T \epsilon ReadyList$)
2. **if** ($T.isRunning()$)
3. $H_B.add(T)$;
4. **end for**
5. **while**((H_B.size () < M) && ($ReadyList$.size() > 0))
6. T = $ReadyList$.getFirst();
7. $H_B.add(T)$;
8. **end while**

The steps are explained in algorithm 3. The for loop checks for each task of the *ReadyList* at the start of a node. If any of them was running just before the end of the previous node, it will be given a priority to run again at the beginning of the node instead of having a preemption.

The computational complexity of *PCH* is O(max(M,N))

Name	The boundaries	Other than the boundaries
MCH	Yes	Yes
PCH	Yes	No
Hybrid	PCH+MCH	MCH

Table 4.2: Activation points of heuristics

4.5.3 Hybrid = (Migration + Preemption) control

When both *MCH* and *PCH* are used, we name the resulting algorithm as Hybrid algorithm. It keeps the benefits of both *MCH* and *PCH* and tries to reduce both migrations and preemptions.

When a main scheduling point arrives, it is *PCH* which starts working at first. It generates a list of running tasks H_B and after that *MCH* starts its operation. At a secondary scheduling point we use only *MCH*.

The table 4.2 shows the points where these heuristics are activated.

5

Experimental setup

Overview

The components involved in conducting the experiments to evaluate the performance of the overhead control heuristics are described in this chapter. The components include task generator, simulator and a tool for analyzing the results. The chapter explains the working of each component in detail.

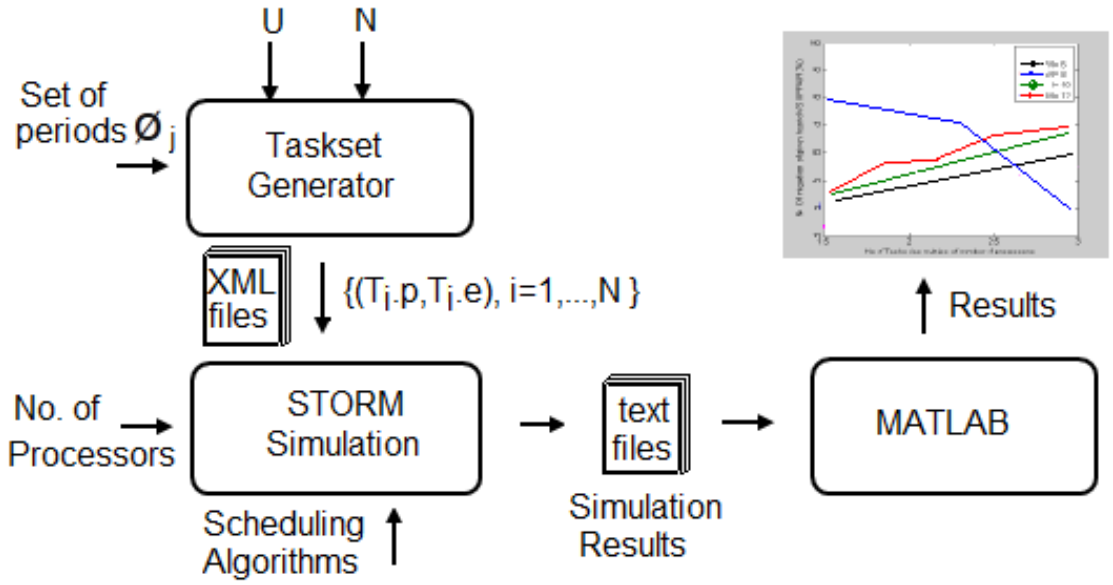


Figure 5.1: Experimental setup

Introduction

To study the performance of overhead control heuristics, we have used a statistical approach by testing them over a significant amount of data. The results help to analyze the behavior and efficiency of heuristics when they are used with different scheduling algorithms, their strengths, weaknesses and to suggest some improvements if possible.

The block diagram of our experimental setup is shown in figure 5.1. It includes a data generator, a simulator and an analysis tool based on MATLAB. Data generator builds data configurations under the constraints put forward by the user. These generated data are given to a multiprocessor scheduling simulator in the form of XML files. A XML file includes all the information about the taskset, indicates the scheduling algorithm and specifies the hardware architecture. The simulator simulates the execution of the taskset over the given hardware architecture according to the specified scheduling policy. The simulator outputs are stored in text files. These text files are treated by MATLAB which interprets the results in some analyzable form. In the next sections, each of these components are presented with more details.

5.1 Taskset generator

The main objective of our dataset generator is to produce a dynamic data to test the efficiency of scheduling algorithms for real-time multiprocessor system. A scheduling

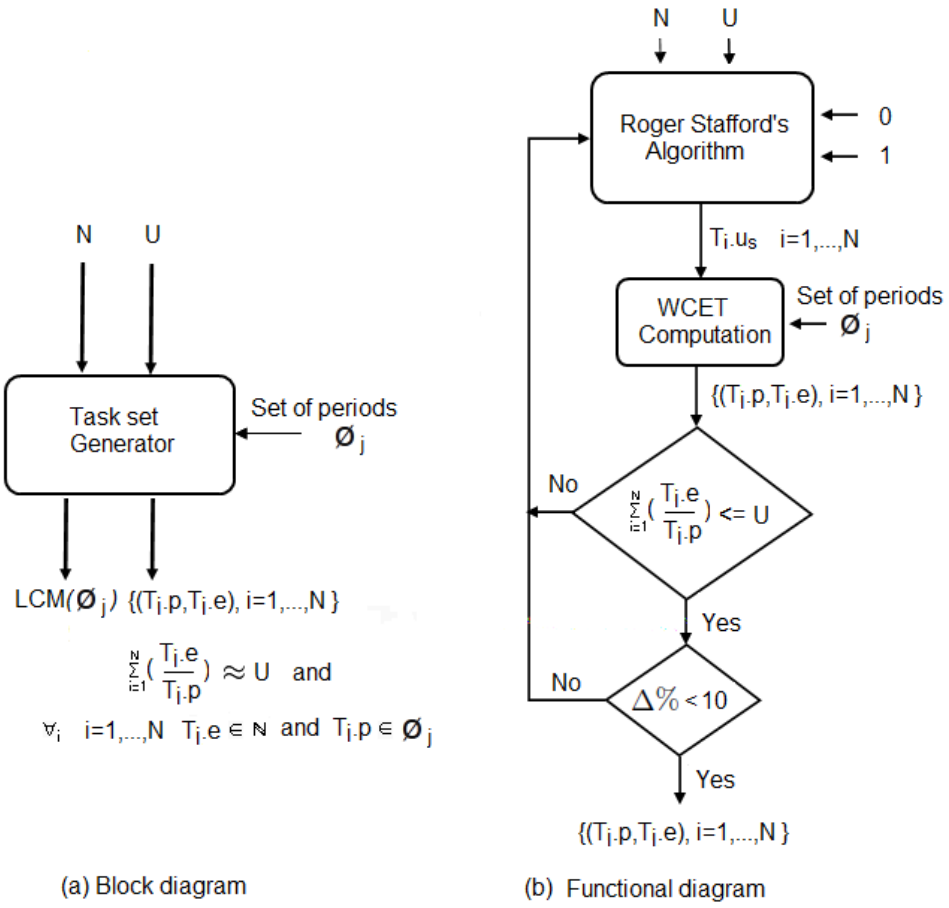


Figure 5.2: Task generator

algorithm needs to be tested extensively over a range of dataset to verify and analyze its efficiency and performance.

The figure 5.2 (a) shows the block diagram of the generator. It takes total utilization factor U, the number of tasks N in each taskset and a set of task periods ϕ_j as input while it gives N couples of $(T_i.e,\ T_i.p)$ such that $\sum_{i=1}^{N}(\frac{T_i.e}{T_i.p})$ is very close to U. The values all the task periods and the worst case execution times are integers.

Our generator generates a variety of data by varying the inputs. The following possibilities may be found:

1. By keeping a constant value of number of tasks in a taskset *N*, the generator builds configurations with variable total utilization factor *U* specified by user. In the same way, by keeping a single value of total utilization factor *U*, the number of tasks in each taskset *N* can be varied. This allows to control the utilization factors

of tasks in the taskset. The ratio *U/N* decides the general trend that if the tasks will be heavy ($T_i.u => 0.5$) or light ($T_i.u < 0.5$) in the output configuration;

2. The set of task periods can be varied. It gives rise the scheduling intervals of different lengths;
3. The generator does not require number of processors *M* as input. When a configuration is generated with a given *N* and *U*, it can be simulated with different values of *M*. The new configurations can be different with some specified ratio *N/M* or *U/M;*

The generator has a possibility of adding new inputs as well. With some simple modifications we could fix the lower and higher values of the utilization factor of the tasks at input.

The functional diagram of the taskset generator is shown in figure 5.2 (b). We used Roger Stafford's randfixedsum algorithm [59] at the heart of the taskset generator. The Stafford's algorithm efficiently generates a fixed number of random real values lying in the interval [a, b] such that their sum gives a specified real number. The MATLAB implementation of the algorithm is publicly available with all necessary documentation [59].

The time periods $T_i.p$ are chosen from a set of pre-defined time periods ϕ_j in a round robin way. Using fix values of task periods helps to limit the hyper period of the taskset and thus bounds reasonably the simulation interval. Moreover, the utilization of different period sets gives rise to various distributions of node length and frequency.

In our case, Stafford's randfixedsum algorithm generates N values of utilization factor $T_i.u_s$ in the interval [0, 1] such that their sum is equal to the total utilization factor U. The value of $T_i.e$ is calculated by taking a time period $T_i.p$ value from period set *T* using round robin way and using $T_i.e = T_i.u_s * T_i.p$. However, the obtained value of $T_i.e$ may not be an integer value. Therefore, an algorithm is used to convert it into the closest lower integer value while taking into account the possible error in the next steps. The algorithm 4 explains it.

The taskset is discarded if the total utilization factor finally calculated is more than the required U. Due to restricting $T_i.e$ to an integer, a difference or error is produced between the utilization factor obtained by Stafford's algorithm $T_i.u_s$ and final utilization factor obtained $T_i.u$. The taskset is not valid if this average error is greater than 10%. This value is a good compromise between the initial Stafford's distribution and a moderate time to generate a large number of configurations.

The table 5.1 gives the average percentage error $\triangle\%$ of the tasks at variable ratio of total utilization factor to number of tasks. This is calculated according to the line 8 of

Algorithm 4 WCET computation

Suppose
$T_i.u_s$ = Utilization factor of task T_i produced by Stafford's algorithm
$T_i.u$ = Utilization factor of task T_i after modification
ϕ_j= {ϕ_{j0}, ϕ_{j1},... ϕ_{jk}}
$\Delta\%$ = Percentage error

1. $\Delta_0 = 0$;
2. $\Delta\% = 0$
3. ***for*** ($i = 1...N$)
4. $T_i.u' = min\left\{\left(T_i.u_s + \Delta_{i-1}\right), 1\right\}$
5. $T_i.p = \phi_{j(i\,\%\,k+1)}$
6. $T_i.e = max\{\lfloor T_i.p \times T_i.u' \rfloor, 1\}$
7. $\Delta_i = T_i.u_s - \frac{T_i.e}{T_i.p}$
8. $\Delta\% = \Delta\% + \frac{|\Delta_i|}{T_i.u_s}$
9. ***end for***
10. $\Delta\% = \frac{\Delta\%}{N}$

algorithm 4. Each value in the table is an average of 30 random generations with 100 tasks in each taskset. The set which we have used for this generation is ϕ_1 = {30, 36, 40, 45, 50}.

U/N	$\triangle\%$
0.25	7.888
0.5	4.793
0.75	1.977

Table 5.1: Percentage error

5.1.1 Properties of task period set

We have used four different sets of task periods with five different values in each set. By varying the values of period in each set, scheduling intervals with different length

and frequency are generated. The task periods we have used give rise to scheduling intervals of variable length between 2 and 30.

The four period sets are :

1. ϕ_1 = {30, 36, 40, 45, 50};
2. ϕ_2= {30, 35, 40, 50, 100} ;
3. ϕ_3 = {30, 45, 90, 150, 200};
4. ϕ_4 = {30, 35, 60, 70, 90}.

5.1.1.1 Set 1

ϕ_1 = {30, 36, 40, 45, 50];
Hyper period = 1800;

The length of scheduling intervals found upto the hyper period and their frequency are given in the table 5.2. This set produces intervals with 12 lengths. Small, medium and large intervals are present in a fairly good proportion.

Length of interval	Frequency of interval	%
2	4	2.38
4	12	7.14
5	16	9.52
6	20	11.90
8	12	7.14
9	8	4.76
10	38	22.61
12	16	9.52
15	16	9.52
18	8	4.76
20	12	7.14
30	6	3.57
	$\sum$ = 168	100

Table 5.2: Scheduling intervals of ϕ_1

5.1.1.2 Set 2

ϕ_2 = {30, 35, 40, 50, 100};

Hyper period = 4200;

The length of scheduling intervals found upto the hyper period and their frequency are given in the table 5.3. This set produces intervals with 6 lengths with relatively higher proportion of smaller lengths.

Length of interval	Frequency of interval	%
5	72	21.42
10	144	42.85
15	36	10.71
20	60	17.86
25	12	3.57
30	12	3.57
	$\sum$ = 336	100

Table 5.3: Scheduling intervals of ϕ_2

5.1.1.3 Set 3

ϕ_3 = {30, 45, 90, 150, 200};
Hyper period = 1800 ;

Length of interval	Frequency of interval	%
5	2	2.33
10	6	6.97
15	38	44.19
20	4	4.65
30	36	41.86
	$\sum$ = 86	100

Table 5.4: Scheduling intervals of ϕ_3

The length of scheduling intervals found upto the hyper period and their frequency are given in the table 5.4. This set produces intervals with 5 lengths with medium and large lengths in relatively higher proportion.

5.1.1.4 Set 4

ϕ_4 = {30, 35, 60, 70, 90};

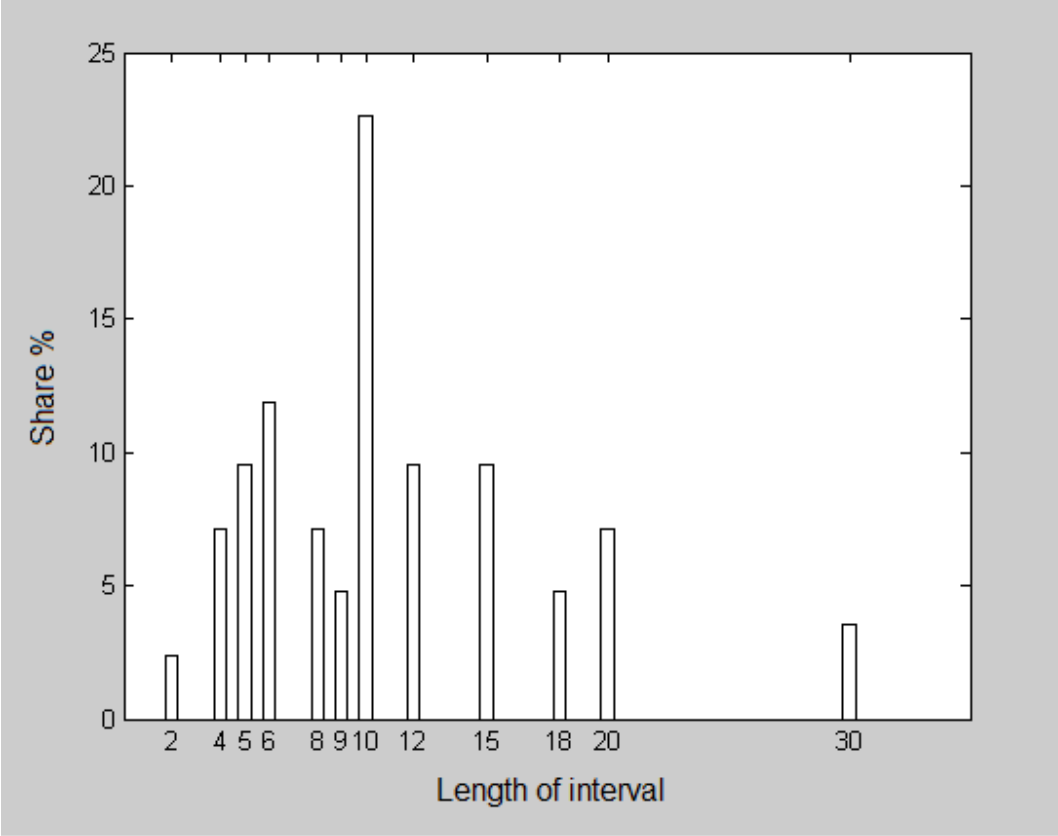


Figure 5.3: Variation of intervals ϕ_1

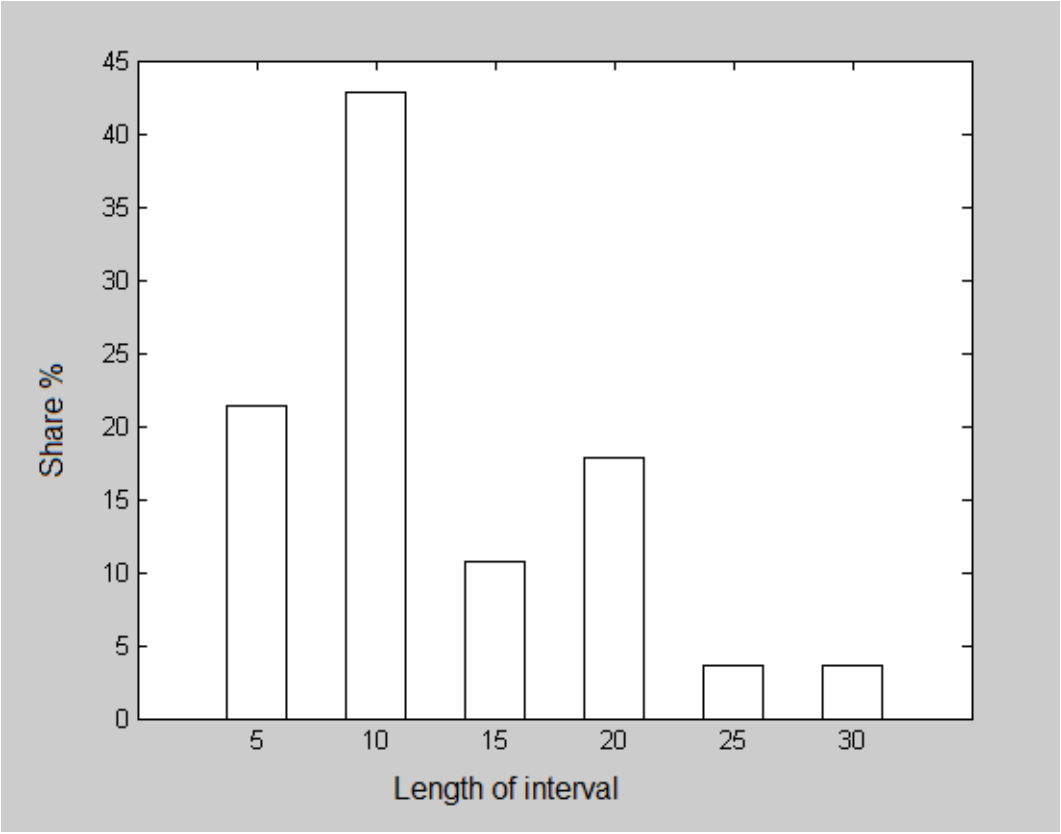


Figure 5.4: Variation of intervals ϕ_2

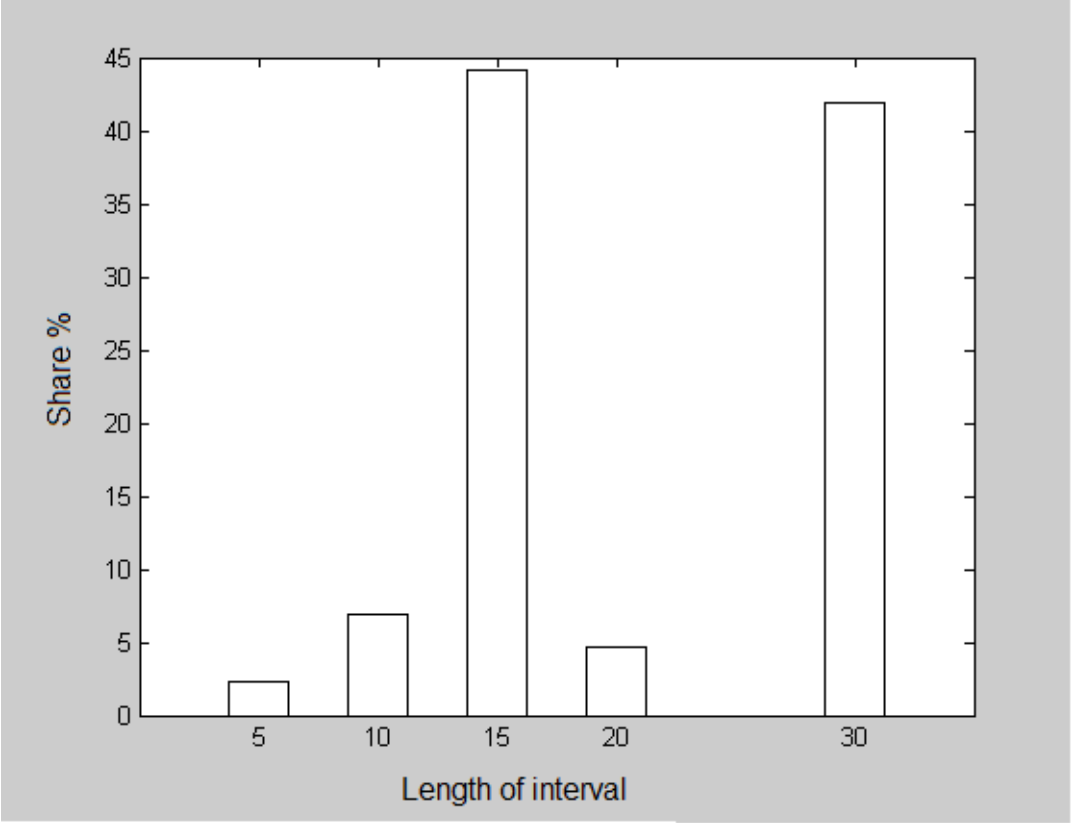


Figure 5.5: Variation of intervals ϕ_3

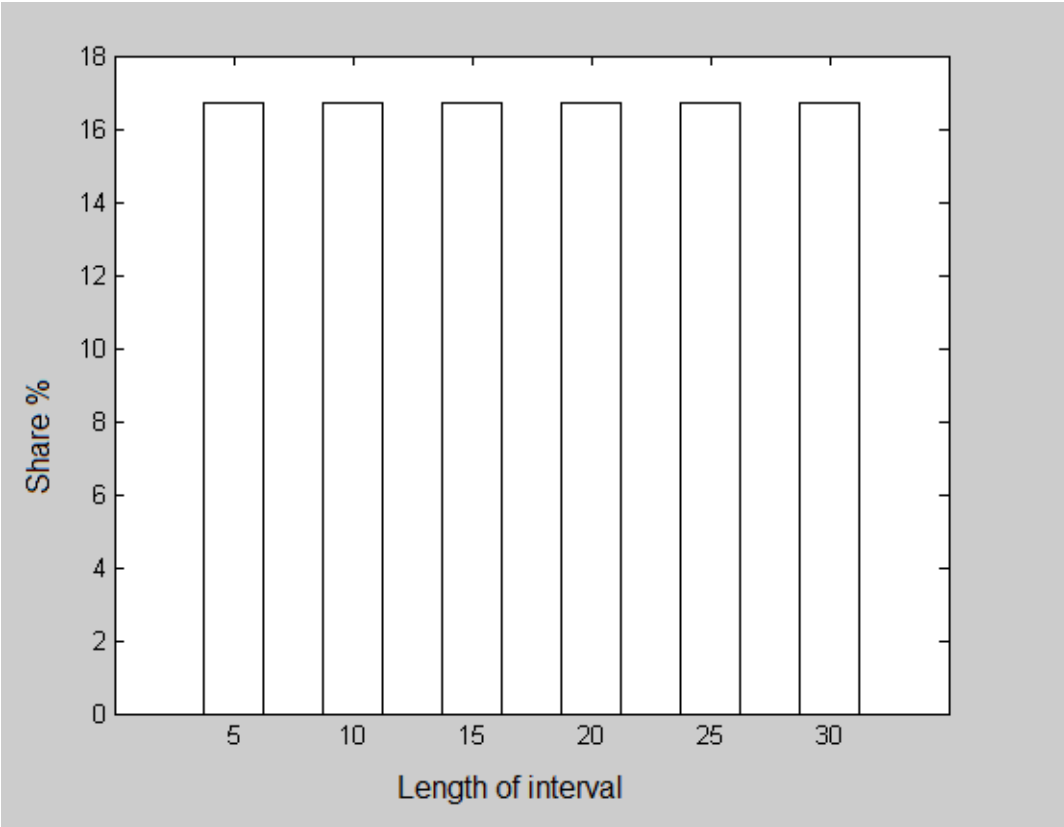


Figure 5.6: Variation of intervals ϕ_4

Hyper period = 1260

The length of scheduling intervals found upto the hyper period and their frequency are given in the table 5.5. This set produces intervals with 6 lengths and all the intervals lie in equal proportion.

Length of interval	Frequency of interval	%
5	12	16.7
10	12	16.7
15	12	16.7
20	12	16.7
25	12	16.7
30	12	16.7
	$\sum$ = 72	100

Table 5.5: Scheduling intervals of ϕ_4

5.1.2 Distribution of task utilization factors

A balanced taskset should have tasks with all possible variations in utilization factors depending on the total utilization factor. It should have light tasks i.e. the tasks with utilization factor less than 0.5 and heavy tasks which have utilization factor higher than 0.5. The following section explains how the utilization factor $T_i.u$ of the tasks are distributed. It explains how the task utilization factor varies as the ratio of U to N changes.

The bar graphs in the figures 5.7, 5.8 and 5.9 show the number of the tasks over a range of utilization factors with variable values of U and N. The vertical axis shows the number of the tasks while the horizontal axis represents the task utilization factor. Each value in the graph represents an average of 30 tasksets. The number of tasks in each taskset was 100.

At U/N = 0.5, there is a uniform distribution i.e. the task utilization factors lie uniformly between 0 and 1. At U/N = 0.25, the number of tasks with relatively lower utilization factors dominate which is quite natural. At U/N = 0.75, there is an opposite trend as that of U/N = 0.25 and tasks with relatively higher utilization factors dominate.

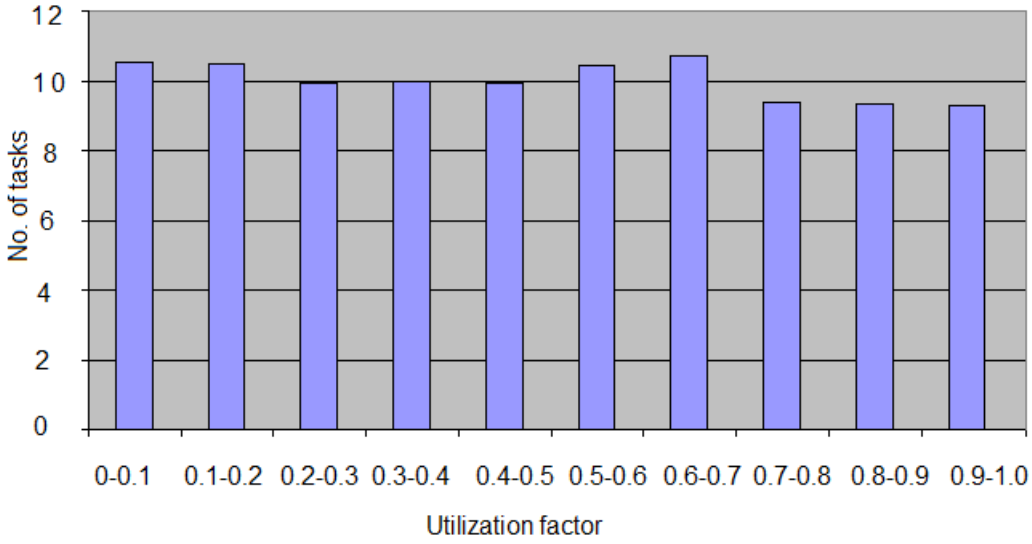


Figure 5.7: Distribution of utilization factor for U/N = 0.5

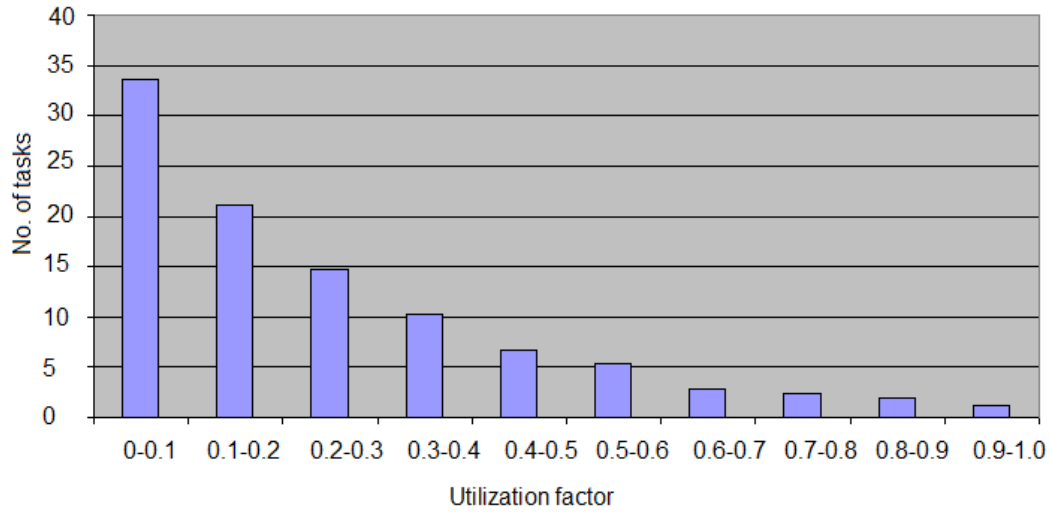


Figure 5.8: Distribution of load for U/N = 0.25

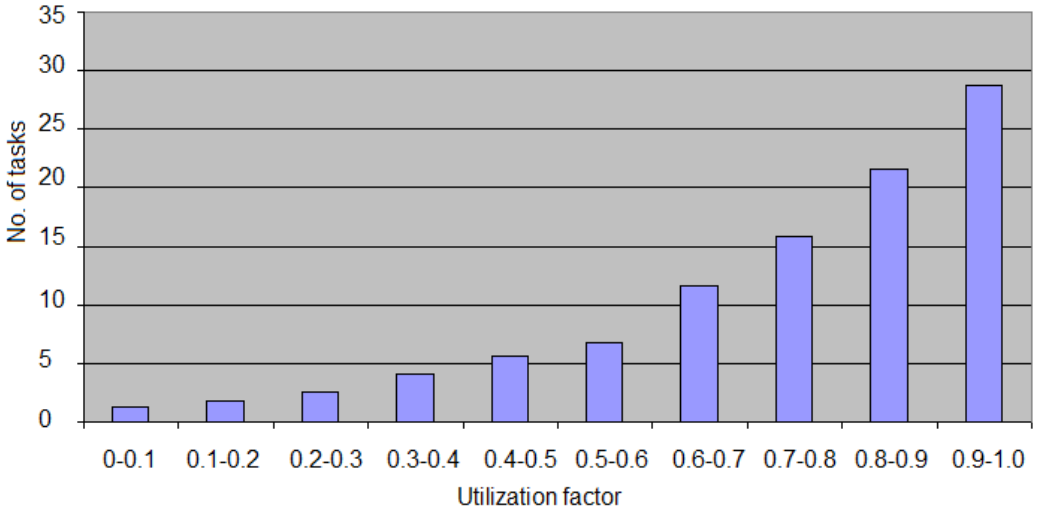


Figure 5.9: Distribution of load for U/N = 0.75

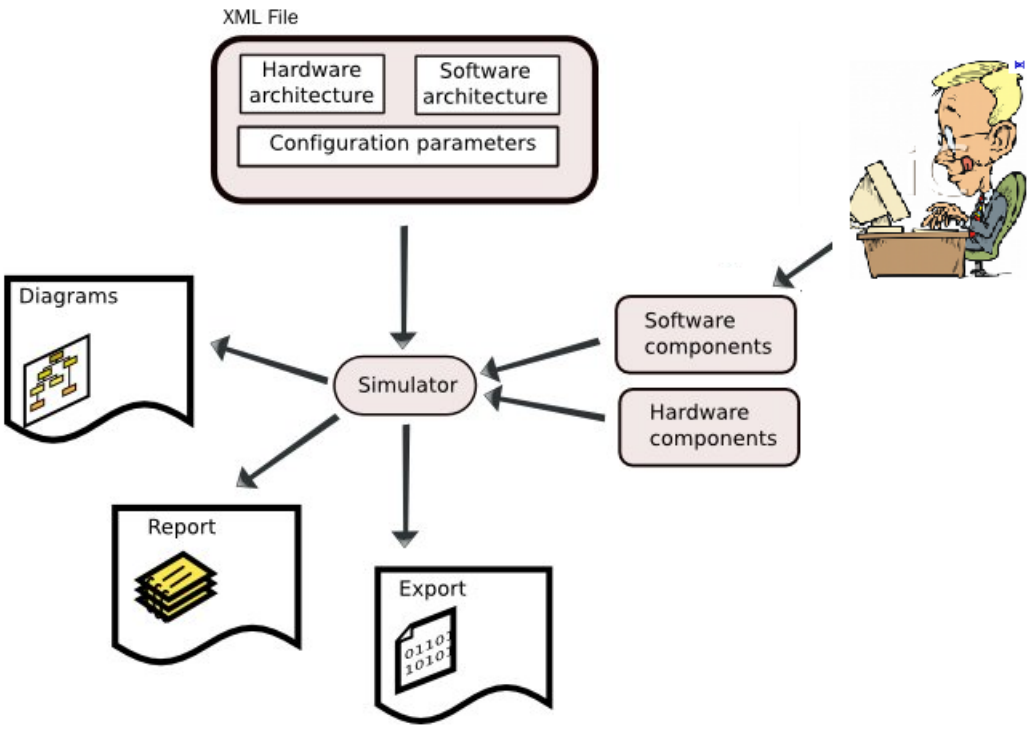


Figure 5.10: Architecture of STORM

5.2 Simulator

5.2.1 Short overview of STORM

We used STORM [2, 60] as the simulation tool. STORM stands for "Simulation TOol for Real time Multiprocessor scheduling". STORM (http://storm.rts-software.org) is a freeware software tool developed in our research team. It is able to simulate the behavior of predefined or user-defined real-time multiprocessor schedulers and to evaluate their performances by computing specified metrices on the schedules it constructs. It has ability to show the execution of given set of tasks over a multiprocessor architecture while taking in account the requirements of both taskset and hardware system. The structure of STORM is given in the figure 5.10.

STORM takes input in the form of an XML file. In our case, XML file will be the output of the taskset generator. It includes all the necessary information about the software system and hardware system:

- Number and type of processors;
- Number and type of tasks;

```
<SIMULATION duration="840">
<SCHED name="sched" className="storm.Schedulers. xxx .SchedulingAlgoName"
<CPUS>
 <CPU className="storm.Processors.CT11MPCore" name="CPU A" id="1" />
 <CPU className="storm.Processors.CT11MPCore" name="CPU B" id="2" />
</CPUS>
<TASKS>
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T1" id="3" period="8" deadline="8" activationDate="0" WCET="4" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T2" id="4" period="12" deadline="12" activationDate="0" WCET="3" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T3" id="5" period="15" deadline="15" activationDate="0" WCET="10" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T4" id="6" period="12" deadline="12" activationDate="0" WCET="1" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T5" id="7" period="14" deadline="14" activationDate="0" WCET="7" />
</TASKS>
</SIMULATION>
```

Figure 5.11: An input XML file for STORM

- Timing parameters of each task;
- Scheduling policy;
- Duration of simulation.

An example of such a XML file is given in the figure 5.11. Here, one can see that 2 processors i.e. CPU A and CPU B, are declared along with five tasks. All the tasks are of the same type and the parameters like task period, deadline, activation time and worst case execution time are specified for each of them. The duration of simulation is also given.

STORM provides some basic scheduling algorithms like *EDF*, *LLF,* etc. in its library. However, user can write new scheduling policies which can be easily used in the system. The scheduler program is written in Java. Through well defined APIs, STORM specifies different methods required to implement the scheduling strategy. Following the rules of inheritance, these methods have to be overrided. User can also define his own dynamic attributes to fulfill the programming requirements of new and relatively complex scheduling policies other than already given by STORM. STORM is discrete time simulator. At each tick of simulation the scheduler component is called.

The output is given in the form of Gantt diagrams for tasks and processors which can be read directly or in the form of text files which can be used by a subsequent analysis tool.

For more details, the reader is invited to have a look at `http://storm.rts-software.org`

5.2.2 Implemented algorithms

Scheduler algorithms

We have implemented the following scheduling algorithms.

1. *BFair/LRE-TL* *(NWC)*

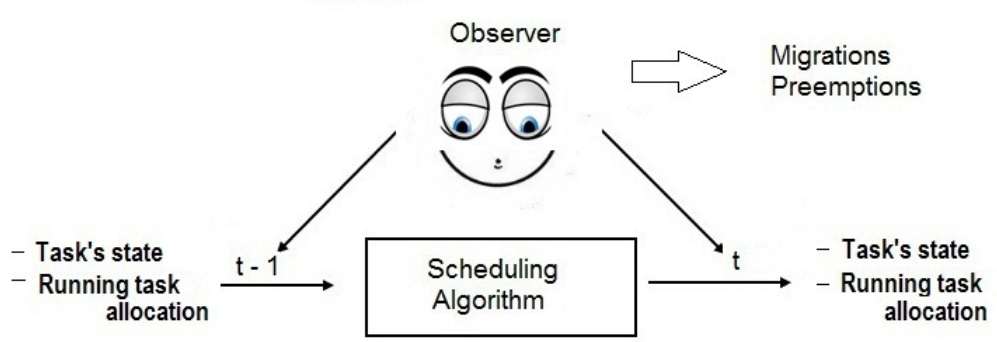


Figure 5.12: Observer program

2. *BFair/LRE-TL* + Migration control heuristic = *MCH* *(NWC)*
3. *BFair/LRE-TL* + Preemption control heuristic = *PCH* *(NWC)*
4. *BFair/LRE-TL* + *MCH* + *PCH* = *Hybrid-NWC* *(NWC)*
5. *BFair-NNLF/LRE-TL* = *NNLF* [1] *(WC)*
6. *BFair-NNLF/LRE-TL*+ *MCH* + *PCH* = *Hybrid-WC* *(WC)*

Overhead measuring program

We wrote a program called *"observer"* for measuring the overhead in our simulation process. The observer has been implemented as a Java class outside the scheduler one without an involvement with it. At each tick of time, the observer counts the number of preemptions and migrations by reading the present and previous values defining the execution state of the tasks. The working of observer is given in algorithm 5.

5.3 Experimentation process

Step 1: Space exploration of experimental data

We have generated experimental data in variation with input parameters.

1. In the *NNLF* presented in the literature, the initial nodal execution time is proportional to the utilization factor of task and free units are arbitrarily distributed among the ready tasks. But in our implemented *NNLF,* we used *BFair* for computation of $T_i.l$ and the task one with the lower value of remaining execution time is given preference for allocation of free units because it lowers the migrations and preemptions.

Algorithm 5 Observer program

Suppose
AET is the time for which the job has executed
$WCET$ is the worst case execution time
$Preemption$ is the number of preemptions
$TaskMigration$ is the number of task migrations i.e. switching of a task to a different processor when a new job is released
$JobMigration$ is the number of job migrations i.e. switching of a task to a different processor during a job
$T.isrunning(t)$ returns true if task T is running at time t

@ each simulation tick t

1. **for** each task $T \epsilon\ ReadyList$
2. **if** (!$T.isrunning(t)$)
3. **if** ($T.isrunning(t-1)$)
4. **if** ($AET\ != WCET$) // the job is not completed
5. $Preemption++$
6. **end if**
7. **else** // T is running
8. **if** (T runs on a different processor from the last one)
9. **if** ($AET == 0$) // a job begins
10. $TaskMigration++$
11. **else** // the job continues
12. $JobMigration++$
13. **endif**
14. **end elseif**
15. **end for**

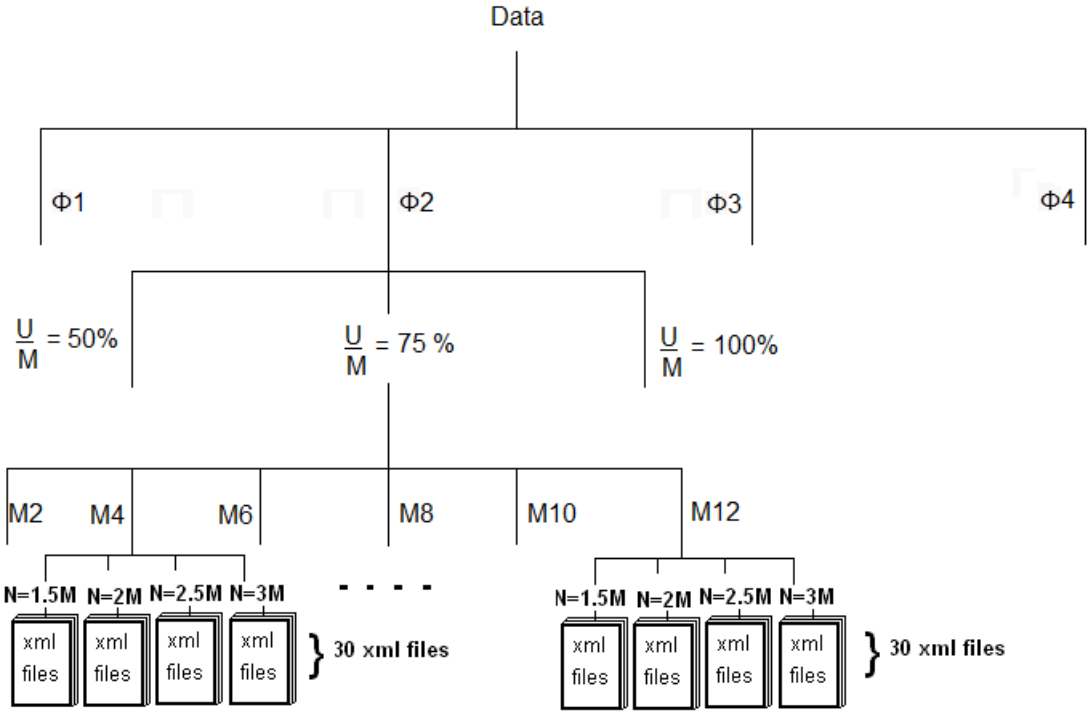


Figure 5.13: Data generation tree

For each set of task periods ϕ_j :

1. The process of experiments is conducted at variable total utilization factor, i.e. U=M, U=0.75M and U= 0.5M;
2. For each value of total utilization factor, experiments are performed with varying number of processors including 2, 4, 6, 8, 10 and 12;
3. For each value of processors, four different values of number of tasks are used, i.e. N = 1.5M, N = 2M, N = 2.5M and N = 3M;
4. For each value of number of tasks, 30 configurations are generated.

In short, the above discussed data distribution is given in the figure 5.13.

1. For a type $\{\phi_j, \frac{U}{M}, M, N\}$, 30 configurations are available;
2. For a type $\{\phi_j, \frac{U}{M}, M\}$, 120 configurations are available;
3. For a type $\{\phi_j, \frac{U}{M}\}$, 720 configurations are available;

U/M	1				0.75				0.5			
N/M	1.5	2	2.5	3	1.5	2	2.5	3	1.5	2	2.5	3
U/N	0.67	0.5	0.4	0.33	0.5	0.38	0.3	0.3	0.33	0.25	0.2	0.17

Table 5.6: *U/N* for given vaues of *(U/M, N/M)*

4. For a type $\{\phi_j\}$, 2160 configurations are available;
5. In total we generated 8640 configurations;

Step 2: Simulation

1. We perform simulation of the 8640 configurations;
2. One file stores the results of 30 configurations of same $\{\phi_j, \frac{U}{M}, M, N\}$.
3. In total there are 288 result files.

Step 3: Analysis of experimental results

We have wrote the MATLAB program which analyzes the result files and gives the output in the graphical form. The program reads the result files in the different directories, computes an average and presents the output in different ways to show the characteristics more clearly. The program gives following different types of output:

1. Overhead control of different algorithms at variable U/M for a given value of N;
2. Overhead control of hybrid algorithm at variable U and M for a given value of N;
3. Overhead control of hybrid algorithm at variable N/M for a given value of U ;
4. Overhead control of *Hybrid* algorithm at variable U/N. The table 5.6 gives variable values of *U/N* that are associated to the various couples of *(U/M, N/M)*. We can observe that the generated configurations are mostly composed of light tasks.

6

Experimental results

Overview

In this chapter, the results of the simulation process carried to find out the efficiency of overhead control heuristics are presented. The simulation process was completed using a variety of data as discussed in data generation process in chapter 5. The results are given for both non-work conserving and work conserving cases.

We performed a series of simulation based experiments to find out the efficiencies of overhead control heuristics which reduce the number of preemptions and migrations. It is the *observer* program, discussed in section 5.2.2 of chapter 5, which measures the numbers of preemptions and migrations.

1. The observer measures the numbers of migrations and preemptions of reference *DP-Fair* algorithm over a duration equal to the *Hyperperiod* of taskset;
2. The observer measures the numbers of migrations and preemptions of reference *DP-Fair* algorithm with our overhead control heuristics. These values are plotted as percentage of the corresponding values of reference *DP-Fair* algorithm.

The chapter is divided into two parts: In first part, the results of non-work conserving case are presented while the results of work conserving algorithms are mentioned in the second part. In each of them we have further categorized the results into migration control and preemption control. Different perspectives of the obtained results are given for each case.

6.1 Non-work conserving case

In non-work conserving case we have used *BFair/LRE-TL* as the reference *DP-Fair* algorithm. The heuristics include *PCH (Preemption Control Heuristic)* and *MCH (Migration Control Heuristic).* There are four following algorithms involved in the whole process:

1. *BFair/LRE-TL*
2. *BFair/LRE-TL + PCH = PCH*
3. *BFair/LRE-TL + MCH = MCH*
4. *BFair/LRE-TL + MCH + PCH = Hybrid-NWC*

For each of these algorithms, we count the number of migrations and preemptions and expressed them as percentages relative to the corresponding value of the basic *BFair/LRE-TL*. For sake of convenience we will denote *BFair/LRE-TL + PCH* only as *PCH and BFair/LRE-TL + MCH* as *MCH*.

6.1.1 Migration control (NWC)

6.1.1.1 Different algorithms at U=M, U=0.75M, U=0.5M

The figures 6.1, 6.2 and 6.3 show the migration control of different algorithms at *U=M*, *U=0.75M* and *U=0.5M* respectively varying number of processors from 2 to 12. The horizontal axis represents the number of processors while the vertical axis shows

the number of migrations of given algorithm in reference to migrations of *BFair/LRE-TL* in terms of percentage.

Each point on a graph represents an average result of experiments conducted on 120 tasksets [1]. The configurations include tasks with different utilization factors, light as well as heavy tasks.

The results in figure 6.1 show that migrations with *MCH* are about 60% of migrations of *BFair/LRE-TL* algorithm at *U=M*. In other words it reduces migrations by 40%. The migration control is approximately remains the same over the number of processors with very slight changes.

In the figure 6.1, the migration control of *PCH* is approximately same as that of a *MCH* and gives *60%* of migrations of *BFair/LRE-TL. PCH* basically controls the preemptions but it reduces the migrations as well because tasks which avoid preemption by using *PCH* may have been migrated to different processors in case of a preemption.

Migration control of *hybrid-NWC* is better than both *MCH* and *PCH* and shows around 52% migrations of *BFair/LRE-TL* algorithm.

In figure 6.2, the migration control of *MCH* at *U=0.75M* is better when compared with migration control at *U=M*. This is because with lower value of total utilization factor, there are more chances of availability of early free processors and consequently there are more chances of task execution on the same processor.

However, the *PCH* is less efficient in migration control at *U=0.75M* than at *U=M*. This is due to the fact that *PCH* works only if the task execution continue upto the end of the node. At *U=0.75M*, the chances of task to continue upto the end of node decreases and hence the migration control of *PCH* is less efficient.

The *hybrid-NWC* is better than both *MCH* and *PCH* and gives 48% of migration of *BFair/LRE-TL*.

The same behavior of both *MCH, PCH* and hybrid continues at *U=0.5M* as shown in figure *6.3*. The migration control of *MCH* is even better than at *U=0.75M* and that of *PCH* even deteriorates than at *U=0.75M*.

The *Hybrid-NWC* algorithm improves its migration control as the total utilization decreases due to the presence of *MCH* in it.

1. Due to space limitations we have presented the results of simulation of taskset period ϕ_1. Also for *M = 2*, results exist only for *N = 2.5M* and *N = 3M* because these two values allow to use all values of task period set indifferent to *N = 1.5M* and *N = 2M*. For example for *N = 1.5M,* we can use only three values of task period which can result in different value of hyperperiod. Hence at M=2, each point represents an average result of experiments conducted on 60 tasksets.

6.1.1.2 Hybrid-NWC algorithm at different sets of taskperiod

The graphs in the figures 6.4, 6.5 and 6.6 show the migration control of the hybrid algorithm for all the four sets of task periods at *U=M*, *U=0.75M* and *U=0.5M* respectively. Each point in the graph shows an average result of experiments conducted on 120 tasksets. The general trend remains the same for all the sets of taskperiod.

The figure 6.4 shows that at *U=M*, the hybrid algorithm gives best performance for ϕ_1 which maximum varying lengths of scheduling intervals. For ϕ_3, hybrid shows the least control of migrations where there are few intervals of small length and most of the intervals are either medium length or of large length.

All the four tasksets also show a consistent behavior at *U=0.75M and U=0.5M*. At *U=0.75M,* the migration control of all the sets is better than at *U=M* and migration control *at U=0.5M* works even better than at *U=0.75M*. In both *U=0.75M* and *U=0.5M,* the migration control is relatively better at lower values of processor except at *M=2*. At *U=0.5M,* the values of migration control of different tasksets are pretty close to each other in different to the values at *U=M* and *U=0.75M*.

6.1.1.3 Hybrid-NWC algorithm at variable task to processor ratio

Figures 6.7, 6.8 and 6.9 show the migration control of hybrid algorithm of ϕ_1 at *U=M*, *U=0.75M* and *U=0.5M* varying the task to processor ratio. The horizontal axis represents the ratio of task to processor while the vertical axis remains the same.

Number of tasks N varies between 6 and 36 in relation with the number of processors. Each point in the graph shows an average result of experiments conducted on 30 tasksets. The results do not include *M=2* due to unavailability of results for *N=1.5M* and *N=2M*.

At *U=M*, the migration control of hybrid is slightly better at lower values of task to processor ratio than at higher values. This is because with more number of tasks on each processor, there are fewer chances of task re-execution on the same processor. This is more prominent at *U=0.75M* and *U=0.5M* than at *U=M* as shown in figures 6.8 and 6.9 respectively. But as discussed earlier, the migration control works better at lower value of total utilization factor. The migration control is more significant at *N=1.5M*.

6.1.1.4 Hybrid algorithm for U/N

The figure 6.10 shows the migration control of *hybrid-NWC* of ϕ_1 varying the ratio of *U* to *N*. The horizontal axis represents the ratio of *U to N*. The graph shows the overhead control as function of light and heavy tasks. It shows that there is no significant effect of *U/N* on migration control.

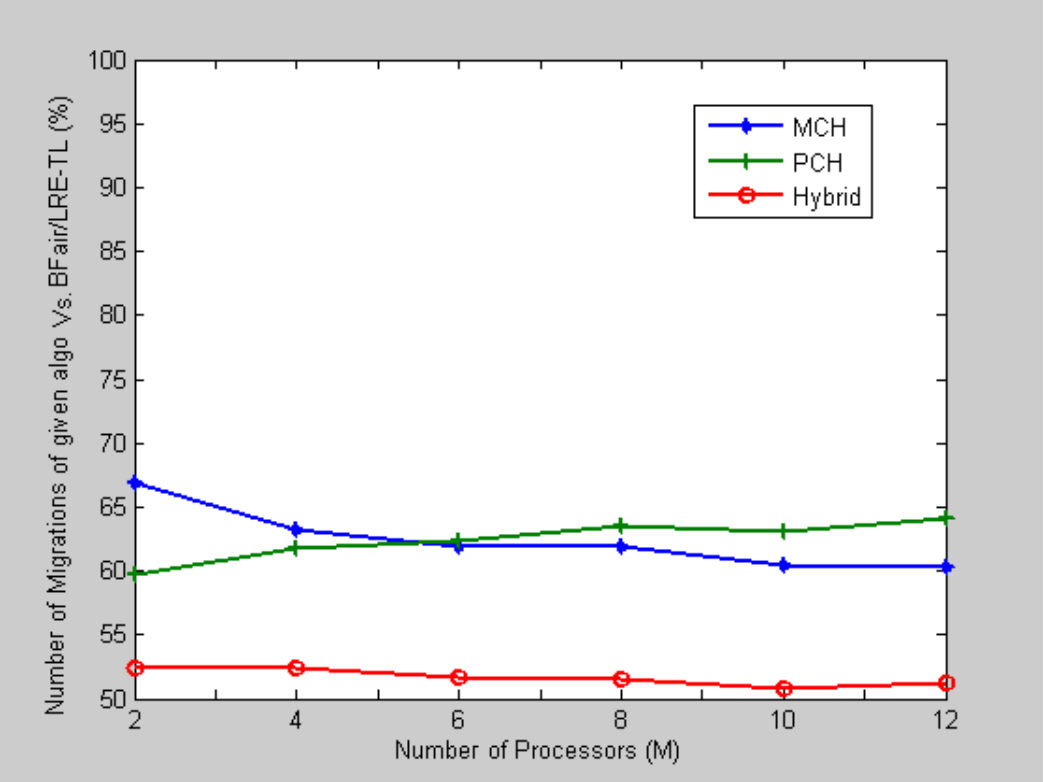


Figure 6.1: Migration control at U=M

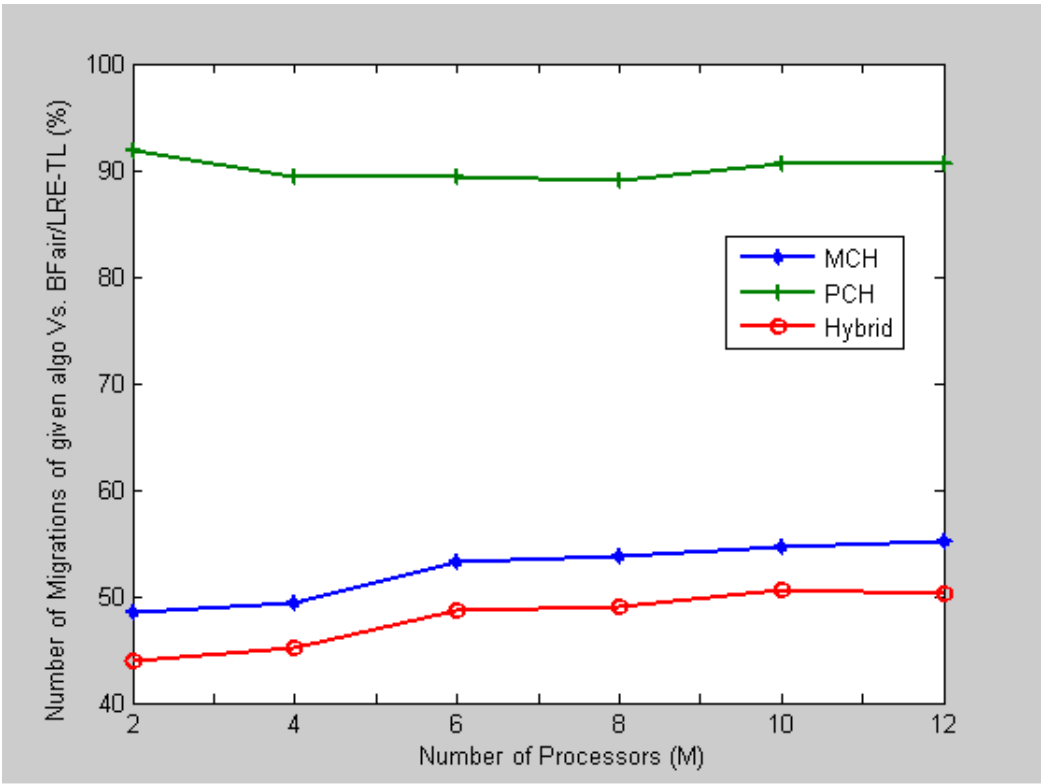


Figure 6.2: Migration control at U=0.75M

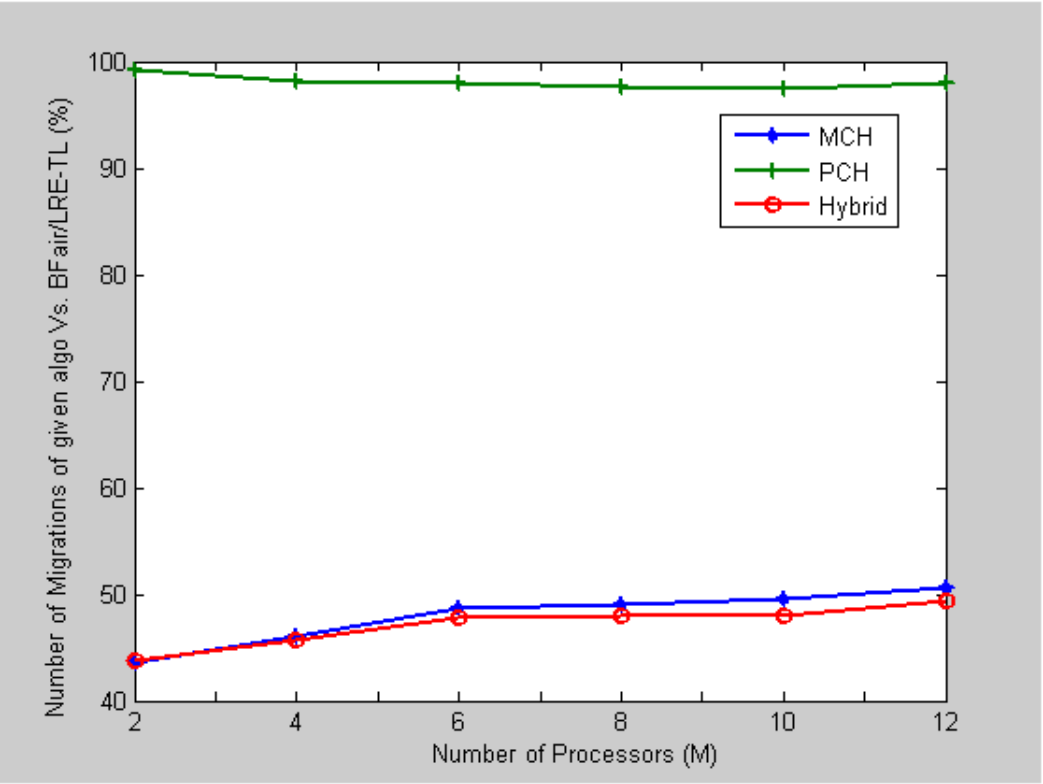


Figure 6.3: Migration control at U=0.5M

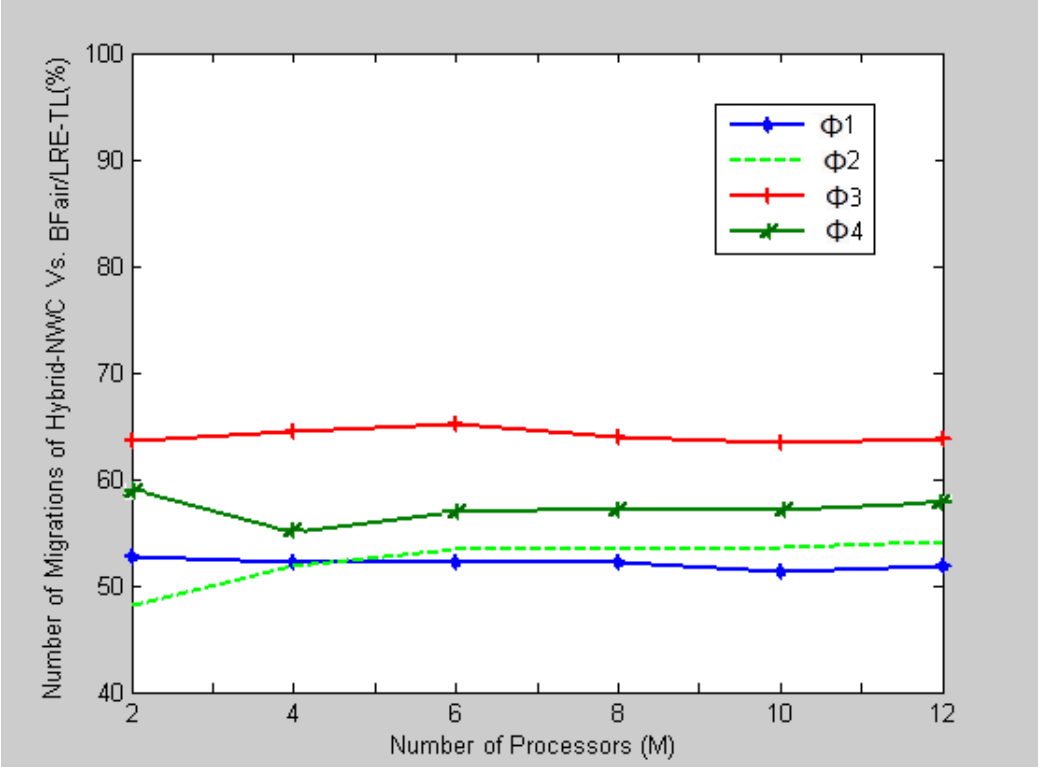


Figure 6.4: Migration control of hybrid-NWC of different period sets at U=M

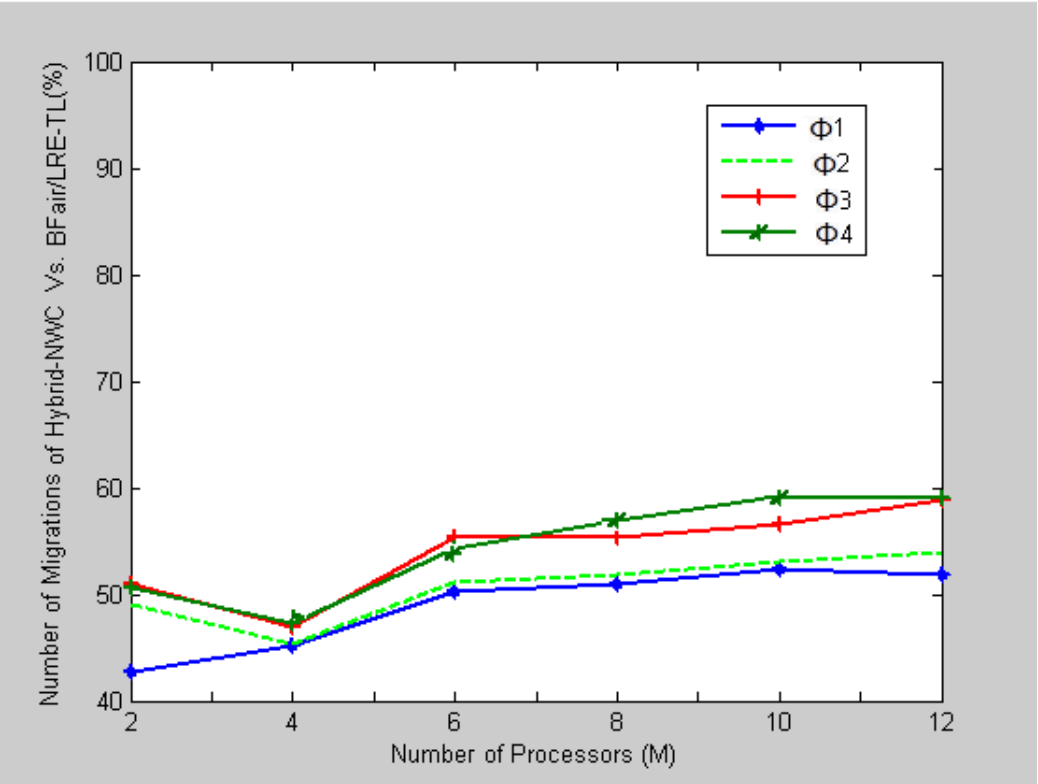


Figure 6.5: Migration control of hybrid-NWC of different period sets at U=0.75M

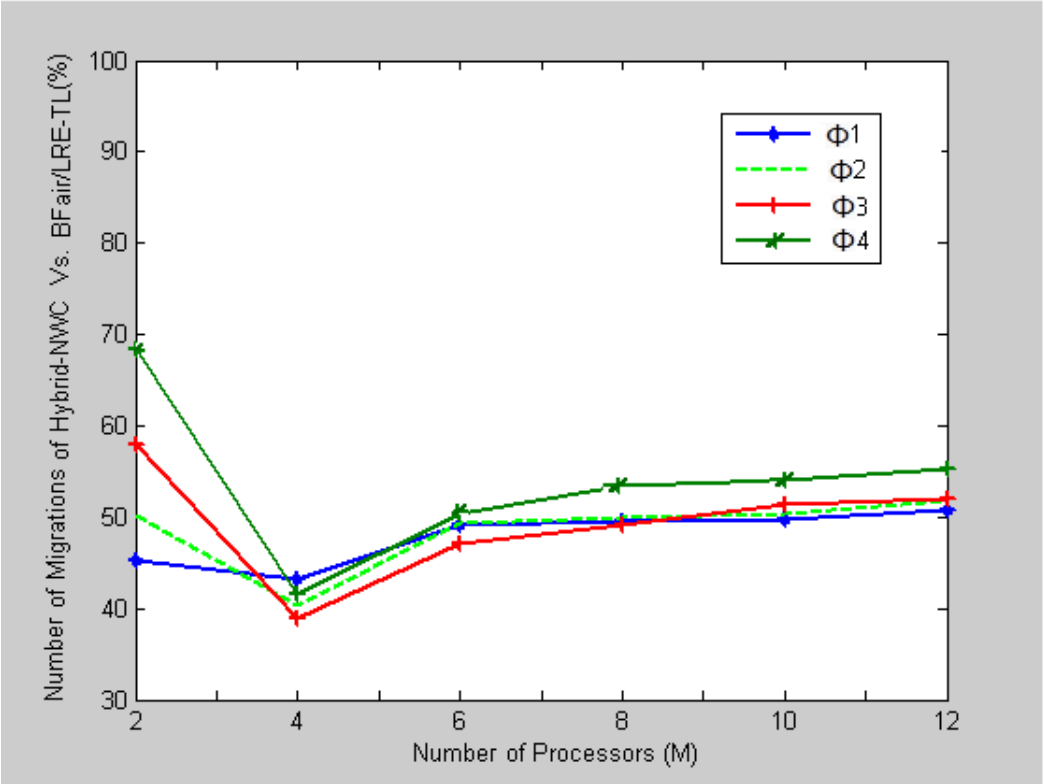


Figure 6.6: Migration control of hybrid-NWC of different period sets at U=0.5M

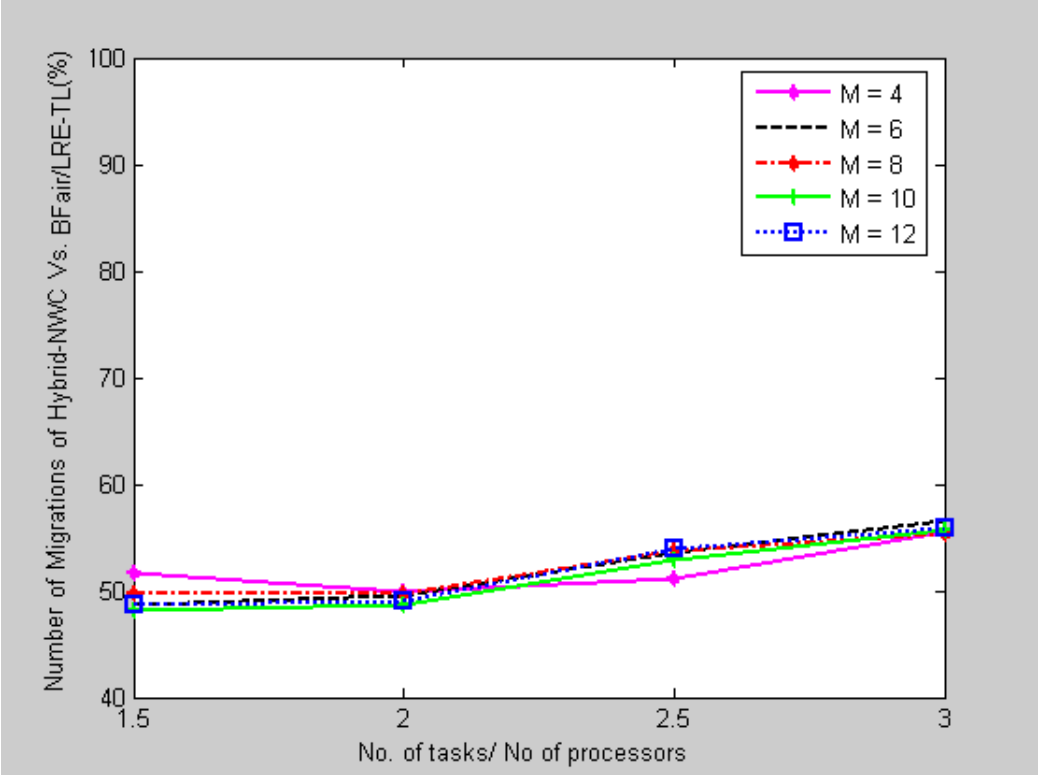


Figure 6.7: Migration control of hybrid-NWC varying N/M at U=M

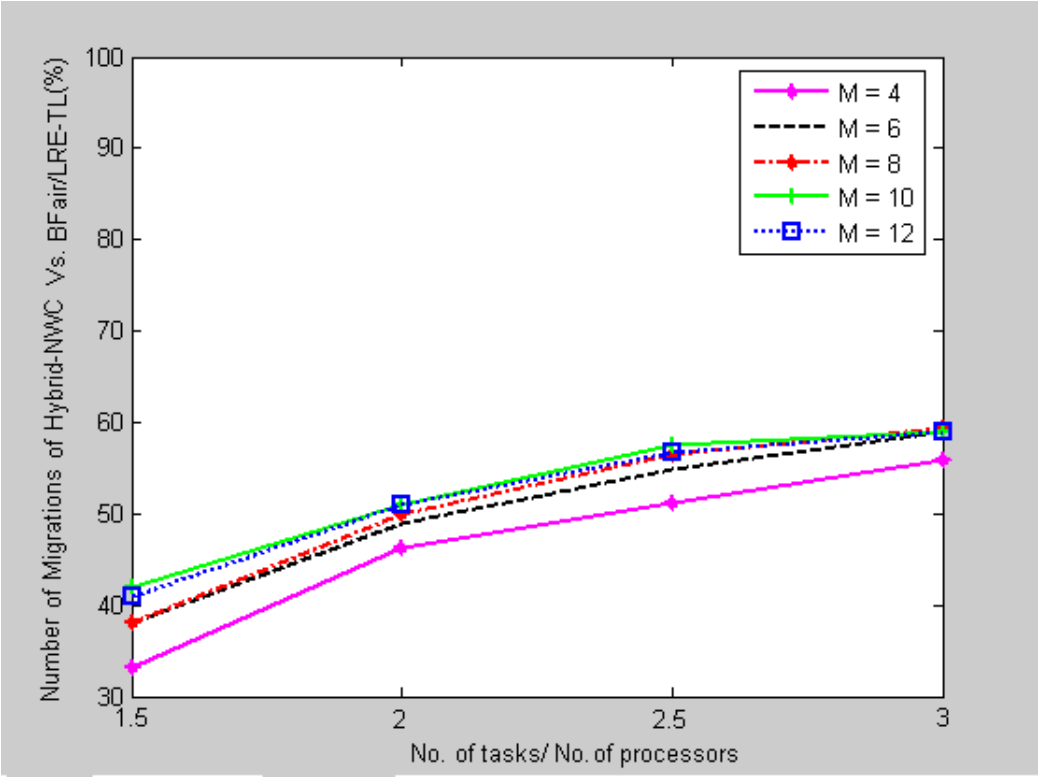


Figure 6.8: Migration control of hybrid-NWC varying N/M at U = 0.75M

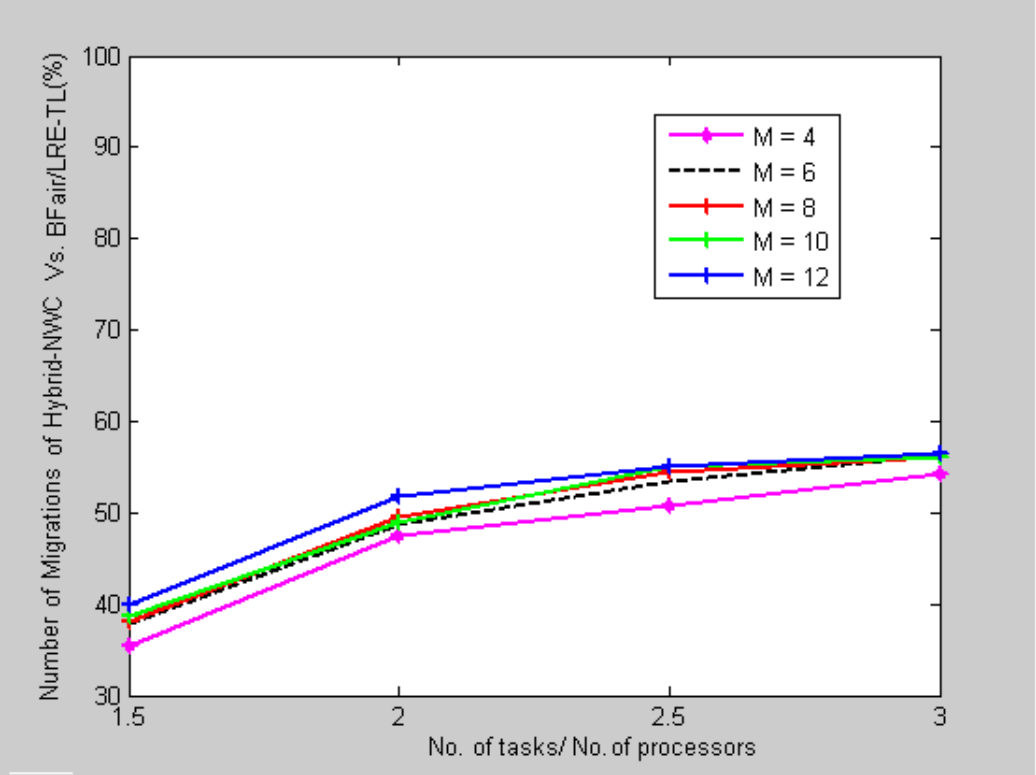


Figure 6.9: Migration control of hybrid-NWC varying N/M at U = 0.5M

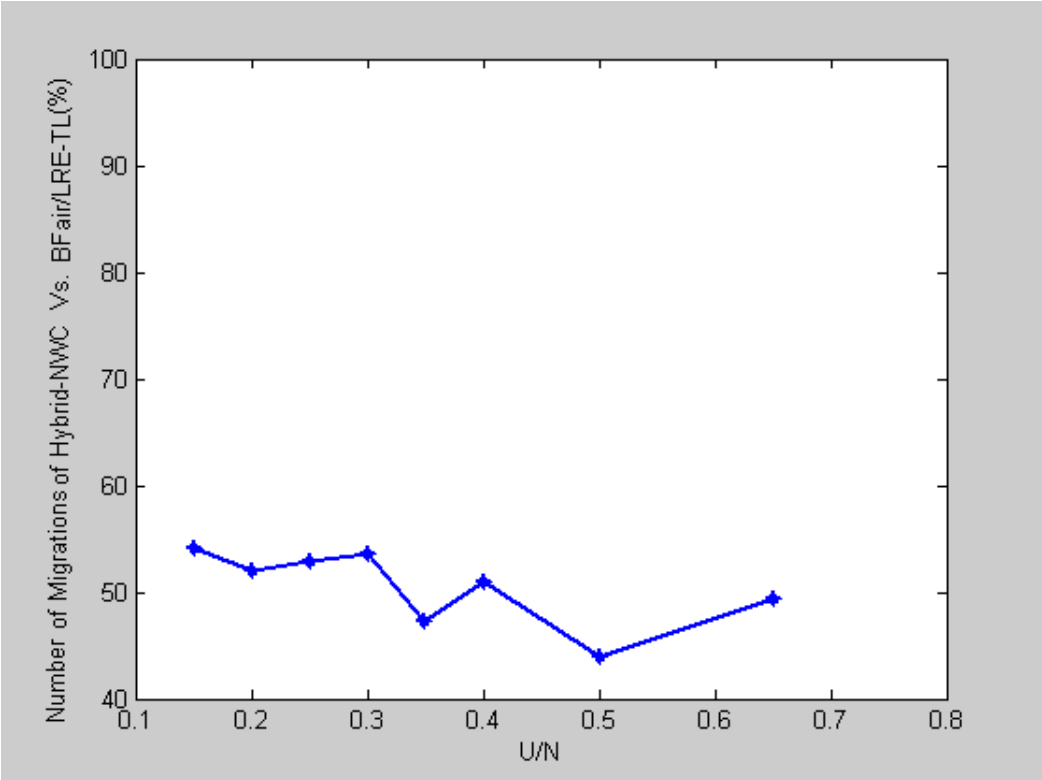


Figure 6.10: Migration control of hybrid-NWC at variable U/N

6.1.2 Preemption control (NWC)

6.1.2.1 Different algorithms at U=M, U=0.75M, U=0.5M

The results given in figures 6.11, 6.12 and 6.13. show the preemption control of different algorithms at *U=M*, *U=0.75M* and *U=0.5M* respectively for ϕ_1. The horizontal axis represents the number of processors while the vertical axis shows the percentage values of number of preemptions of given algorithm using *BFair/LRE-TL* as reference. Each point on the graph represents an average result of experiments conducted on 120 tasksets. The configurations include tasks with different utilization factors, light as well as heavy task.

The results shows that *MCH* does not contribute anything in reducing the preemptions and this behavior continues for *U=M, U=0.75M* and *U=0.5M*. This is due to the fact that *MCH* does not perform any role in the selecting tasks from ready tasks for execution but is applied after the tasks are finalized for execution.

The preemption control of *PCH* at *U=M* is shown in figure 6.11. It is approximately 65 % of the preemptions of *BFair/LRE-TL*.

At *U=0.75M*, the preemption control of *PCH* is less efficient than at *U=M* and it avoids 5 to 7 % of preemptions. This is because the *PCH* avoids a preemption for a task only if the task continues upto the end of the node. At relatively smaller value of total utilization, the chances of task execution upto the end of the node decrease.

The preemption control of *PCH* is negligible at *U=0.5M*. It avoids 1 to 2 % of preemptions. This is because there are very few tasks which continue upto the end of the node and hence *PCH* has no effectiveness.

The *hybrid-NWC* exhibits the same behavior as that of *PCH* and this behavior continues for *U=M, U=0.75M* and *U=0.5M*.

6.1.2.2 Hybrid-NWC algorithm at different sets of taskperiod

The graph in the figures 6.14, 6.15 and 6.16 shows the preemption control of the *hybrid-NWC* algorithm for all the four sets of task periods at *U=M*, *U=0.75M* and *U=0.5M* respectively. Each point in the graph shows an average result of experiments conducted on 120 tasksets. The general trend remains steady for all the sets of task periods.

The figure 6.14 shows that at *U=M*, similar to that of migration control, the *hybrid-NWC* algorithm gives best preemption control for ϕ_1 where we have maximum varying lengths of scheduling intervals. The hybrid gives the least control of preemptions for ϕ_3. The same behavior can be seen at U=0.75 in figure 6.15.

At *U=0.5M,* the value of hybrid algorithms remains the same for different sets of task period is shown in figure 6.16.

6.1.2.3 Hybrid-NWC at variable task to processor ratio

Figures 6.17, 6.18 and 6.19 show the preemption control of hybrid algorithm for ϕ_1 while varying the task to processor ratio. Each point in the graph shows an average result of experiments conducted on 30 tasksets. Number of tasks *N* varies between 6 and 36 in relation with the number of processors. The values are very close to each other for all the results.

At *U=M*, the hybrid algorithm gives approximately 65 % of original number of preemptions as in *BFair/LREE-TL*. The results are steady but the at *N=2M* its relatively better as shown in the figure 6.17.

We know that the preemption control ability of the *hybrid-NWC* algorithm due to *PCH* is relatively weaker at lower value of total utilization factor. Also with higher values of task to processor ratios, the number of light tasks in the taskset increases. Both of these factors reduce the chances of tasks to continue up to the next node. It leaves the idle time units at the end of the node that prohibits the work of *PCH* which controls both migrations and preemptions. This is the reason of the deterioration of preemption control at *U=0.75M* with higher values of task to processor ratio as shown in figure 6.18. Since the *PCH* controls the migration as well, the migration control of hybrid also follows the same pattern specially when *U<M* as shown in figures 6.8 and 6.9.

At higher values of task to processor ratio, it is more difficult for a task to re-execute on the same processor than at lower value of task to processor ratio. At *U=0.5M*, the same trend continues but again preemption is negligible. Caution that variations on the scale are very small and it is stretched.

6.1.2.4 Hybrid-NWC algorithm varying U/N

The 6.20 shows the variation in the preemption control of *hybrid-NWC* varying the ratio of *U* to *N* of ϕ_1. As the ratio increases the preemption control increases. At the lowest value of i.e. *U/N=0.1*, the preemption control becomes approximately zero. This is because with heavy tasks (higher value *U* to *N*) there are more chances of a task to continue upto the end of a node in different to lighter tasks. Recall that a preemption control of *hybrid-NWC* is because of *PCH* which only effective for a task which continues upto the end of the node.

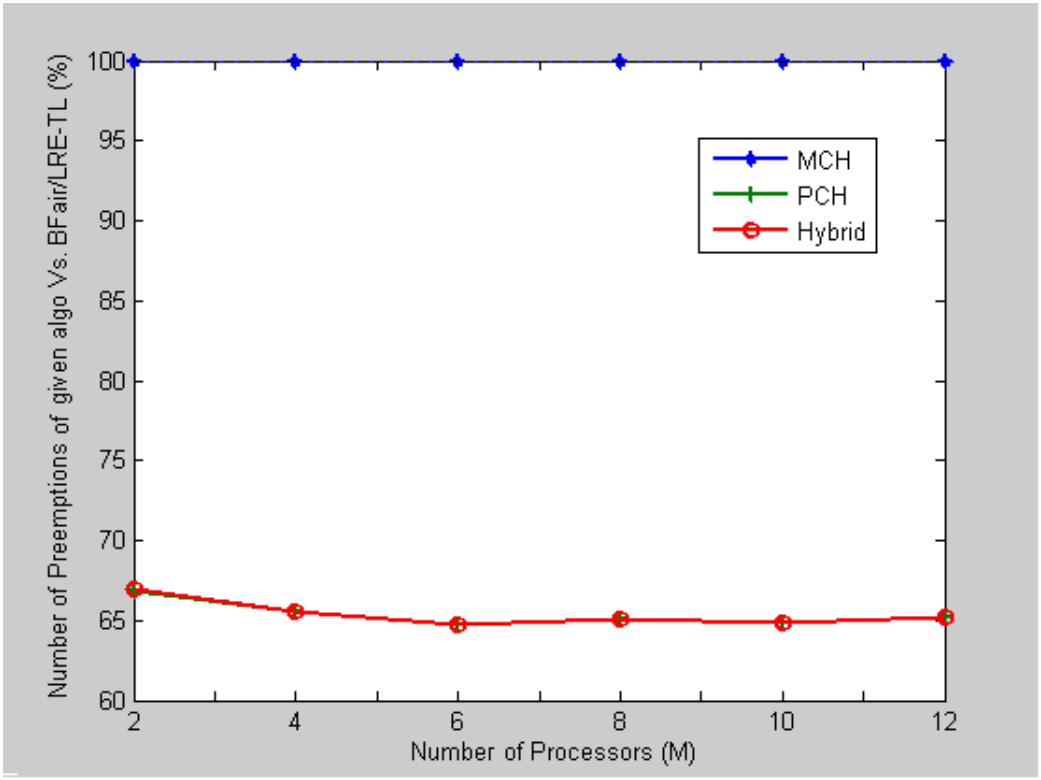


Figure 6.11: Preemption control at U=M

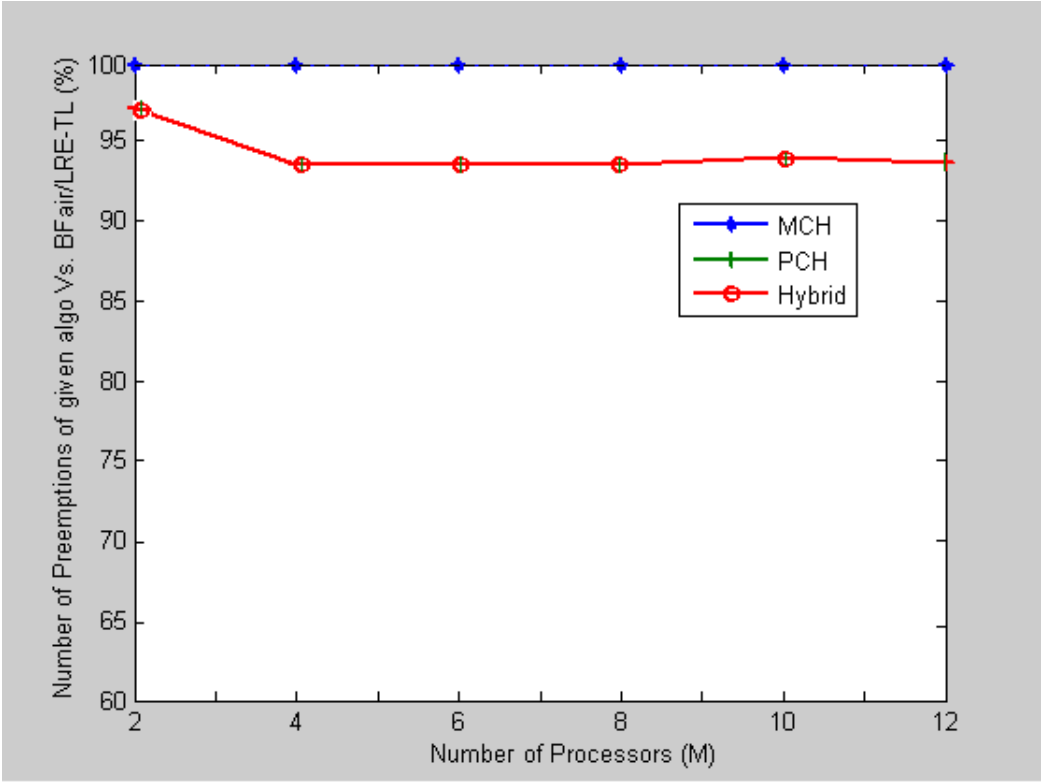


Figure 6.12: Preemption control at U=0.75M

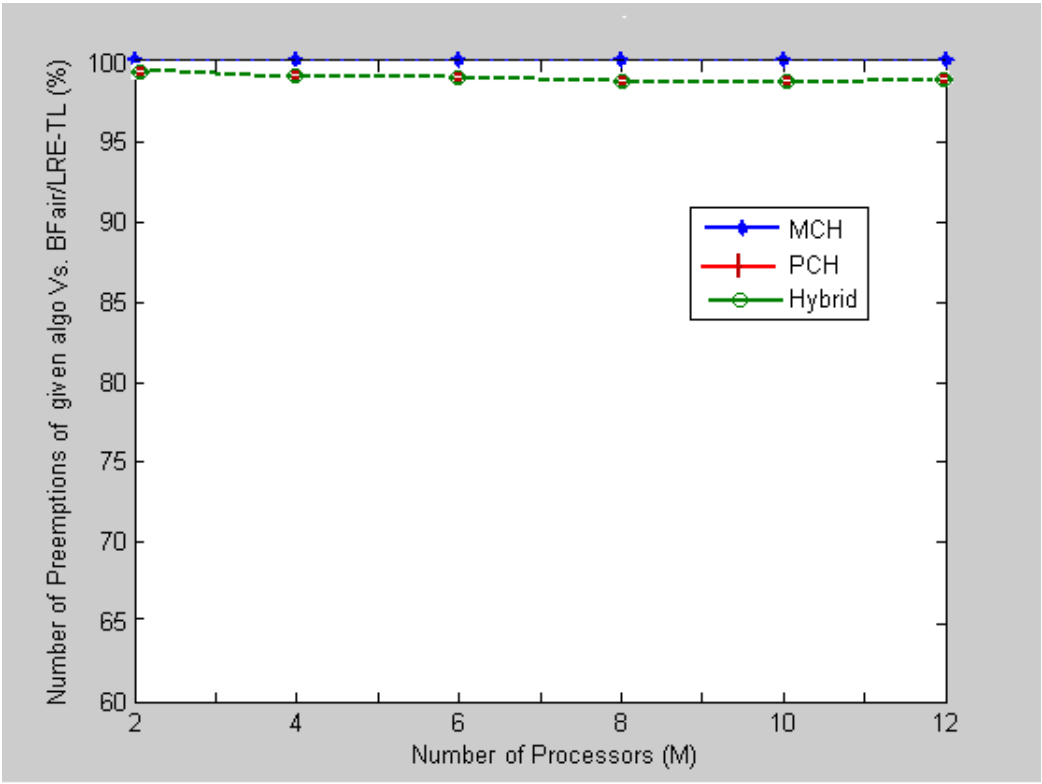


Figure 6.13: Preemption control at U=0.5M

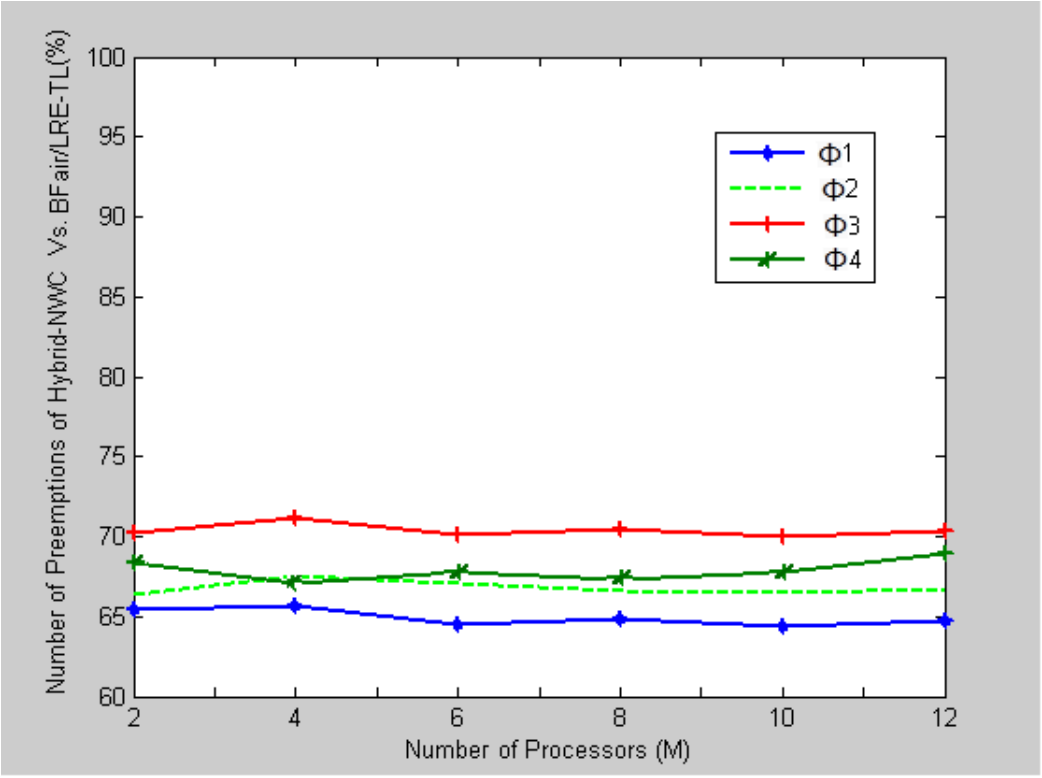


Figure 6.14: Preemption control of hybrid-NWC of different period sets at U=M

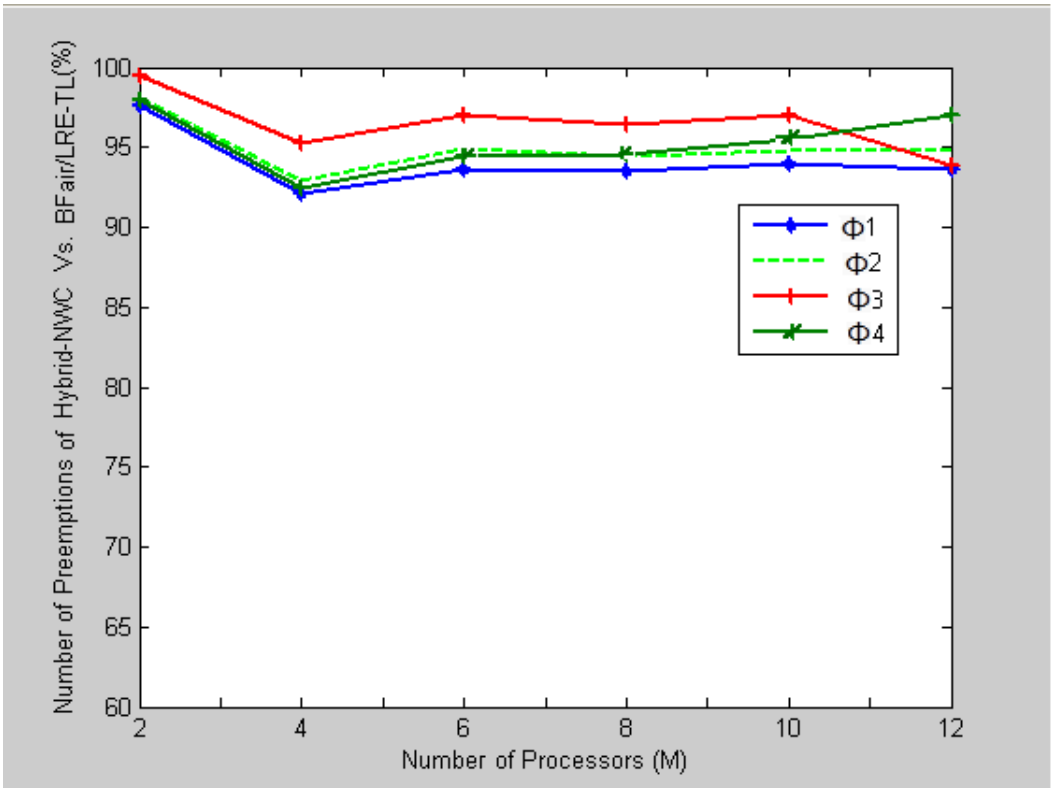


Figure 6.15: Preemption control of hybrid-NWC at different period sets at U=0.75M

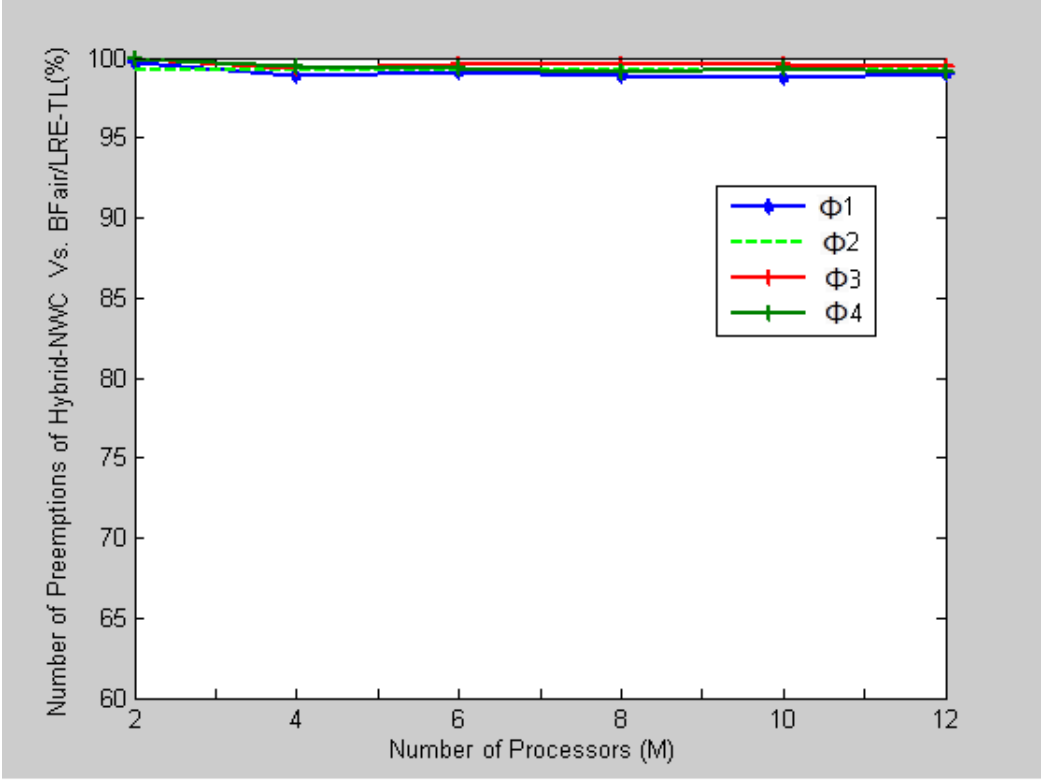


Figure 6.16: Preemption control of hybrid-NWC at different period sets at U=0.5M

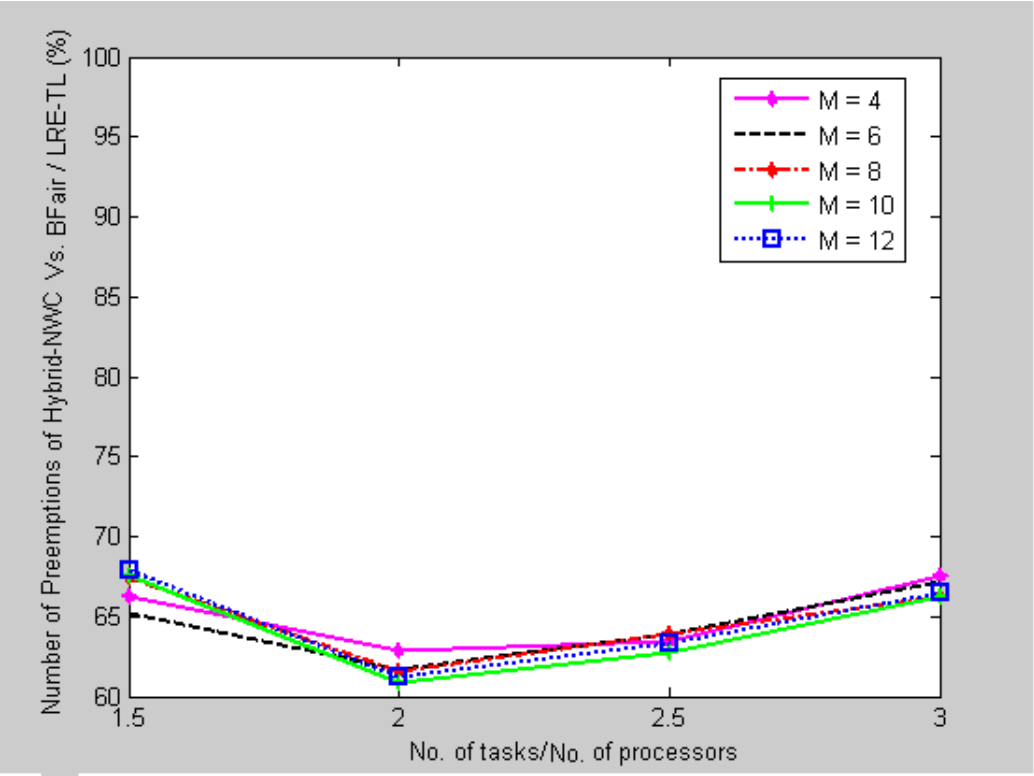


Figure 6.17: Preemption control of hybrid-NWC at variable N/M at U = M

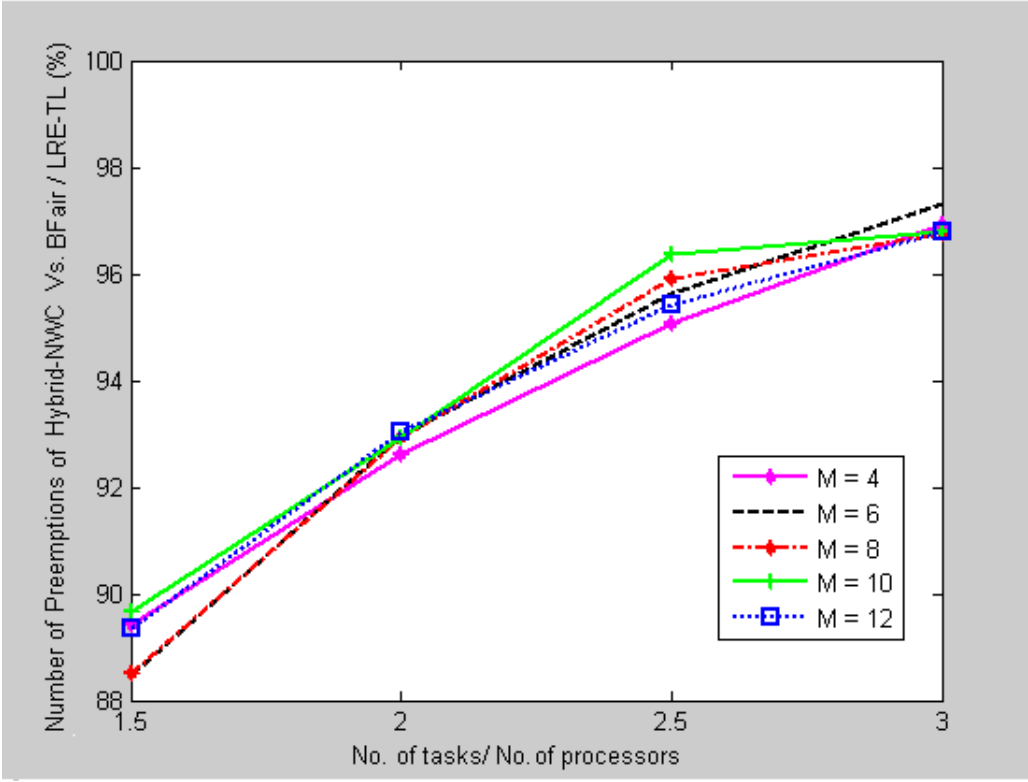


Figure 6.18: Preemption control of hybrid-NWC at at variable N/M at U = 0.75M

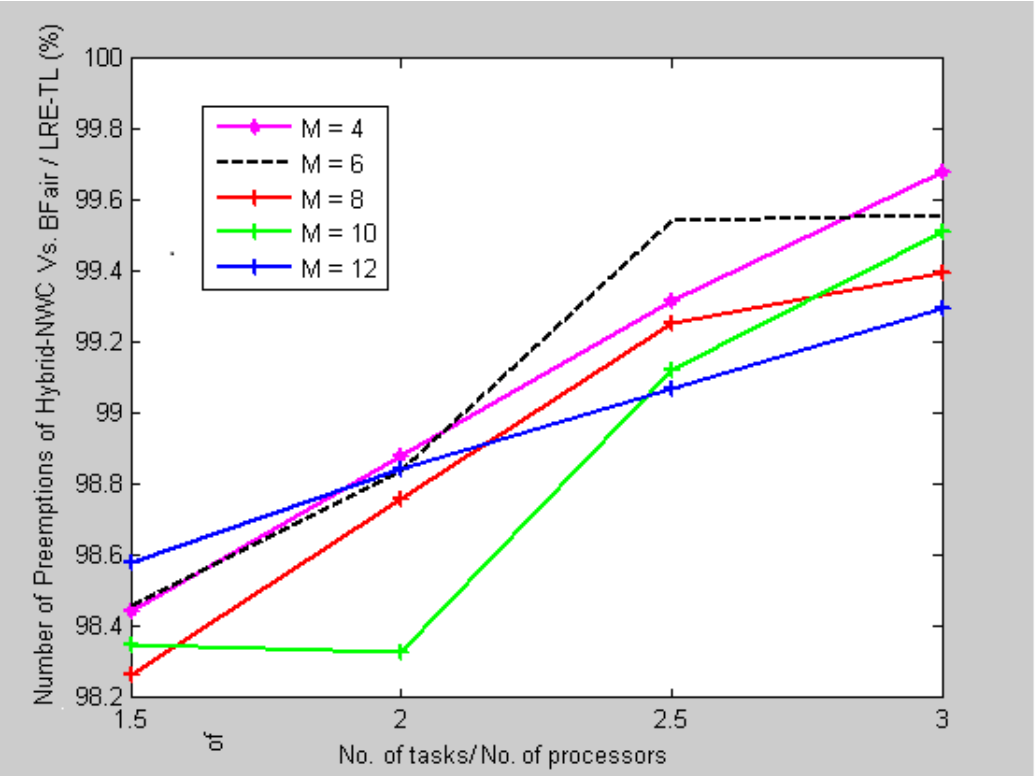


Figure 6.19: Preemption control at variable task to processor ratio U = 0.5M

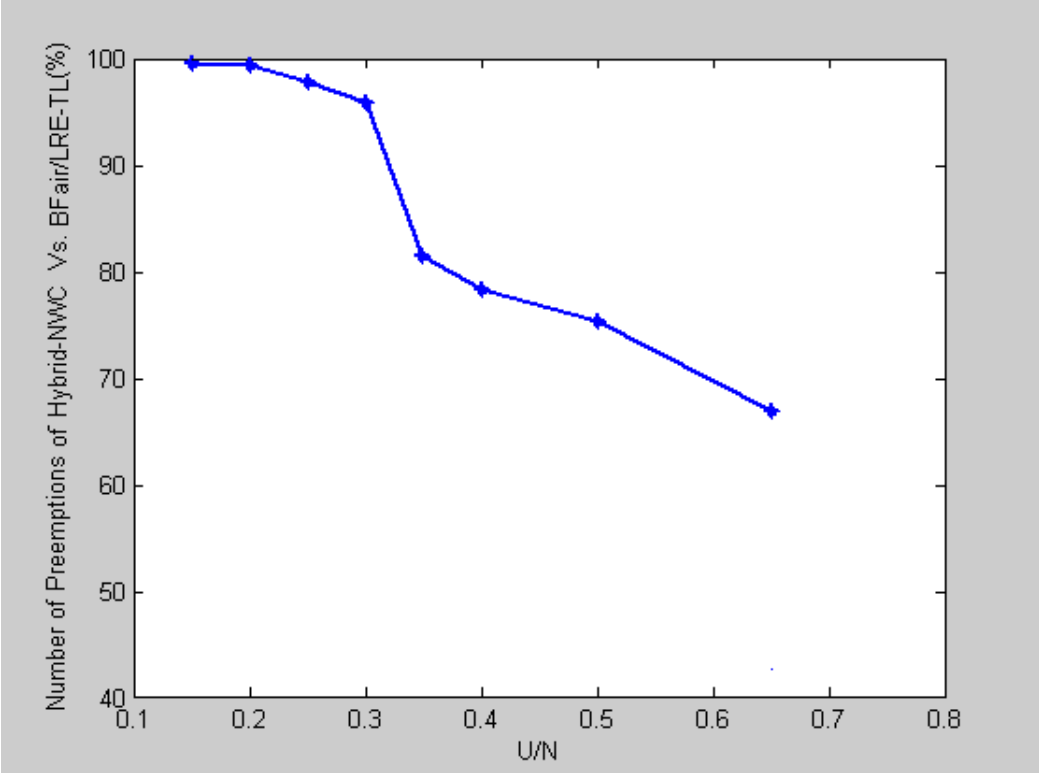


Figure 6.20: Preemption control of hybrid-NWC at variable U/N

M	Migration control %	Preemption control %
2	26.71	3.28
4	7.75	3.10
6	4.01	2.68
8	3.25	2.39
10	2.89	1.92
12	2.47	1.85

Table 6.1: Standard deviation of graphs in figure 6.1 and 6.11

6.1.3 Standard deviation

The *Standard Deviation* shows how much variation or "dispersion" exists from the average (mean). A low standard deviation indicates that the data points tend to be very close to the mean; high standard deviation indicates that the data points are spread out over a large range of values.

We have computed the *Standard Deviation* for all the experiments. The table 6.1 shows the values of standard deviation of *hybrid-NWC* in the figure 6.1 and the figure 6.11 as an example. The table shows very low values of standard deviation except the standard deviation of migration control at *M=2*. The small value shows the consistency in the results.

6.2 Work conserving case

As discussed earlier, there is no difference between non-work conserving and work conserving case at *U=M*. Therefore, we have performed the simulation process only at *U=0.75M* and *U=0.5M*. In work conserving context, we have used non-work conserving *BFair/LRE-TL* as a reference *DP-Fair* algorithm. The heuristics include *PCH (Preemption Control Heuristic)* and *MCH (Migration Control Heuristic).* There are three following algorithms involved in the whole process:

1. *BFair/LRE-TL (NWC)*
2. *BFair-NNLF/LRE-TL= NNLF*
3. *BFair-NNLF/LRE-TL+ PCH + MCH = Hybrid-WC*

For each of these algorithm, we count the number of preemptions and migrations expressed as percentages relative to the basic *BFair/LRE-TL* on vertical axis. In work conserving case, we have studied only *NNLF* and *Hybrid-WC* and then have compared the results with the results of *BFair/LRE-TL* already obtained during simulation of non-work conserving case for the same data. We have also introduced the *hybrid-NWC* for comparison in few graphs.

6.2.1 Migration control (WC)

6.2.1.1 Different algorithms at U=0.75M, U=0.5M

The figures 6.21 and 6.22 present the migration control of *NNLF, Hybrid-WC and Hybrid-NWC* in reference to the non-work conserving *BFair/LRE-TL* algorithm at *U=0.75M* and *U=0.5M* for ϕ_1. Each point in the graph shows an average result of experiments conducted on 120 tasksets. The work conserving has naturally fewer number of migrations because a processor does not remain idle if there is an active task for execution. It results in less preemptions and consequently in less migrations.

The figure *6.21* shows that *NNLF* has around 50% of preemptions of *BFair/LRE-TL* at *U=0.75M*. It is approximately same as that of *Hybrid-NWC*. After using the overhead control heuristics the *hybrid-WC* has approximately 24% of migrations of *BFair/LRE-TL* or half as that of *NNLF*.

The trend in migration control is same *at U=0.5M* as shown in figure *6.22* except there is an inversion of performance between *NNLF* and *hybrid-NWC*. It shows that *Hybrid-WC* shows approximately *15%* of the migration of *BFair/LRE-TL*.

6.2.1.2 Hybrid-WC algorithm at different sets of taskperiod

The figures 6.23 and 6.24 show the migration control of the *hybrid-WC* algorithm for all the four sets of task periods at *U=0.75M* and *U=0.5M* respectively. Each point in the graph shows an average result of experiments conducted on 120 tasksets. The general trend remains steady for all the sets of task periods.

The results show that *hybrid-WC* gives the best migration control for ϕ_1 and least for ϕ_3. The ϕ_1 has about 15% while ϕ_3 has about 35% of migrations of *BFair/LRE-TL*.

The same trend continues at *U=0.5M*. However, in this case the ϕ_1 has about 10% while ϕ_3 has about 25% of migrations of *BFair/LRE-TL* while the *hybrid-WC shows* approximately 40% of the migrations.

6.2.1.3 Hybrid-WC algorithm at variable task to processor ratio

Figures 6.25 and 6.26 show the migration control of *hybrid-WC* algorithm at variable the task to processor ratio. Each point in the graph shows an average result of experiments conducted on 30 tasksets. Number of tasks *N* varies between 6 and 36 in relation with the number of processors. The values are very close to each other for all the results.

The trend of slight decrease in the performance with an increase in ratio of task to processor remains here as was seen in the non-work conserving case.

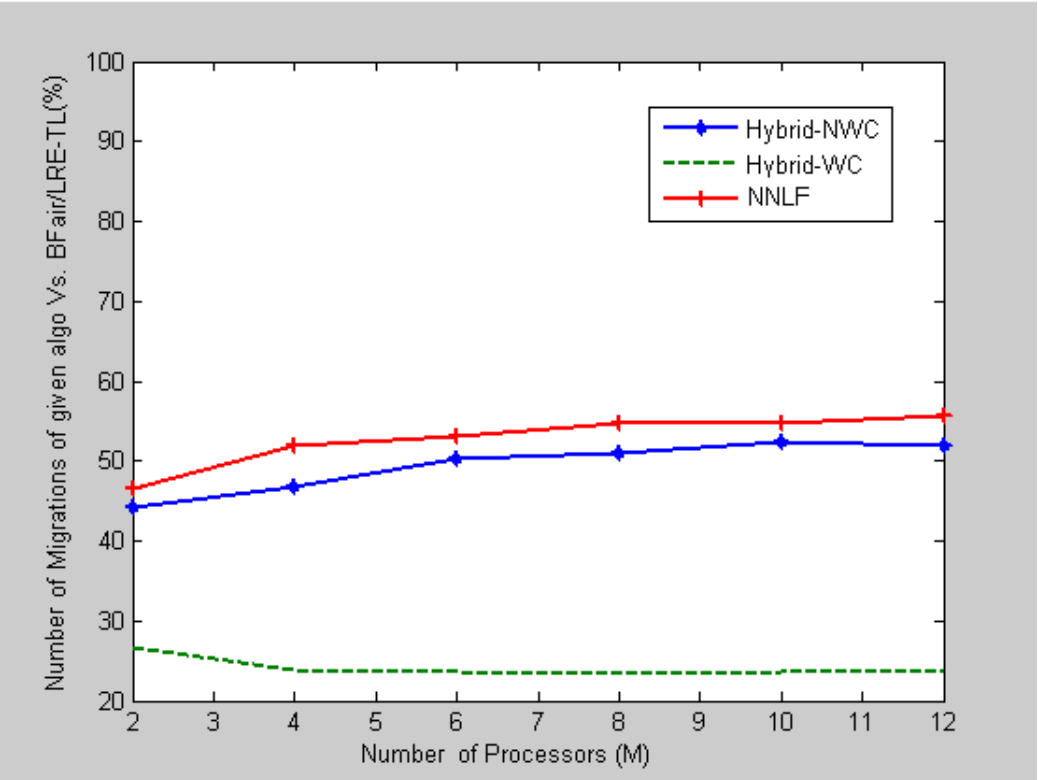


Figure 6.21: Migration control at U=0.75M

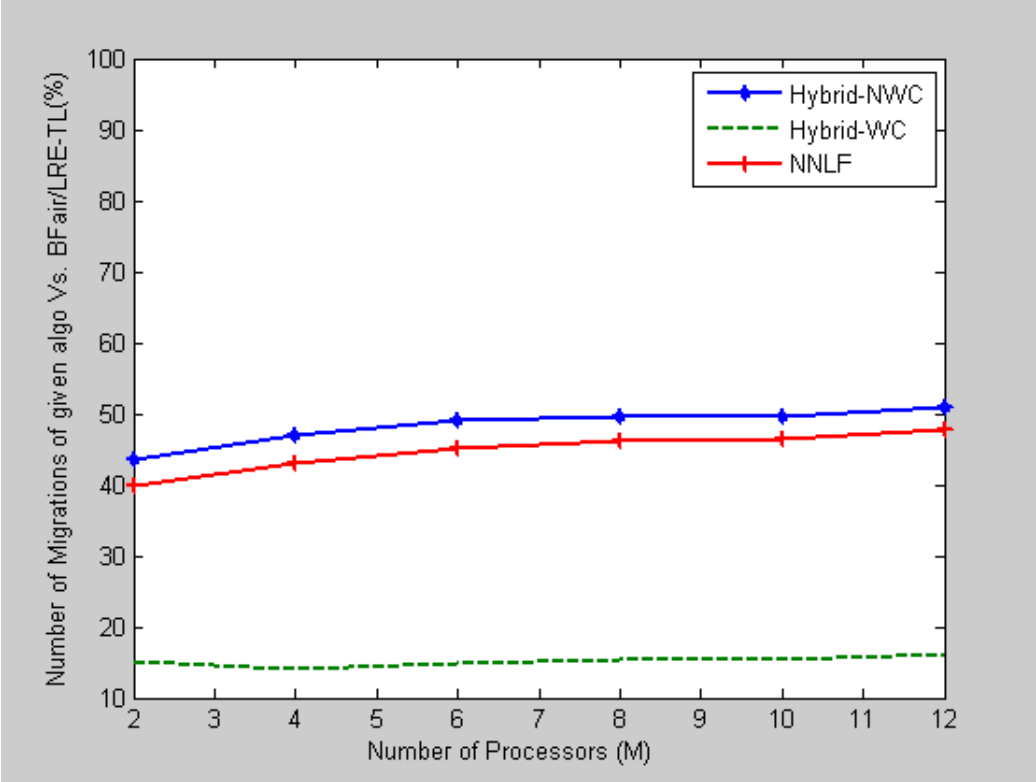


Figure 6.22: Migration control at U=0.5M

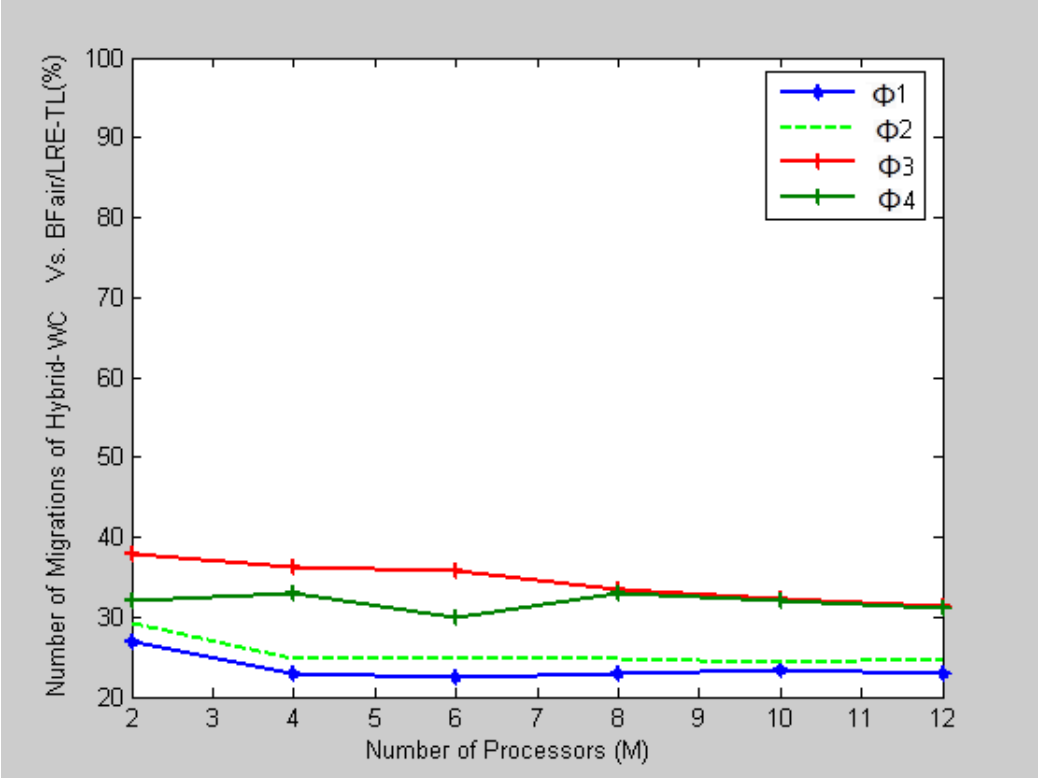


Figure 6.23: Migration control of hybrid-WC of different period sets at U=0.75M

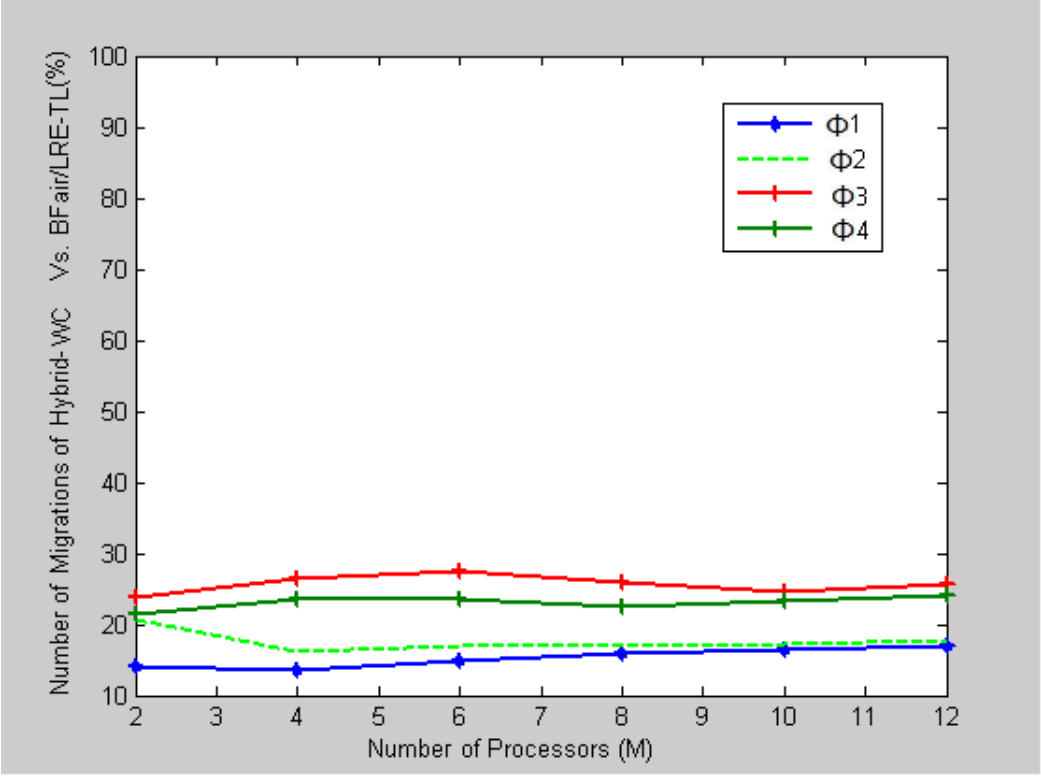


Figure 6.24: Migration control of hybrid-WC of different period sets at U=0.5M

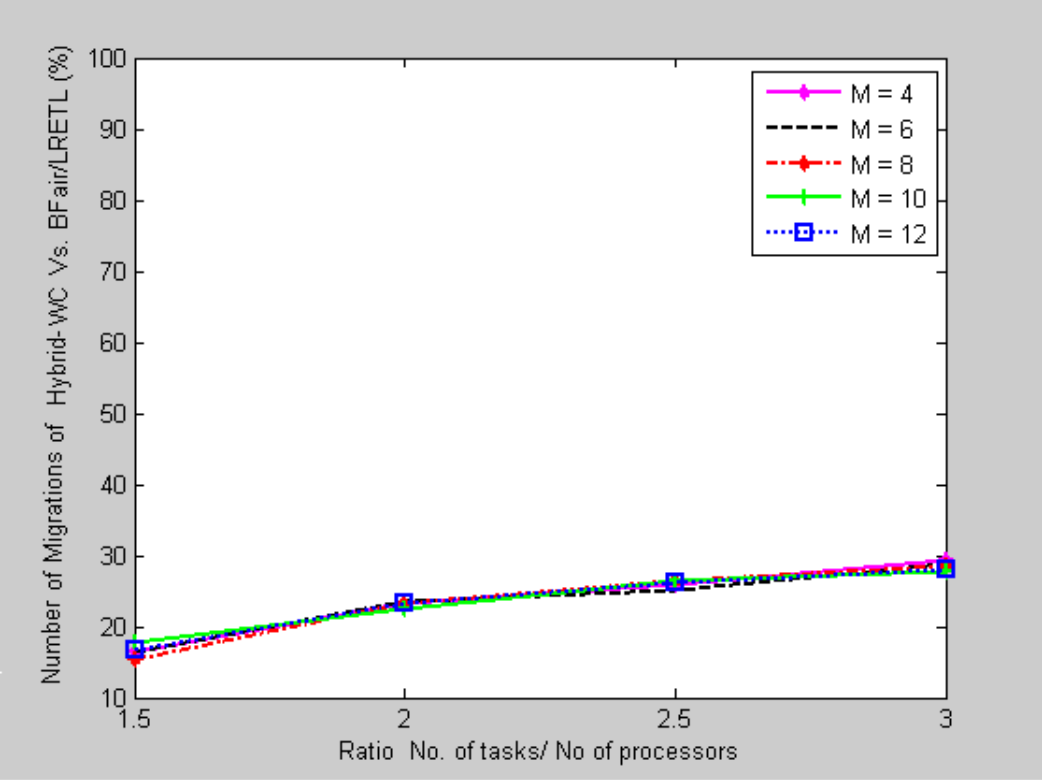


Figure 6.25: Migration control of hybrid-WC at variable N/M at U=0.75M

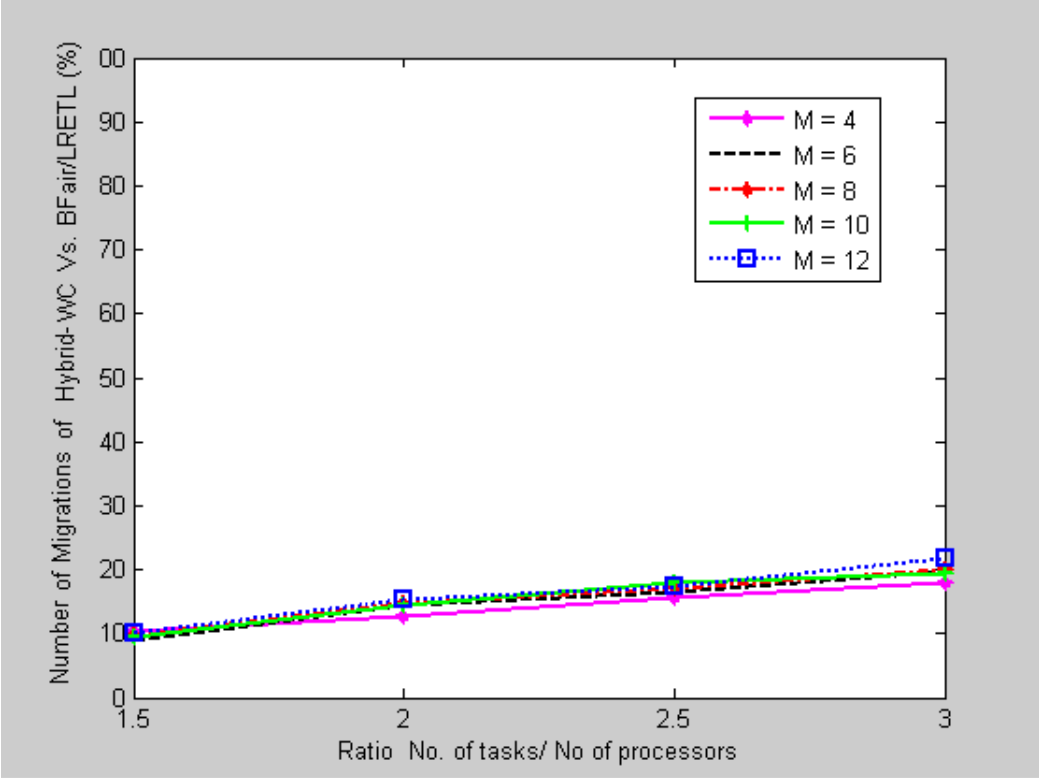


Figure 6.26: Migration control of hybrid-WC at variable N/M at U=0.5M

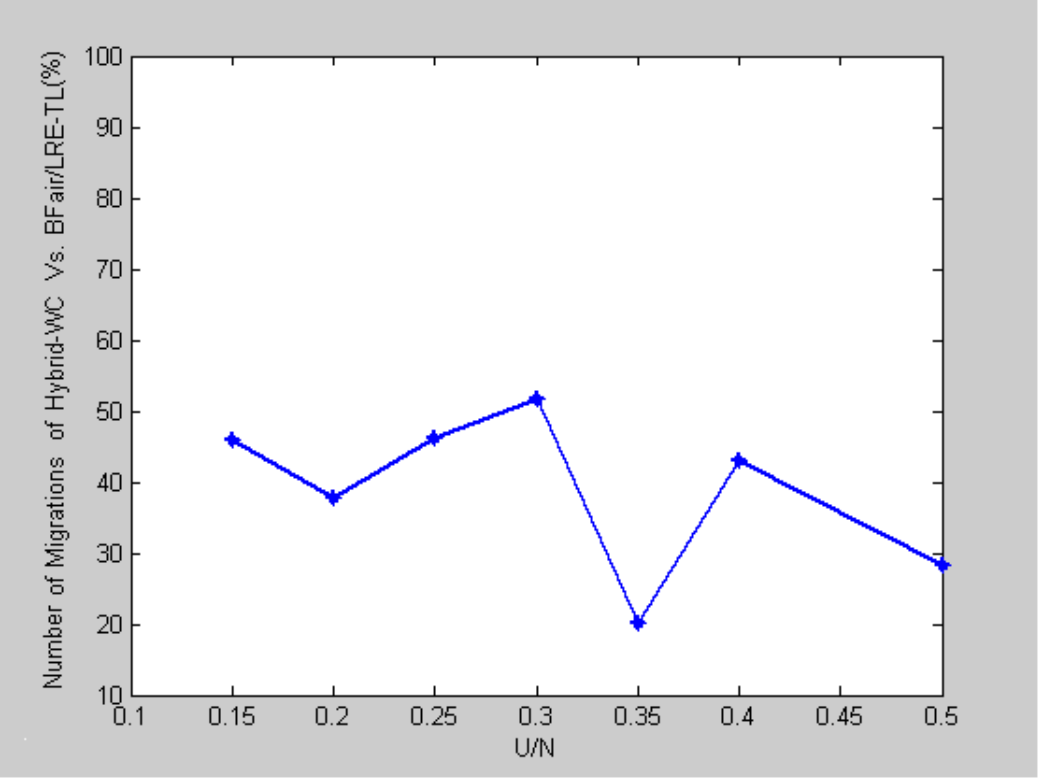


Figure 6.27: Migration control of hybrid-WC at variable U/N

6.2.1.4 Hybrid algorithm for U/N

The figures 6.27 shows the variation in the migration control of *hybrid-WC* varying the ratio of *U* to *N*. The graphs shows that most of the tasks are lighter one. It shows that there is no specific effect on migration control with this variation because some times the migration control increases and some times decreases with an increase in *U/N*.

6.2.2 Preemption control (WC)

6.2.2.1 Different algorithms at U=0.75M, U=0.5M

The figure 6.28 and 6.29 present the preemption control of *hybrid-WC* and a non-work conserving algorithm as a percentage of number of migration of the non-work conserving *BFair/LRE-TL* algorithm at *U=0.75M* and *U=0.5M* for ϕ_1. Each point in the graph shows an average result of experiments conducted on 120 tasksets.

The figure 6.28 shows that the number of preemptions of *NNLF* is around *14% to 20%* of the preemptions of *BFair/LRE-TL* which is quite significant. The *hybrid-WC* algorithm gives 9 to 10% of preemptions of *BFair/LRE-TL*, so a very high reduction

compared to *BFair/LRE-TL.* We can see that the preemption control of non-work conserving is insignificant and it avoids only 5% of preemptions.

At *U=0.5M,* the *NNLF* shows about *4%* of the preemptions of *BFair/LRE-TL.* The *hybrid-WC* preemptions gives approximately *2%* of preemptions of *BFair/LRE-TL,* a very high reduction. The preemption control of *hybrid-NWC* has about 98 to 99% of preemptions of reference algorithm.

The reason of this huge reduction in preemptions in case of *hybrid-WC* is due to the fact that when a tasks starts at lower total utilization factor there is naturally lower number of preemptions as shown by *NNLF* in the figures 6.28 and 6.29. When the tasks continue their execution upto the end of node, *PCH* works and it further reduces the number of preemptions.

6.2.2.2 Hybrid-WC algorithm at different sets of taskperiod

The figures 6.30 and 6.31 show the preemption control of the *hybrid-WC* algorithm for all the four sets of task periods at *U=0.75M* and *U=0.5M* respectively. Each point in the graph shows an average result of experiments conducted on 120 tasksets. The general trend remains steady for all the sets of task periods. All the values in the graphs are close to each other.

6.2.2.3 Hybrid-WC at variable task to processor ratio

Figures 6.32 and 6.33 show the preemption control of *hybrid-WC* algorithm while varying the task to processor ratio. Each point in the graph shows an average result of experiments conducted on 30 tasksets. The preemption control are very slightly better at lower value of task to processor ratio. This is because when there are fewer tasks the number of preemptions decreases due to lower chances of event *C*.

6.2.2.4 Hybrid-WC algorithm for U/N

The figure 6.34 shows the variation in the preemption control of *hybrid-WC* varying the ratio of *U* to *N* for ϕ_1. It shows that preemption control decreases with an increase in ratio of *U* to *N* except at *U/N=0.35*. This is because with at lower values of utilization factor, a task finishes earlier with less preemption as compare to the task with a higher utilization factor.

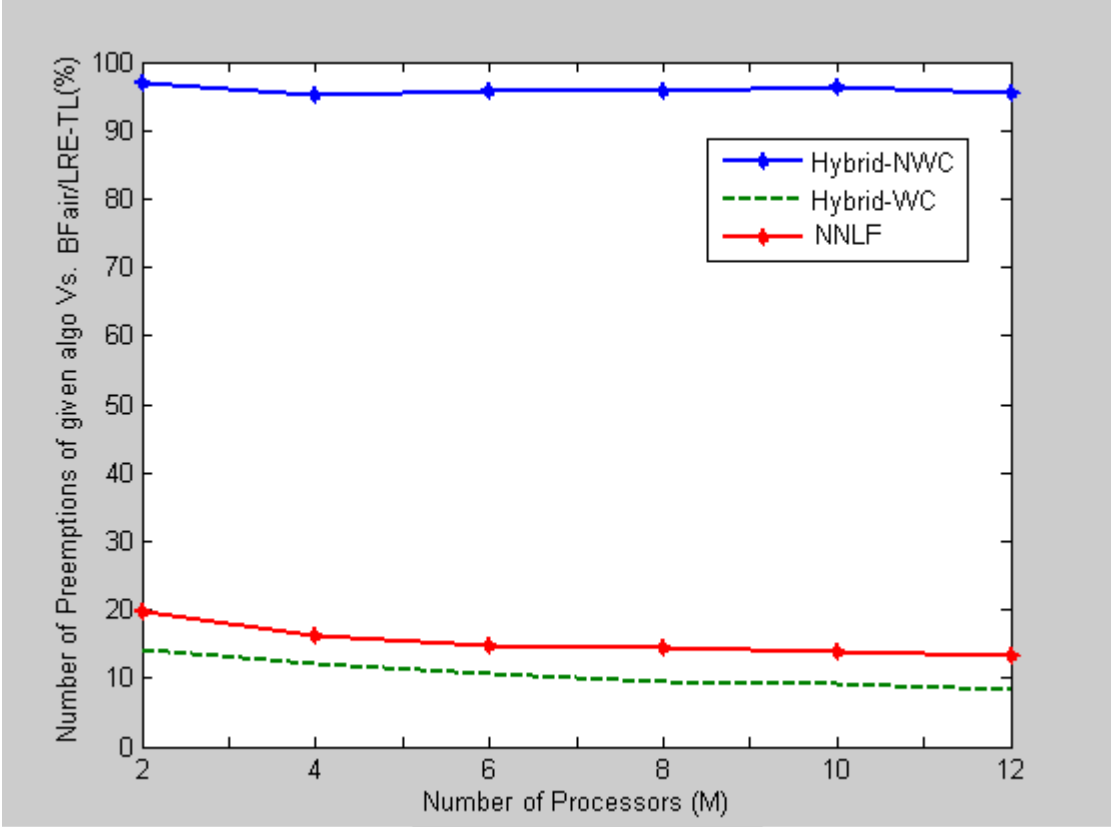


Figure 6.28: Preemption control at U=0.75M

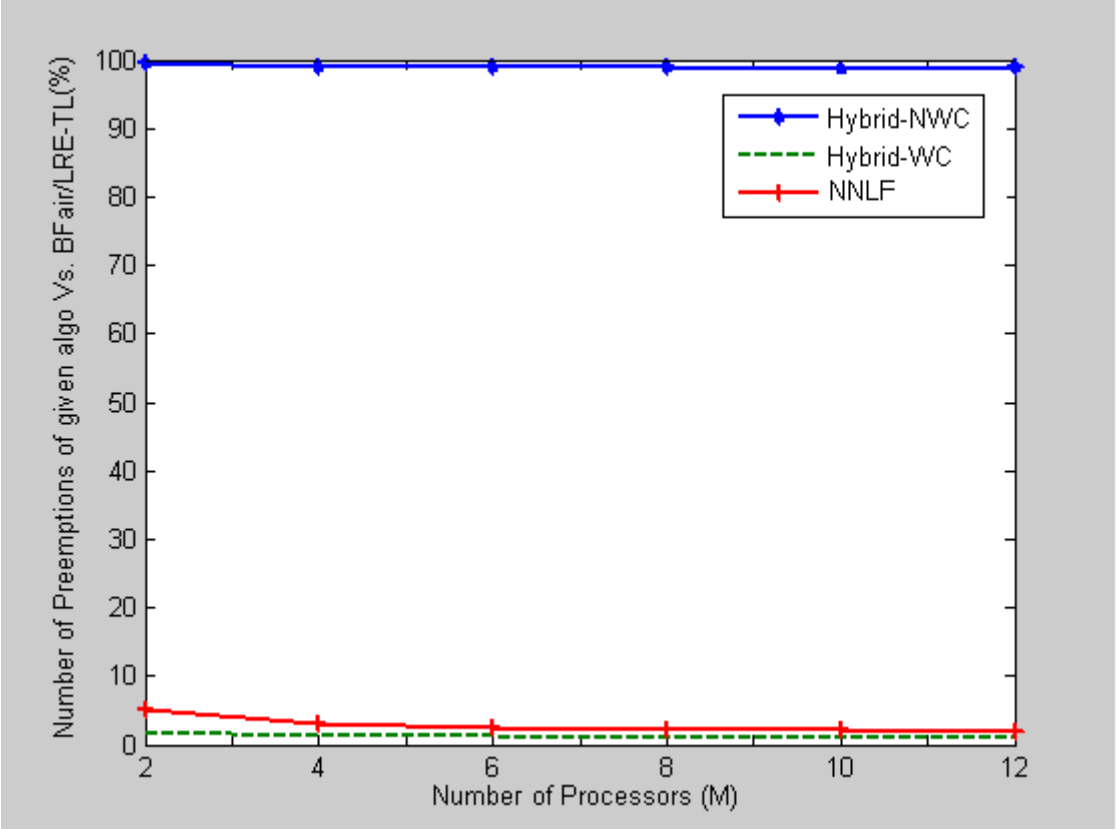


Figure 6.29: Preemption control at U=0.5M

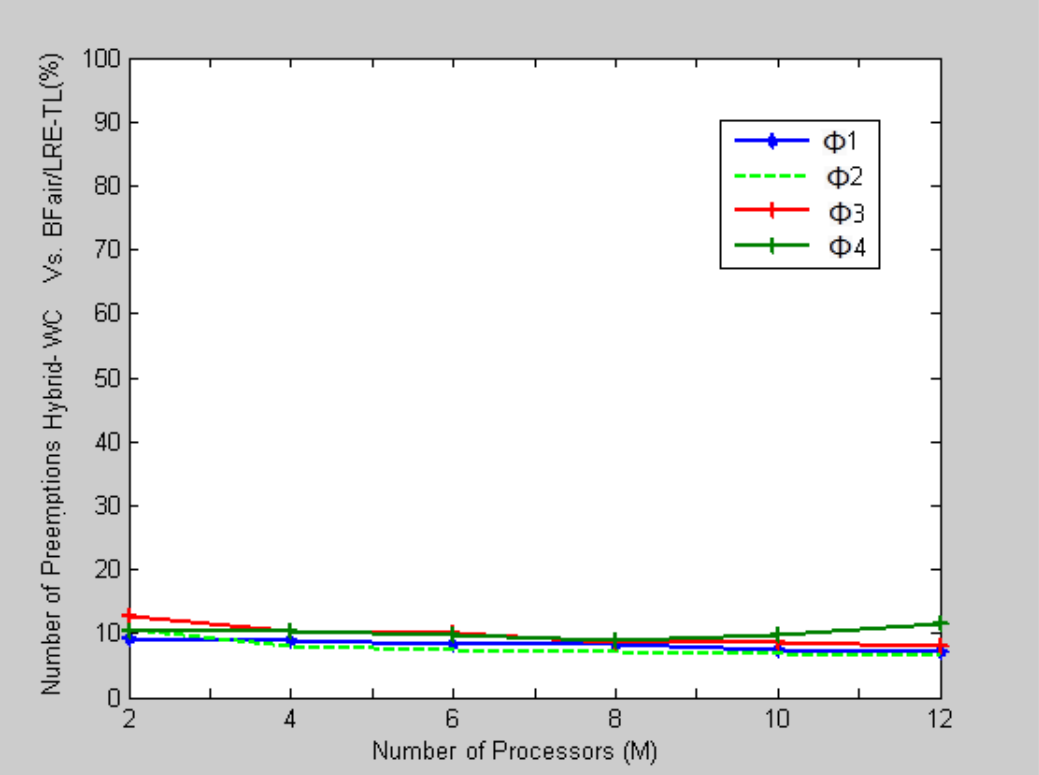


Figure 6.30: Preemption control of hybrid-WC of different period sets at U = 0.75M

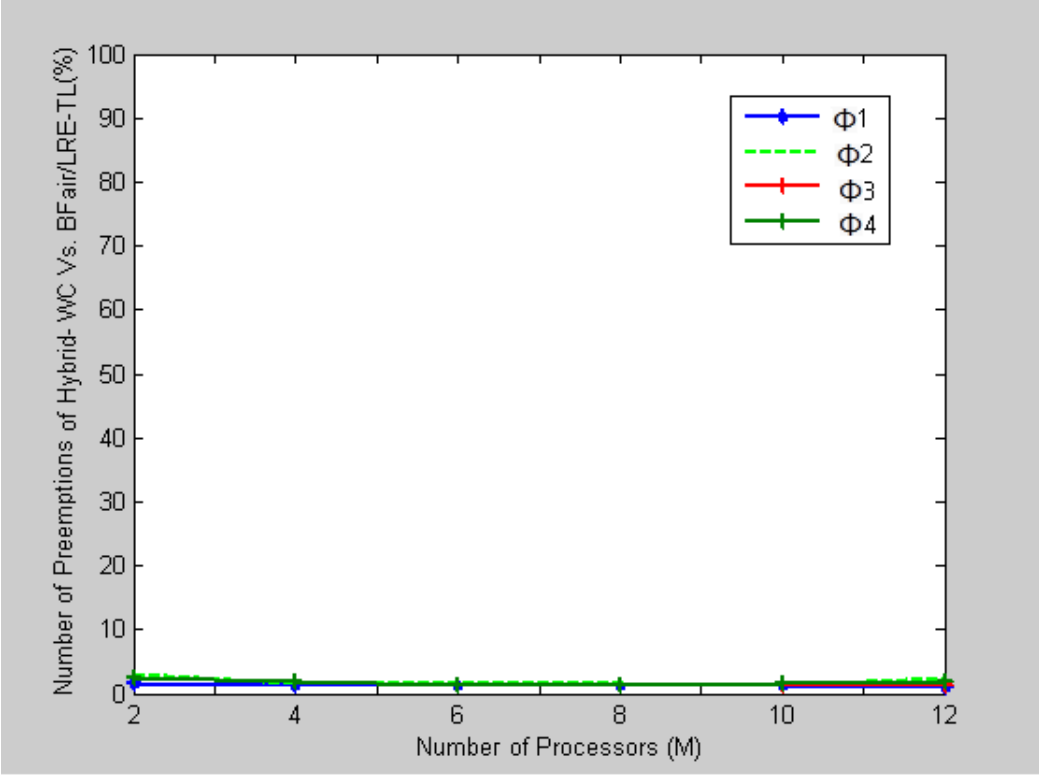


Figure 6.31: Preemption control of hybrid-WC of different period sets at U = 0.5M

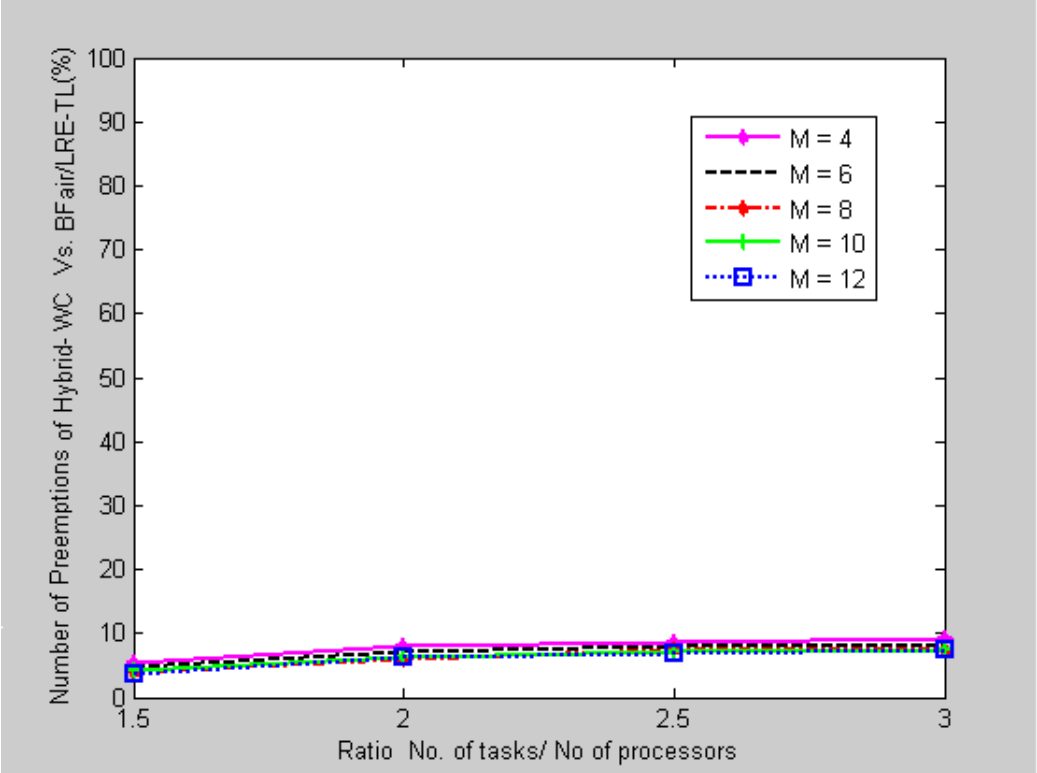


Figure 6.32: Preemption control at variable task to processor ratio at U = 0.75M

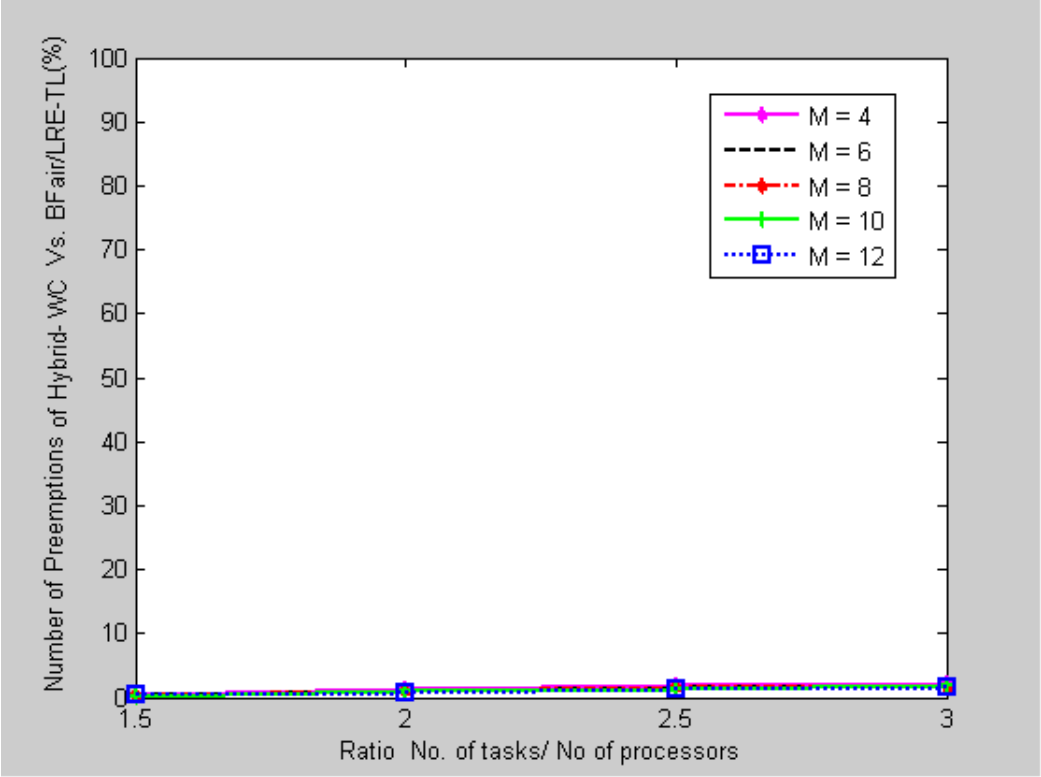


Figure 6.33: Preemption control at variable task to processor ratio at U = 0.5M

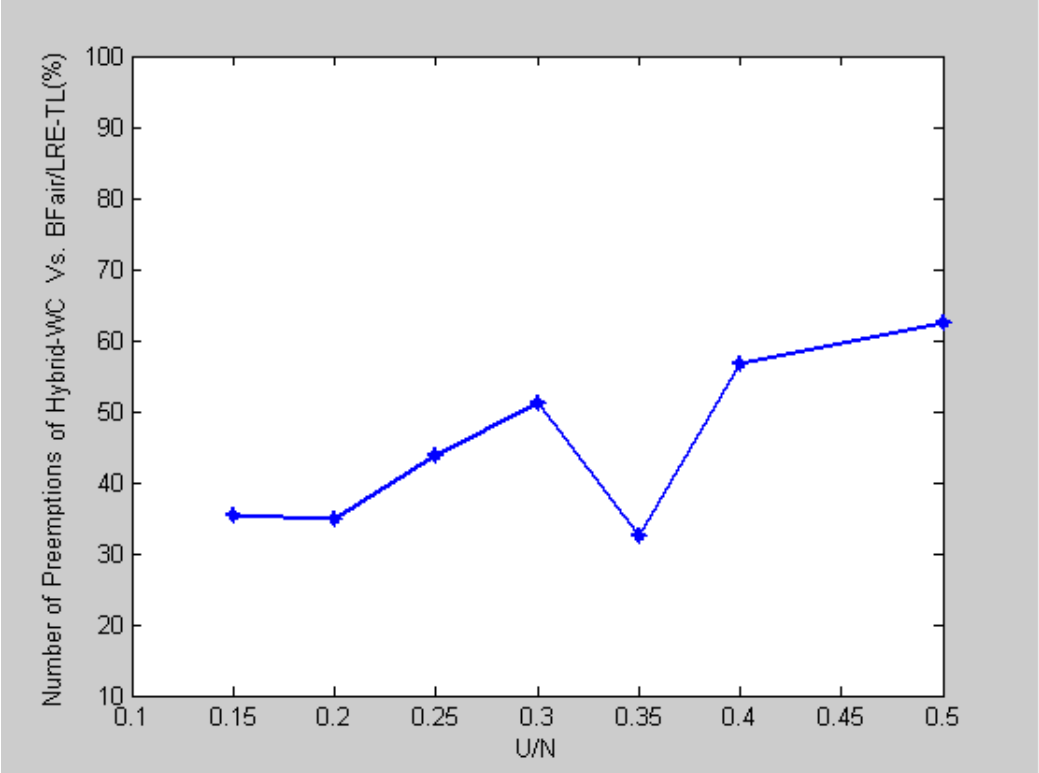


Figure 6.34: Preemption control of hybrid-WC for U/N

6.2.3 Standard deviation

The table 6.2 presents the values of *Standard Deviation* of hybrid-WC.

M	Migration control %	Preemption control %
2	11.91	8.24
4	5.99	11.06
6	3.80	9.23
8	3.49	8.79
10	2.88	7.52
12	2.70	7.26

Table 6.2: Standard deviation of hybrid-WC in figure 6.21 and figure 6.28

The values of standard deviation for migration control in the table corresponds graph of *hybrid-WC* in the figure 6.21 and the preemption control corresponds the graph of *hybrid-WC* in the figure 6.28. The table shows low values of standard deviation which shows the consistency in the results.

control of	Migrations			Preemptions		
U/M	1	0.75	0.5	1	0.75	0.5
Hybrid-NWC	52 %	47 %	47 %	65 %	94 %	98 %
Hybrid-WC	NA	24 %	15 %	NA	9 %	2 %

Table 6.3: Summary of overhead as percentages of overhead of BFair/LRE-TL

6.3 Conclusion

The results presented in this chapter are based on an extensive simulation process. The results include both non-work conserving and work conserving case and are presented with variation of the total utilization factor, number of task to processor ratio and U to N ratio. The results show significant control of overhead due to both migrations and preemptions. The hybrid algorithms are always better than the individual heuristics *PCH* and *MCH* at each point. This behavior of the hybrid algorithm is very logical and can be explained with reasons. The summary of main results are given in the table 6.3. It shows that *hybrid-NWC* decreases the migrations between 48% to 53% and preemptions between 2% to 35% of the corresponding value of *BFair/LRE-TL* at different values of total utilization factor. The *hybrid-WC* decreases the migration between 78 to 85% and preemption between 91% to 98% of corresponding value of *BFair/LRE-TL* at different values of total utilization factor.

7

General conclusion

The problem of the thesis was:

"*Overhead control in optimal global scheduling algorithms in real-time multiprocessor systems*".

It is important to know that the optimal global scheduling algorithms are not considered practical at the moment due to their unbearable overhead cost which is the main objection of its critics. The overhead is due to the frequent scheduling points, migrations and preemptions. This problem occurs because most of the optimal algorithms (with exception of recently proposed *RUN*) are based on the principle of fairness. The two main classes of optimal scheduling algorithms are *PFair* and *DP-Fair*. Both *PFair* and *DP-Fair* are interval based algorithms but *DP-Fair is* preferred because it has lower number of scheduling points and there is more space for overhead control in this class of algorithms.

In this work, a classification of *DP-Fair* algorithms has been proposed. All of the *DP-Fair* algorithms presented in the literature lie some where in this classification. Different *DP-Fair* algorithms are studied and then one of these algorithms has been selected on the bases of the following reasons:

1. There are no practical constrains in its implementation;

2. It is inherently more efficient than the other class of optimal global scheduling algorithms i.e. *PFair*;
3. There is a space of using overhead control in this algorithm.

We named this *DP-Fair* algorithm as *BFair/LRE-TL*.

As discussed above, a *DP-Fair* algorithm has lower overhead because it has lower number of scheduling points, therefore the next step to control the overhead was to reduce the number of migrations and preemptions. Some simple heuristics were chosen which avoid the possible migrations and preemptions without affecting the optimality of the of *DP-Fair* algorithm. These heuristics do not increase the complexity of the original scheduling algorithms.

Finally, a series of simulation based experiments were performed to find out and to verify the efficiency of our overhead control heuristics when used along with a *DP-Fair* technique. These experiments were performed using a large amount of well specified data which highlights the validity of the results.

Some of the general conclusions of the thesis are:

1. The classification of DP-Fair algorithm

In this work a classification of *DP-Fair* algorithms has been proposed. This classification is based on independent steps involved in accomplishing the scheduling process after the establishment of scheduling intervals in *DP-Fair*. This classification allows to use different combinations of techniques used for solving these two steps.

2. The overhead control heuristics efficiently reduce the number of preemptions and migrations

The results have shown that use of our heuristics reduced significantly the number of migrations and preemptions in the reference *DP-Fair* algorithm i.e. *BFair/LRE-TL*. The results are pretty consistent across different number of processors, different values of total utilization factor and for different sets of task periods. This is true for migration control and preemption control and both in case of non-work conserving as well as work conserving model.

In the non-work conserving case, the migration control is more significant in the sense that *hybrid-NWC* has about 50% of migrations of reference *BFair/LRE-TL* when the system is fully utilized, i.e. $U=M$. The migration control remains efficient as the value of total utilization factor is lowered to $U=0.75M$ and $U=0.5M$. On the other hand, although the preemption control works efficiently when the system is fully utilized. The *hybrid-NWC* has about 65% of preemptions at $U=M$. However its efficiency lowers with the lowering of system load.

In the work conserving case, inherently there are less number of migrations and preemptions but use of our heuristics reduces it further. The *hybrid-WC* significantly reduces the number of migrations and preemptions. The *hybrid-WC* has 22% of the migrations of reference *BFair/LRE-TL* at *U=0.75M* which reaches to 15% at *U=0.5M*. There is a change of behavior in preemption control of *hybrid-WC* in different to that of non-work conserving case. The *hybrid-WC* has about 9% of the preemptions of reference *BFair/LRE-TL* at *U=0.75M* which further reduces to 2% when *U=0.5M*. It remains effective even at lower value of total utilization factors because tasks are not preempted in work conserving model and continue to the end of the node.

3. The overhead control heuristics can be applied to all DP-Fair algorithms
The overhead control heuristics are general and can be used in all the *DP-Fair* algorithms. In addition, we have successfully utilized the *MCH* in migration control of *PFair*. This is equally applicable to other global algorithms which uses dynamic dispatching technique.

Extension of this work
Some of the possible extensions of this work are given in the following :

- It will be very interesting to implement a STORM version of *RUN* to compare the corresponding results with our results;
- To experiment the heuristics on a real multicore hardware;
- In our research team using Trampoline European source *RTOS*;
- To perform the experiments that can evaluate the real values of overheads discussed in chapter 4 (to estimate *CRPD* (Cache related Preemption Delays)). Once the measurements are done, it will be interesting to study their impact on schedulability.

Although our results are based on the simulation based experiments but it can be said that the two overhead control heuristics have resulted in a significant amount of reduction of overhead due to migrations and preemptions. If the research continues on the same track, it may further reduce the overhead. However, a high level of difficulties can be faced while doing the real implementation of optimal global algorithms we have studied.

French Summary

8

Introduction générale

Durant les vingt dernières années il y a eu une augmentation très importante du nombre de systèmes embarqués temps réel, et ceci dans de nombreux domaines : télécommunications, contrôle de production, équipements militaires, équipements médicaux et les transports, aussi bien aériens que terrestres.

L'augmentation constante des prestations offertes par ces systèmes nécessite une augmentation des performances des processeurs. Afin de satisfaire cette demande une première réponse a été l'augmentation de la vitesse des processeurs, mais des limites sont rapidement apparues à cause des difficultés à dissiper les puissances engendrées sur d'aussi petites surfaces. Une autre solution a alors été l'apparition des architectures multiprocesseur, d'abord au niveau des ordinateurs grand public puis ensuite dans les microcontrôleurs utilisés dans les systèmes embarqués.

A côté de ces progrès pour les architectures matérielles il y a aussi eu beaucoup de progrès sur les aspects logiciels. Dans le domaine qui nous intéresse les architectures matérielles sont gérées via un système d'exploitation, connu sous le nom de système d'exploitation temps réel (*RTOS*), lorsque les applications exécutées ont des caractéristiques temps réel. La conception de telles applications est généralement basée sur un ensemble de tâches périodiques, avec des exigences temporelles qui doivent être respectées. L'ordonnanceur, un des éléments du système d'exploitation, doit gérer les ressources, et notamment la ressource processeur, afin de respecter les contraintes de l'application (contrainte temporelle sur les tâches, contrainte énergétique sur le système ...). Les règles utilisées par l'ordonnanceur pour gérer les différentes contraintes con-

stituent une *politique d'ordonnancement*.

Les recherches sur l'ordonnancement ont débuté vers les années 50 lorsque des industriels se sont trouvés confrontés à la gestion efficace des ressources, notamment dans le domaine de la production. Plus tard avec l'émergence des technologies informatiques les travaux dans ce domaine ont connu un nouvel essor, notamment avec la publication du célèbre modèle de tâche de Liu et leyland en 1973, et les algorithmes *RM* et *EDF*. L'arrivée des architectures multiprocesseur homogènes ou hétérogènes a bien sûr contribué à un renouveau pour les travaux sur l'ordonnancement, avec malheureusement (ou heureusement pour la recherche !) le constat que les résultats acquis en ordonnancement monoprocesseur ne pouvaient se transposer dans le cas multiprocesseur. De plus à cette époque des résultats théoriques avaient conclus hâtivement (Liu et Dhall en 1978) à des difficultés spécifiques à l'ordonnancement multiprocesseur. C'est ainsi que l'ordonnancement multiprocesseur s'est principalement focalisé sur l'ordonnancement multiprocesseur partitionné, qui se ramène à l'ordonnancement de systèmes monoprocesseur, le partitionnement étant fait ; L'ordonnancement global n'a été abordé quant à lui que beaucoup plus tard.

L'ordonnancement global a été très étudié par les chercheurs vers la fin des années 90 et depuis l'effort n'a pas cessé, l'événement majeur étant la proposition par Sanjoy Baruah et ses collègues du célèbre algorithme *PFair* en 1996, algorithme optimal pour une configuration de tâches périodiques à échéances implicites. Etonnamment, et algorithme est basé sur le concept de *fairness*, au sens de la distribution équitable selon les besoins de chacun, alors que les algorithmes pour le temps réel ont toujours été conçus sur la notion de priorité, quel que soit la forme que prend celle-ci. Le principal reproche fait à *PFair* est d'engendrer de très fréquentes commutations de contexte, voire migrations et de ce fait il est considéré, tel quel, comme peu performant à l'implémentation. C'est pourquoi de nombreux chercheurs ont cherché à réduire ses trop nombreux points d'ordonnancement. Un modèle connu sous le nom de *Deadline Partitioning Fair (DP-Fair)* apporte effectivement beaucoup d'améliorations et c'est sur celui-ci que nous avons travaillé durant cette thèse.

Contributions

L'objectif de cette thèse était de proposer des techniques pour maîtriser les surcoûts à l'exécution dus aux préemptions et migrations, avec un ordonnancement optimal et global. Le modèle utilisé est celui des tâches périodiques avec échéances implicites pour une architecture matérielle multiprocesseur symétrique.

Nous avons proposé une classification des algorithmes fondés sur le modèle *DP-Fair*, classification basée sur la manière dont sont réalisées les deux étapes principales de ce modèle.

Le modèle n'imposant pas de politique quant à l'allocation des tâches prêtes aux processeurs , et l'ordre d'exécution des tâches dans un intervalle d'ordonnancement étant également libre, nous avons proposé des heuristiques avec l'objectif de réduire préemptions et migrations des tâches sans entacher l'ordonnançabilité et l'optimalité de l'algorithme de base utilisé.

Afin d'évaluer l'efficacité de nos heuristiques nous avons mené une étude statistique sur différentes métriques, en faisant varier les paramètres essentiels des configurations testées. Les résultats obtenus montrent en moyenne une très large réduction des préemptions et migrations, dans la version non-conservative et bien sûr encore plus dans la version conservative de l'algorithme.

Organisation de la thèse

La thèse est organisée en six chapitres.

Le chapitre 8 rappelle les bases théoriques des systèmes temps réel. Il présente les deux catégories d'ordonnancement multiprocesseur ainsi que les modèles utilisés, aussi bien logiciel que matériel.

Le chapitre 9 est un chapitre d'état de l'art. Il présente principalement les politiques d'ordonnancement global optimales *PFair* et *DP-Fair*, en proposant une classification des algorithmes dérivés. Un algorithme récent *RUN* est également évoqué.

Le chapitre 10 analyse les surcoûts et introduit les heuristiques que nous avons proposées pour la maîtrise de ces surcoûts.

Le chapitre 11 présente la manière dont l'étude expérimentale a été conduite, à partir d'un générateur de configurations et d'un simulateur d'ordonnancement.

Le chapitre 12 présente quant à lui les résultats complets de cette expérimentation.

Enfin le chapitre 13 tire les conclusions de ce travail et propose quelques pistes d'un travail futur.

9

Ordonnancement STR [1] multiprocesseur

1. STR=Systèmes Temps Réel

9.1 Introduction

Un système temps réel diffère des systèmes classiques car en plus de la contrainte classique de fournir des résultats justes, il doit également les fournir « à temps » : son comportement est assujetti à des contraintes temporelles. L'activité de base de tels systèmes, la tâche, peut avoir des contraintes sur sa date d'activation et/ou sa date de terminaison. Selon que le respect exigé des contraintes de temps est plus ou moins strict on parle de système temps réel dur (Hard Real-Time) ou de système temps réel mou (Soft Real-Time). Ceci correspond bien sûr à des domaines d'applications différents : calculateur ABS dans une automobile pour un exemple de temps réel dur, flux vidéo en streaming pour un exemple de temps réel mou.
Généralement on peut considérer un système temps réel comme construit autour de trois composantes :

1. le logiciel d'application : il est constitué des programmes qui implémentent les fonctionnalités requises. L'exécution de ces programmes sur un support d'exécution engendre les tâches gérées par le système d'exploitation temps réel ; elles sont caractérisées par des exigences temporelles ;
2. l'unité de traitement : elle représente les ressources physiques sur lesquelles s'exécutent les tâches applicatives. Selon le cas on peut disposer d'un unique processeur ou bien de plusieurs processeurs. La mémoire est en général hiérarchisée avec au moins un cache de premier niveau et une mémoire externe ;
3. la politique d'ordonnancement : elle est responsable, au travers de l'ordonnanceur, de l'exécution des tâches applicatives, de manière à respecter les contraintes temporelles des tâches, en définissant chaque fois que nécessaire quelle tâche doit s'exécuter et sur quel processeur dans le cas d'un système multiprocesseur.

Dans les paragraphes suivants nous détaillons chacune de ces composantes.

9.2 Le logiciel d'application

Le logiciel d'application est pris en compte au travers d'un modèle pour les tâches applicatives. Une tâche peut être périodique (répétée après un intervalle de temps fixe), sporadique (seul un intervalle minimum entre deux activations est connu) ou apériodique (pas de connaissance sur l'intervalle d'activation). Le modèle classique [44] considère que τ est un ensemble de N tâches temps réel périodiques $\tau = \{ T_i, i = 1, 2...N \}$. La tâche T_i est caractérisée par une date de réveil $T_i.o$, une période $T_i.p$, un pire temps d'exécution $T_i.e$ et une échéance $T_i.d$. Une exécution d'une telle tâche est appelée un *job* de la tâche. En ce qui concerne les échéances on peut considérer 3 cas. Pour les systèmes à *échéances arbitraires*, il n'y a pas de contrainte entre l'échéance

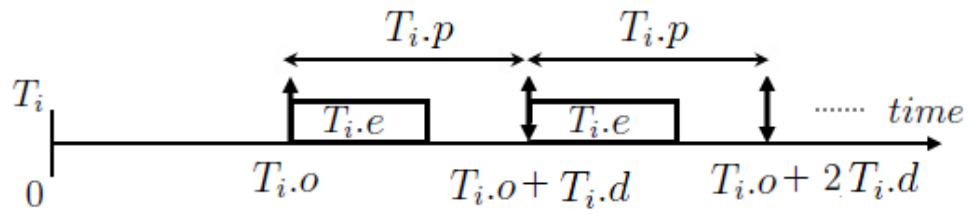


Figure 9.1: Paramètres d'une tâche

et la période, en particulier l'échéance peut être plus grande que la période. Pour les systèmes à *échéance implicite* l'échéance est égale à la période ($T_i.d = T_i.p$). Pour les systèmes à *échéances contraintes* l'échéance est toujours inférieure ou égale à la période ($T_i.d \leq T_i.p$). La configuration est dite *synchrone* si les offsets ($T_i.o$) de toutes les tâches sont égaux. Un exemple de tâche périodique T_i, est montré à la figure 9.1.

Le facteur d'utilisation $T_i.u$ d'une tâche T_i est défini par $T_i.e/T_i.p$. Le facteur d'utilisation total du système ,U , est la somme des facteurs d'utilisations des tâches de la configuration.

$$U = \sum_{i=1}^{N} (\frac{T_i.e}{T_i.p}) \tag{9.1}$$

On peut définir également le facteur d'utilisation du système comme le temps durant lequel les processeurs sont utilisés :

$$U_S = \frac{U}{M} \tag{9.2}$$

Le modèle que nous considérons est composé de tâches périodiques synchrones à échéances implicites. Les tâches sont indépendantes : elles ne partagent pas de ressources et n'ont pas de relations de précédence entre elles.

9.3 L'unité de traitement

Le processeur est l'unité de base du système de traitement : il est constitué d'un cœur de calcul sur une puce associé à de la mémoire et éventuellement des périphériques. Lorsqu'il y a plus d'un cœur sur une même puce on parle de processeur multicoeur. Si les processeurs sont dans le même boitier de circuit intégré (ICs), on parle de «System on Chip »(SoC). Si chaque processeur possède plus d'un coeur l'architecture est qualifiée de multiprocesseur multicoeur. Les architectures multiprocesseur ont les avantages suivants :

1. on obtient de meilleures performances avec plusieurs processeurs travaillant en parallèle (meilleur temps de réponse pour la fourniture des résultats). Cependant

l'exécution d'une même tâche sur plusieurs processeurs engendre des surcoûts système. Egalement le partage de ressources comme la mémoire engendre des conflits et finalement le gain escompté est plus faible ;

2. un système multiprocesseur permet de construire des systèmes plus fiables puisque l'on peut exploiter la redondance matérielle pour pallier à la défaillance d'un processeur ;

3. un système multiprocesseur est plus économique que de multiples systèmes monoprocesseur grâce au partage des périphériques, de la mémoire et des alimentations.

Selon la nature de l'architecture physique les systèmes multiprocesseur peuvent être classés en deux catégories : ceux à couplage faible et ceux à couplage fort. Dans un système multiprocesseur *faiblement couplé* les processeurs sont reliés par des liens de communication rapides comme Ethernet, chacun ayant sa propre mémoire et ses circuits d'entrée/sortie. Dans un système multiprocesseur *fortement couplé* les processeurs sont reliés à un bus commun (parallèle) et partagent de la mémoire commune.

Récemment des architectures avec un grand nombre de coeurs sont devenues disponibles, mais nous ne les considérons pas car elles n'ont pas d'application (pour le moment) dans le domaine des systèmes embarqués temps réel. Dans les travaux sur l'ordonnancement une classification est habituellement faite sur la base des similitudes et différences des processeurs. Ainsi on distingue les architectures : **1 - Homogènes :** dans ce modèle tous les processeurs sont identiques et partagent la même mémoire principale. Chaque tâche peut être exécutée sur n'importe quel processeur. Ce modèle est aussi connu sous le vocable *Symmetric MultiProcessor* (SMP). Un exemple d'une telle architecture est donné dans la figure 9.2. **2 - Uniformes :** dans ce modèle les processeurs peuvent exécuter n'importe quelle tâche mais avec des vitesses différentes. Ainsi si la vitesse du processeur 1 est double de celle du processeur 2, le processeur 1 met deux fois moins de temps que le processeur 2 pour exécuter la même tâche. **3 - Hétérogènes :** dans ce modèle les processeurs ne sont pas identiques et ils ne peuvent donc pas forcément exécuter toutes les tâches. En général ils sont spécialisés pour une fonctionnalité comme par exemple un traitement graphique (GPU, Graphics Processing Unit) ou encore le traitement de signal (DSP, Digital Signal Processor). Ce modèle est aussi connu sous le vocable *Asymmetric MultiProcessor* (AMP).

La plupart des recherches sur l'ordonnancement temps réel se concentre sur le modèle «Homogène »(SMP) parce que le problème d'ordonnancement est plus simple que pour le modèle «Uniforme ». Nous utilisons également ce modèle pour lequel M sera le nombre de processeurs identiques.

9.4 La politique d'ordonnancement

L'objectif d'une politique d'ordonnancement est de trouver un ordonnancement *faisable* pour la configuration de tâches, toutes les tâches respectant alors leurs échéances. La configuration de tâches est *ordonnançable* par un algorithme si ce dernier engendre un ordonnancement faisable. Un algorithme est dit *optimal* pour un modèle de tâches, s'il peut générer un ordonnancement faisable pour toutes les configurations ordonnançables par n'importe quel autre algorithme [17]. La politique d'ordonnancement définit la priorité relative de chaque tâche dans la configuration, ce qui permet de décider de l'allocation du processeur lorsque cela est nécessaire, par l'ordonnanceur qui choisit alors toujours la tâche de plus haute priorité. Un ordonnancement est qualifié de *preemptif* s'il autorise la suspension d'une tâche lors du réveil d'une tâche plus prioritaire, ou lorsque la politique est de nature *non-conservative*. [2]

La préemption, associée au changement de contexte, est une des causes des surcoûts système de la politique d'ordonnancement.

Cependant, pour simplifier les analyses, les coûts de préemption, changement de contexte et d'exécution des services système associés sont en général considérés comme nuls (ou pouvant être intégrés dans le temps d'exécution des tâches).
Du fait de la nature critique des systèmes temps réel durs il est nécessaire de vérifier a priori que toutes les tâches respectent leurs échéances. Diverses techniques sont disponibles pour cela, selon le modèle de tâches. Par exemple la connaissance d'une borne maximale d'utilisation pour un algorithme permet de vérifier que le facteur d'utilisation de la configuration de tâches respecte bien la borne. C'est en général une condition nécessaire, mais pas suffisante.

2. Un algorithme d'ordonnancement est dit "conservatif"s'il ne laisse jamais de processeur inactif tant qu'il y a une tâche pouvant s'exécuter dans le système ; Il est dit "non-conservatif"dans le cas contraire

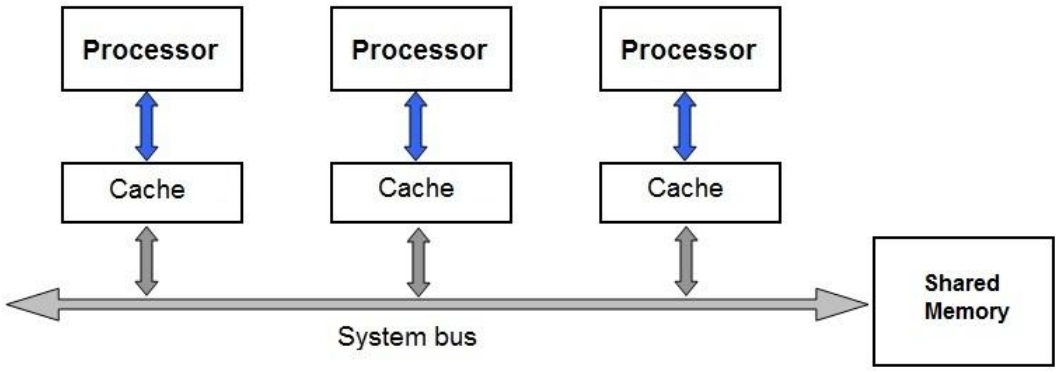


Figure 9.2: Une architecture SMP

9.4.1 Ordonnancement monoprocesseur

Dans un système monoprocesseur une seule unité d'exécution est utilisée pour toutes les tâches, et l'ordonnanceur décide alors de l'allocation temporelle du processeur aux tâches. De très nombreux algorithmes d'ordonnancement ont été proposés comme Least Laxity First (LLF) [50], Earliest Deadline First (EDF) [44, 24], Rate monotonic (RM) [44], Round Robin (RR) [14], et bien d'autres. Par exemple, LLF [50] et EDF [44, 24] sont optimaux pour les configurations de tâches périodiques à échéances implicites. Les priorités des tâches peuvent être : fixes pour la tâche (fixed task priority), fixes pour le job (fixed job priority) ou bien dynamiques.

Priorité fixe pour la tâche : la priorité de la tâche est définie à sa première exécution et elle reste constante. Rate monotonic (RM) [44] est un exemple d'algorithme à priorité fixe (ou statique) pour la tâche. Avec cet algorithme une configuration de N tâches périodiques synchrones et à échéances implicites est ordonnançable si le facteur d'utilisation satisfait la condition suivante [45] :

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq N(2^{1/N} - 1) \tag{9.3}$$

Priorité fixe pour le job : chaque job se voit assigner une priorité qui reste fixe jusqu'à sa terminaison. Earliest Deadline First (EDF) est un exemple d'algorithme à priorité fixe pour les jobs ; il donne la plus forte priorité au job qui a l'échéance la plus proche. Avec EDF, une configuration de tâches périodiques à échéances implicites est ordonnançable si et seulement si la condition suivante est vérifiée [45]:

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq 1 \tag{9.4}$$

Priorité dynamique : la priorité assignée à un job peut varier dans le temps, au cours de son exécution, même à l'intérieur d'une période. Least Laxity First (LLF) [50] est un exemple d'algorithme avec priorités dynamiques pour les jobs. La laxité d'un job est définie comme la durée maximale que ce job peut attendre avant de s'exécuter tout en respectant son échéance ; LLF donne alors la plus forte priorité au job qui a la plus petite laxité. Avec LLF une configuration de tâches périodiques à échéances implicites est ordonnançable sous la même condition nécessaire et suffisante que EDF.

9.4.2 Ordonnancement multiprocesseur

L'ordonnancement multiprocesseur peut être défini par : « Comment exécuter une configuration de tâches τ sur un jeu de M processeurs tout en respectant les contraintes temporelles des tâches ? ». Ce problème peut en fait être divisé en deux sous-problèmes :

- un problème d'allocation spatiale. Sur quel processeur doit s'exécuter une tâche?
- un problème d'affectation de priorité. Quel doit être l'ordre relatif d'exécution des tâches sur un processeur ?

Un ordonnanceur multiprocesseur est différent d'un ordonnanceur monoprocesseur car, en plus, il doit prendre en compte l'allocation spatiale des tâches. De ce fait ce ne peut malheureusement pas être une simple extension du problème d'ordonnancement monoprocesseur. Ainsi les politiques optimales en monoprocesseur comme EDF [44, 24] et LLF [50] ne conduisent pas aux mêmes que résultats dans le cas multiprocesseur amenant à de faibles facteurs d'utilisation pour les processeurs [27]. Par ailleurs on peut facilement mettre en évidence des anomalies d'ordonnancement dans le cas multiprocesseur. Une anomalie se produit lorsqu'un changement positif de caractéristiques temporelles dans la configuration conduit à un résultat négatif. Par exemple pour une configuration ordonnançable, l'augmentation de la période d'une tâche, en gardant constant les autres paramètres diminue le facteur d'utilisation (résultat positif), mais peut dans certains cas conduire à des ratés d'échéances, et donc une non-ordonnançabilité.

Les règles suivantes sont à respecter dans le cadre de l'ordonnancement multiprocesseur :

- un processeur ne peut pas exécuter plus d'une tâche à la fois ;
- une tâche (un job) ne peut pas s'exécuter sur plus qu'un processeur à un instant donné.

Selon le type d'ordonnancement une tâche peut être préemptée sur un processeur et, éventuellement après une pause, continuer sur un autre processeur ; ceci correspond à une *migration*. Les algorithmes d'ordonnancement sont alors divisés en 3 classes selon les possibilités de migration qu'ils autorisent aux tâches :

1. Pas de migration. La migration n'est pas autorisée et une tâche s'exécute toujours sur le même processeur ;
2. Migration limitée. Une tâche peut migrer sur un autre processeur seulement après l'exécution complète d'un job ;
3. Migration totale. Une tâche peut migrer vers un autre processeur même durant l'exécution d'un job.

Les algorithmes d'ordonnancement qui n'autorisent pas la migration sont les algorithmes partitionnés ; à l'opposé ceux qui l'autorisent sont les algorithmes globaux. Un événement important qui influença la recherche dans le domaine de l'ordonnancement multiprocesseur fut la publication de résultats de Dhall et Liu [27] en 1978. Ils utilisèrent EDF pour l'ordonnancement global de tâches périodiques à échéances implicites avec une configuration toutefois particulière. En effet, pour M processeurs, la configuration était constituée d'un jeu de M tâches avec une période unitaire et une très petite durée d'utilisation (2ϵ), et une tâche de période $1+\epsilon$ avec une durée unitaire. Bien

que le facteur total d'utilisation soit inférieur au nombre de processeurs cette configuration n'est pas ordonnançable quelle que soit la valeur de ϵ. Ce phénomène connu sous le nom d'*effet Dhall* a orienté les recherches vers les algorithmes partitionnés, les algorithmes globaux étant alors considérés comme inférieurs. Ce n'est que bien plus tard que la raison réelle de cet effet (problème du fort taux d'utilisation de la tâche unique) fut analysée (travaux de Philips et al. en 1997 [54] et Funk et al. en 2001 [32]), et que les travaux sur l'ordonnancement global prirent leur essor. Baruah et al. [56] proposèrent Proportionate fair, ou simplement PFair en 1996. C'est un algorithme global, optimal pour les configurations de tâches périodiques à échéances implicites. Il est fondé sur le concept de «fairness », c'est à dire l'équité à l'exécution et sera expliqué plus en détail au chapitre suivant. D'autres algorithmes ont ensuite été proposés, beaucoup fondés sur ce principe d'équité. Un résultat essentiel attaché à cet algorithme est qu'une configuration de tâches périodiques à échéances implicites est ordonnançable sur un jeu de M processeurs si et seulement si les conditions suivantes sont vraies :

$$U = \sum_{i=1}^{N} \left(\frac{T_i.e}{T_i.p} \right) \leq M \tag{9.5}$$

$$\forall i \;\; T_i.u_{(max)} \leq 1 \tag{9.6}$$

Classification des algorithmes d'ordonnancement multiprocesseur Comme déjà indiqué la classification principale est fondée sur les possibilités de migration offertes aux tâches. On distingue 3 catégories principales :

- l'ordonnancement partitionné ;
- l'ordonnancement global ;
- l'ordonnancement hybride.

Les techniques partitionnées et globales ont déjà été évoquées. L'ordonnancement hybride est une combinaison des deux précédents.

9.4.2.1 Ordonnancement partitionné

L'ordonnancement partitionné est le plus mature puisqu'il a profité de nombreux travaux antérieurs et de la réutilisation des résultats d'ordonnancement en monoprocesseur. En bref l'ordonnancement partitionné sur une architecture à M processeurs est décomposé en deux étapes :

- le partitionnement de la configuration de tâches en sous-ensembles ;
- l'ordonnancement monoprocesseur de chaque partition.

Le partitionnement de la configuration de tâches est similaire au problème de « bin packing », problème *NP-hard* [33] au sens fort. Dans notre cas les sacs sont les processeurs tandis que les éléments à placer sont les facteurs d'utilisation des tâches. La capacité des processeurs (les sacs) est donnée par le critère d'ordonnançabilité qui dépend de l'algorithme d'ordonnancement utilisé. De très nombreuses heuristiques de « Bin Packing » existent et peuvent être appliquées à ce problème. On peut citer First Fit (FF), Next Fit (NF), Best Fit (BF), First Fit with Decreasing Utilization $(FFDU)$ etc. On donne ci-après quelques éléments de comparaison en se basant sur un indicateur appelé *Approximation ratio*. Il permet de comparer la performance de l'algorithme par rapport à un algorithme optimal. « L'approximation ratio » R_A d'un algorithme A est défini par :

$$R_A = \underset{M_0 \to \infty}{max} \left(\underset{\forall \Gamma}{max} \left(\frac{M_A}{M_0} \right) \right) \tag{9.7}$$

avec M_0 le nombre de processeurs nécessaire à un algorithme optimal et M_A le nombre de processeurs nécessaire à l'algorithme A. La valeur minimale de 1 indique que l'algorithme A est optimal. Après l'étape de partitionnement on est ramené à un ordonnancement monoprocesseur avec une file d'attente des tâches prêtes par processeur. On donne ci-après quelques exemples de couples « Algorithme d'ordonnancement - Technique de partitionnement », en les caractérisant. Pour chacun on indique : le nom de l'algorithme, la signification de l'acronyme, la valeur de « L'approximation ratio », la borne supérieure du facteur d'utilisation (B_{sup}) et la complexité calculatoire (CC).

RM-FF [51] ; Rate Monotonic - First Fit ; RA=2.33 ; B_{sup}=$M(2^{1/2}-1)$; CC=*ONlogN*.

RM-NF [27] ; Rate Monotonic - Next Fit ; RA=2.67 ; B_{sup}=$\frac{M+1}{2}$; CC=*ONlogN*.

RM-FFDU [52] ; Rate Monotonic - First Fit with Decreasing Utilization ; RA=1.66 ; B_{sup}=$M(2^{1/2}-1)$; CC=*ONlogN*.

RM-GT [16] ; Rate Monotonic - General Task algorithm ; RA=2.33 ; B_{sup}=$0.5(M-1.42)$. ; CC=*ONlogN*.

EDF-FF [33] ; Earliest Deadline First - First Fit algorithm ; RA=1.7 ; CC=*ONlogN*.

EDF-BF [33] ; Earliest Deadline First - Best Fit algorithm ; RA=1.7 ; CC=*ONlogN*.

L'ordonnancement partitionné est avantageux car il réduit le problème de l'ordonnancement multiprocesseur à celui de l'ordonnancement monoprocesseur, beaucoup mieux maîtrisé, une fois le placement effectué. De plus il n'y a pas de surcoûts dus à la migration. Cependant son principal inconvénient est qu'il est sous-optimal et ne permet pas une

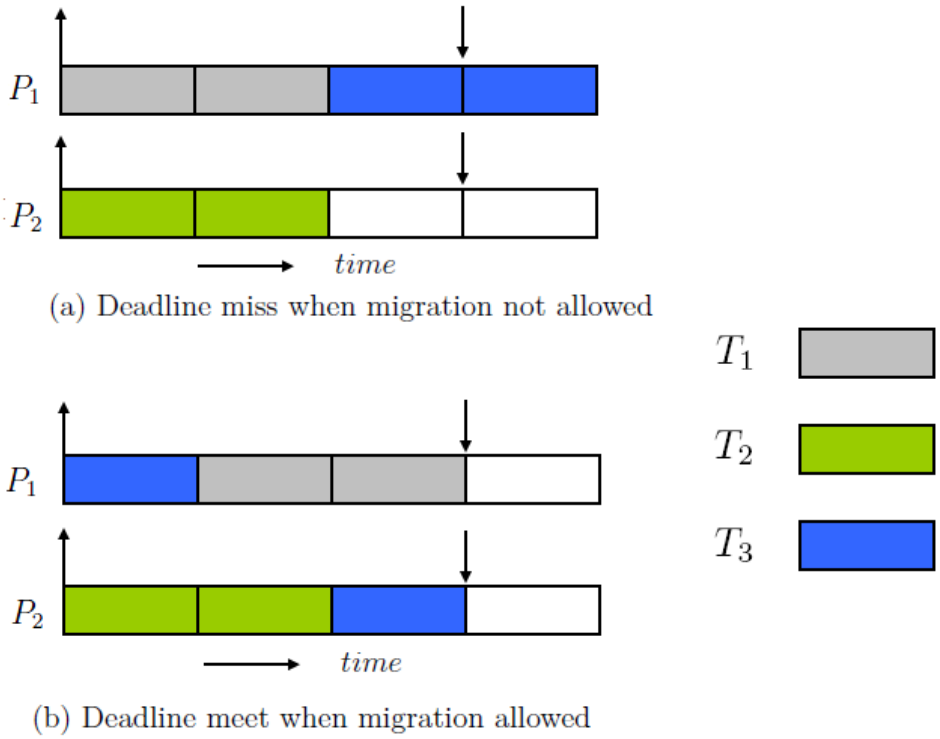


Figure 9.3: La configuration est ordonnançable avec la migration

utilisation maximale des ressources. En effet la borne supérieure du facteur total d'utilisation pour une configuration de tâches périodiques à échéances implicites est [12] :

$$U_P = \frac{M+1}{2} \tag{9.8}$$

Il est également facile de montrer que certaines configurations ne sont ordonnançables qu'avec un ordonnancement autorisant la migration. L'exemple de la figure 9.3 le montre avec une configuration de 3 tâches et 2 processeurs. Chaque tâche a une échéance de 3 unités de temps et une durée d'exécution de 2 unités de temps. Dans la figure 9.3 (a) l'échéance de T3 est violée car la migration n'est pas autorisée, ce qui n'est pas le cas de la figure 9.3 (b) ou la migration est autorisée.

9.4.2.2 Ordonnancement global

Dans la technique d'ordonnancement global il n'y a qu'une seule file d'attente pour les tâches prêtes, pour tous les processeurs, et un seul ordonnanceur responsable de l'allocation spatiale et temporelle des tâches. La migration des tâches est autorisée, au niveau tâche ou au niveau job selon les algorithmes. Certains algorithmes sont optimaux alors que d'autres ne le sont pas.

Ordonnancement global non optimal On retrouve la transposition des algorithmes classiques en monoprocesseur (fixed task priority et fixed job priority) ainsi que quelques améliorations. Un résultat important de B. Anderson et al. [9] a montré que la borne supérieure du facteur d'utilisation pour une configuration de tâches périodiques avec échéances implicites et un algorithme d'ordonnancement global à priorités fixes pour les jobs est donnée par l'équation :

$$U_{FJP} = \frac{(M+1)}{2} \tag{9.9}$$

On peut noter que c'est également la valeur déjà mentionnée pour les algorithmes partitionnés.

En plus de global RM, global DM et global EDF d'autres algorithmes globaux ont été proposés dans cette classe. On peut citer *RM-US*(ς) proposé par B. Andersson et al. [9] en 2003, dans lequel les tâches avec un facteur d'utilisation supérieur à ς sont les plus prioritaires. Les auteurs ont montré que *RM-US*($\frac{M}{3M-2}$) avait une borne supérieure du facteur d'utilisation de $\frac{M^2}{3M-2}$. D'autres exemples sont fournis dans la version étendue en langue anglaise.

Ordonnancement global optimal Les algorithmes précédents utilisaient une technique conservative alors que ceux évoqués maintenant sont majoritairement basés sur une technique non-conservative (il y a des exceptions). Deux branches principales ont été établies pour ces algorithmes globaux optimaux, conçus pour des configurations de tâches périodiques à échéance implicite : $PFair$ [56] et Deadline Partitioning Fair ou plus simplement $DP\text{-}Fair$ [43]. Les deux catégories sont basées sur le principe de l'équité (« fairness »), c'est la manière dont elle est obtenue qui les différencie. Dans la catégorie $PFair$ trois algorithmes optimaux [56], PF, PD et PD^2, ont été proposés en technique non-conservative, et un quatrième *ER-Fair* en technique conservative. Dans la catégorie *DP-Fair* quatre algorithmes, *DP-Wrap* [43], Boundary fair [61], $LLREF$ [36] et *LRE-TL* [30] peuvent être cités en technique non-conservative, et deux autres, $TRPA$ and $ETNPA$, en technique conservative. Ces algorithmes sont présentés en détail dans le chapitre suivant.

Les algorithmes globaux optimaux autorisent une meilleure utilisation des ressources de par le principe de l'équité et par la migration. En contrepartie la migration peut engendrer des surcoûts importants à cause du trafic généré sur les bus et le rechargement des caches de premier niveau.

9.4.3 Ordonnancement hybride

L'ordonnancement partitionné engendre de faibles surcoûts mais a des bornes d' utilisation relativement basses. D'un autre côté l'ordonnancement global offre de meilleures performances avec des surcoûts importants. L'ordonnancement hybride est alors un compromis entre les deux : il permet d'obtenir de meilleures bornes supérieures du facteur d'utilisation avec des surcoûts moins importants que ceux des techniques globales. Deux techniques principales ont émergé : celle du semi-partitionnement et celle de « clustering ». A titre d'exemple d'une technique de semi-partitionnement on peut citer EKG [10] proposé en 2006 par B. Anderson et Tovar pour des configurations de tâches périodiques à échéances implicites. Dans cette technique certaines tâches sont fixées à des processeurs via un algorithme de Bin packing et d'autres sont autorisées à migrer à l'intérieur d'un groupe de processeurs. Les tâches sont ordonnancées par EDF. A titre d'exemple d'une technique de clustering on peut citer Earliest Deadline Deferrable Portion (EDDP) proposé par Shin et al. [41] en 2008. Dans cette technique les processeurs sont répartis dans un certain nombre de clusters et les tâches sont assignées aux clusters. Dans chaque cluster les tâche sont ordonnancées selon une technique EDF globale. Si le nombre de processeurs dans chaque cluster vaut 1, on retrouve le cas de l'ordonnancement partitionné ; s'il n'y a qu'un seul cluster on se retrouve dans le cas de l'ordonnancement global

9.5 Conclusion

L'ordonnancement partitionné et l'ordonnancement global ont chacun leurs avantages et leurs inconvénients, l'ordonnancement hybride essayant de ne garder que les avantages des deux. Même si l'ordonnancement global offre des techniques qui sont optimales de nombreuses questions se posent quant à leur usage en pratique à cause des surcoûts importants engendrés par le principe même de ces algorithmes. Dans ce travail de thèse ce sont néanmoins les techniques d'ordonnancement globales qui sont privilégiées avec l'objectif de mettre en oeuvre des techniques de minimisation des surcoûts dus aux préemptions et migrations, sans toutefois augmenter la complexité des algorithmes. Notre travail s'appuie sur un modèle de tâches périodiques à échéances implicites s'exécutant sur une architecture symétrique.

10

Etat de l'art des algorithmes globaux optimaux

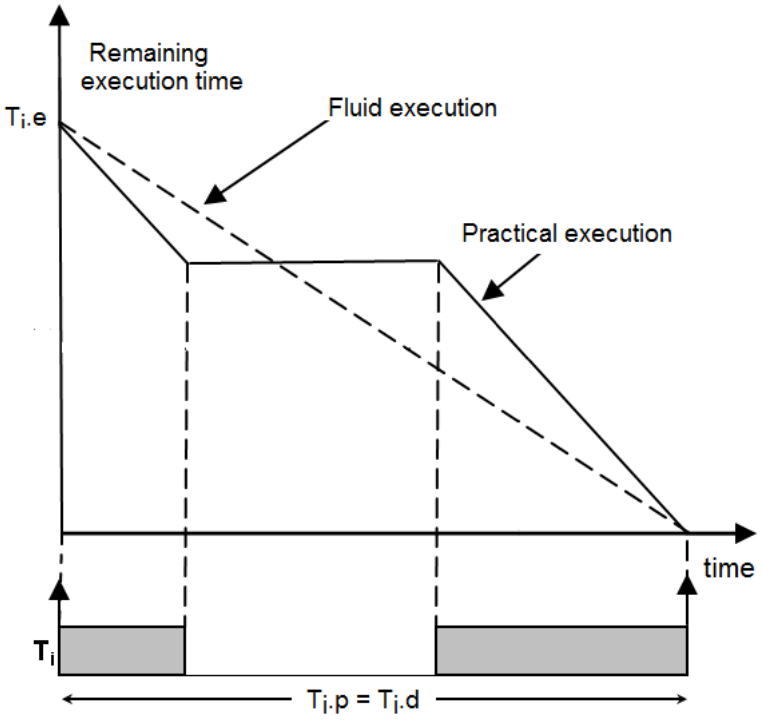


Figure 10.1: Exécution fluide et exécution reéle

10.1 Introduction

Les algorithmes multiprocesseur optimaux actuellement connus sont de la classe des algorithmes globaux avec priorité dynamique pour les jobs. Jusqu'à récemment les deux catégories d'algorithmes optimaux pour des tâches périodiques avec échéances implicites s'exécutant sur une architecture multiprocesseur symétrique étaient *PFair* [56] et *DP-Fair* [43]. Récemment (fin 2011) un autre algorithme, prouvé optimal, a été proposé ; il se nomme *RUN* [55]. Dans ce chapitre nous allons essentiellement présenter *PFair* et *DP-Fair*, une petite partie étant consacrée à *RUN*.

Les deux catégories précédemment mentionnées, *PFair* et *DP-Fair* ont été conçues en s'appuyant sur le principe de *fairness,* c'est-à-dire de l'équité, non pas au sens strict mais au sens de "recevoir en fonction de ce dont on a besoin", donc dans notre cas en fonction du facteur d'utilisation d'une tâche. On parle également d'ordonnancement *fluide* et ce concept est illustré sur la figure . Elle montre l'exécution d'un job de la tâche T_i avec ses caractéristiques $T_i.e$ (la durée d'exécution) et $T_i.d$ (l'échéance égale à la période). L'exécution fluide serait celle qui verrait l'exécution du job à la vitesse constante $-T_i.u$, l'exécution "pratique ayant pour sa part une pente -1 quand le job est exécuté, et une pente de valeur nulle quand il est au repos.

Les deux stratégies *PFair* et *DP-Fair* font de l'ordonnancement par intervalle. Elles

diffèrent par :

- la longueur de l'intervalle ;
- la précision avec laquelle l'ordonnancement fluide est suivi ;

Les deux stratégies sont expliquées en détail dans les paragraphes qui suivent.

10.2 PFair : Proportionate Fair scheduling

10.2.1 Principe

PFair a été proposé par Sanjoy Baruah et al. en 1996 [56] pour les tâches périodiques à échéances implicites. Comme indiqué précédemment le principe est fondé sur la distribution équitable de la ressource processeur en fonction des besoins : chaque tâche reçoit une allocation de la ressource proportionnelle à son facteur d'utilisation. Une illustration en est donnée par la figure 10.2 (a) qui montre l'exécution de deux tâches : T_1 avec $T_1.u = 0.6$ et T_2 avec $T_1.u = 0.4$, sur un seul processeur P. Compte-tenu des deux valeurs de facteur d'utilisation le processeur est à pleine charge et la figure 10.2 (a) est une représentation de ce partage proportionnel. En pratique, le processeur est attribué à une tâche avec sa pleine capacité, et une manière de réaliser l'ordonnancement fluide est de diviser l'exécution de chaque tâche en petits intervalles alloués aux jobs selon leur facteur d'utilisation. C'est ce que montre la figure 10.2 (b) dans laquelle le temps est conditionné en intervalles unitaires (peu importe la durée réelle), donc égaux (principe de *PFair*). La figure montre également l'allocation des slots temporels selon le facteur d'utilisation du job (6 slots pour T_1 et 4 slots pour T_2 . La notion de fluidité impose, si l'on veut se rapprocher de l'ordonnancement fluide idéal, de positionner convenablement dans le temps l'attribution des slots d'un job. Afin de mesurer la différence entre ordonnancement idéal et pratique, la notion de *lag* a été introduite. Cette notion qui se traduit par une fonction est définie pour une tâche T_i à l'instant t par :

$$lag(T_i,\, t) = T_i.u * t - \sum_{l=0}^{t-1} S(T_i,\, l) \tag{10.1}$$

avec $T_i.u*t$ qui représente le temps total d'exécution alloué à la tâche T_i à l'instant t. S est la fonction d'exécution réelle définie pour chaque slot. Si $S(T_i, l) = 1$ cela signifie que la tâche a été exécutée durant le slot l tandis que $S(T_i, l) = 0$ indique qu'elle n'a pas été exécutée. A titre d'exemple, sur la figure 10.2 (b) la fonction lag des deux tâches à l'instant 8 donne comme résultats

$lag(T_1, 8)$ = (0.6 $\times$ 8) - 5 = -0.2
$lag(T_2, 8)$ = (0.4 $\times$ 8) - 3 = 0.2

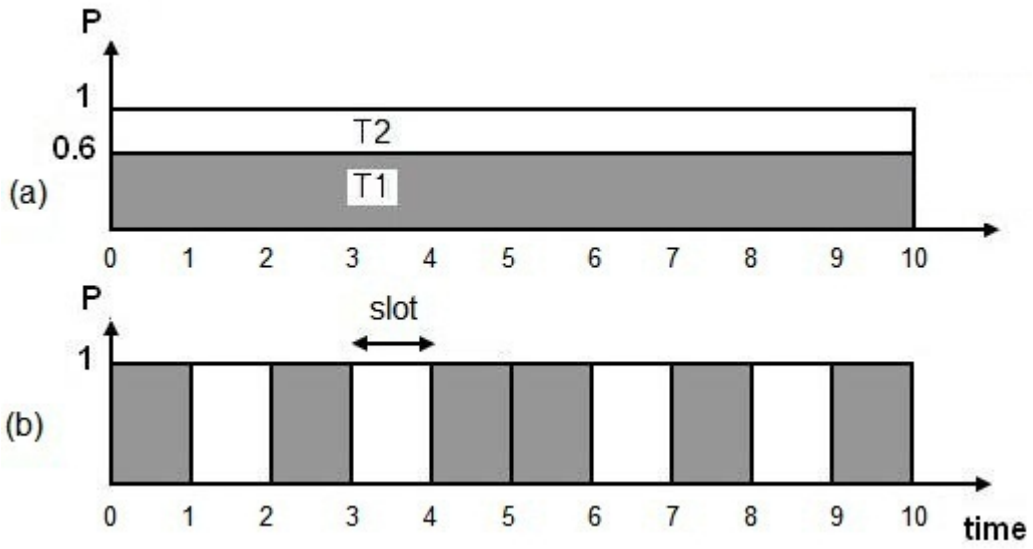


Figure 10.2: (a) Ordonnancement fluide (b) Allocation des slots temporels avec $T_1.u$ = 0.4 and $T_2.u$ = 0.6

A partir de cette fonction de mesure de l'écart par rapport au schéma fluide idéal, un ordonnancement est qualifié de *PFair* s'il remplit, pour tout job, la condition suivante :

$$-1 < lag(T_i, t) < 1 \quad \forall\, t > 0 \; and \; \forall\, i \tag{10.2}$$

Ainsi *PFair* réalise l'ordonnancement des tâches sur les processeurs de telle manière qu'à tout instant t le temps cumulé alloué à la tâche T_i de facteur d'utilisation $T_i.u$ est, soit $\lfloor t * T_i.u \rfloor$, soit $\lceil t * T_i.u \rceil$. Il revient au même de dire que l'allocation réalisée ne s'éloigne pas de plus d'un quantum de temps par rapport à une exécution fluide.

La figure 10.3 montre l'exécution d'une tâche T_i avec un facteur d'utilisation de $T_i.u = 0.6$ qui respecte ce principe *PFair* (c'est le temps effectif d'exécution qui est représenté en ordonnée et non le temps restant comme dans la figure 10.3 . On y retrouve la ligne d'exécution fluide en pointillé, et la ligne d'exécution réelle, toujours comprise entre les deux lignes de respect de la propriété de fluidité (lag compris entre -1 et +1).

Baruah a démontré le théorème suivant, conséquence des principes d'un ordonnancement *PFair*.

Théorème [56] Un ordonnancement *PFair* sur M processeurs existe si et seulement si le facteur total d'utilisation est inférieur ou égal au nombre de processeurs :

$$U \leq M \tag{10.3}$$

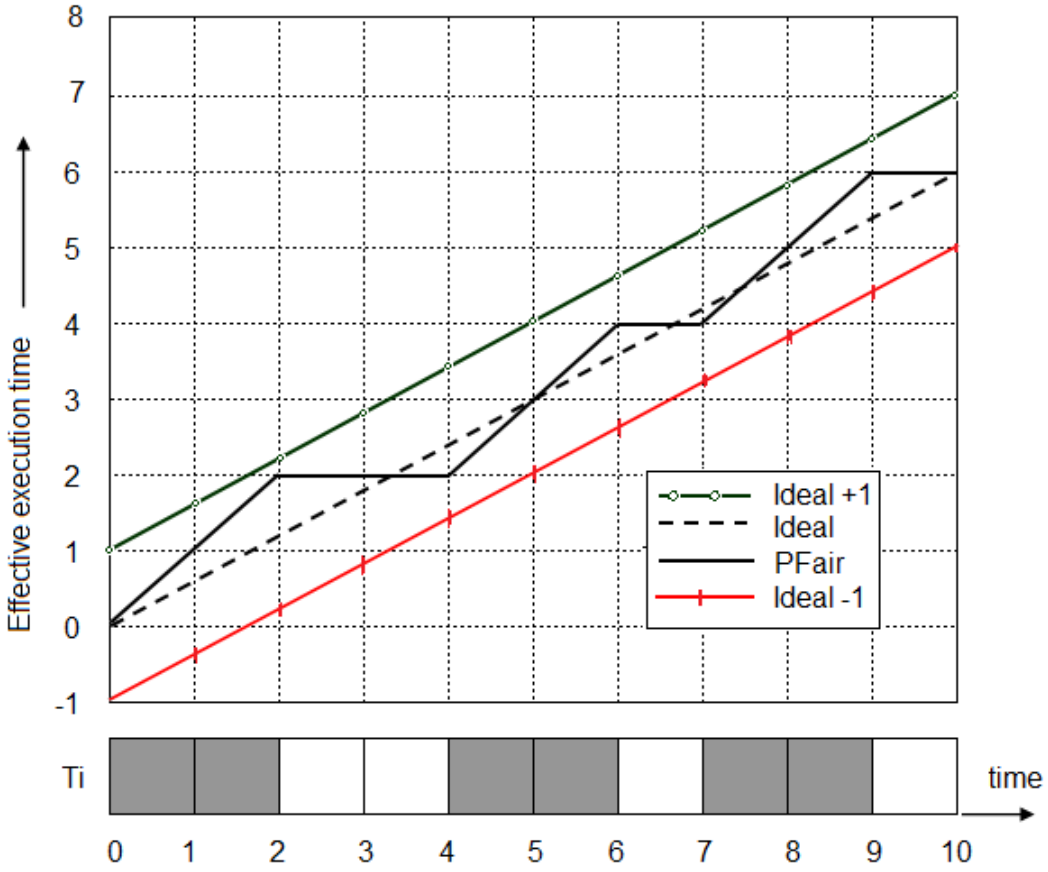


Figure 10.3: Exécution d'une tâche PFair

(chaque facteur individuel étant bien sûr inférieur ou égal à 1).

10.2.2 Ordonnancement PFair

L'ordonnancement *PFair* est réalisé à chaque quantum de temps. Il se fait non pas sur les tâches mais sur une division de ces dernières : les sous-tâches. Chaque tâche T_i est en effet divisée en plusieurs sous-tâches, chacune devant recevoir un quantum de temps pour s'exécuter. Ainsi le nombre de sous-tâches d'une tâche est égal à sa durée d'exécution exprimée en quantums de temps. Pour une tâche T_i, $T_{i.k}$ représente la k_{th} sous-tâche de la tâche, avec $k \geq 1$. Pour construire l'ordonnancement *PFair* il faut calculer pour chaque sous-tâche $T_{i.k}$ deux grandeurs : la pseudo-date-de-réveil $r(T_{i.k})$ et la pseudo-échéance $d(T_{i.k})$. Elles sont définies par :

$$r(T_{i.k}) = \left\lfloor \frac{k-1}{T_i.u} \right\rfloor \tag{10.4}$$

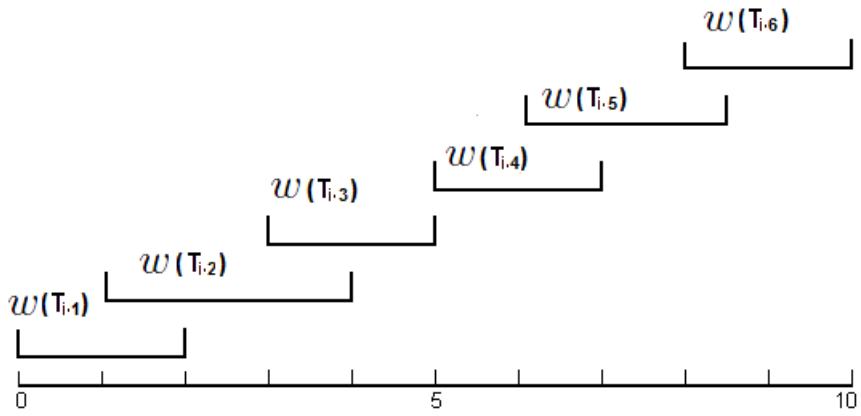


Figure 10.4: Les fenêtres de l'ordonnancement PFair pour une tâche avec $T_i.u = 0.6$

$$d(T_{i.k}) = \left\lceil \frac{k}{T_i.u} \right\rceil \tag{10.5}$$

La durée entre les deux dates caractérisant la sous-tâche est dénommée une *fenêtre* (sous-entendu d'exécution). La fenêtre pour l'exécution de la k^{th} sous-tâche de la tâche T_i est donc définie par :

$$w(T_{i.k}) = [r(T_{i.k}), d(T_{i.k}))] \tag{10.6}$$

Chaque sous-tâche doit s'exécuter dans sa fenêtre pour garantir une exécution *PFair*. La longueur de la fenêtre dépend du facteur d'utilisation de la tâche : plus le facteur est grand et plus la fenêtre est petite, et vice-versa ; si le facteur d'utilisation de la tâche est 1 la longueur de la fenêtre est 1 puisqu'il doit y avoir une exécution continue de la tâche. La sous-tâche ne pouvant s'exécuter que durant un seul quantum de temps, et sa fenêtre pouvant "s'étaler"sur plusieurs quantums on peut remarquer que cela est caractéristique d'un comportement non-conservatif pour l'ordonnancement *PFair*.

La figure 10.4 illustre la décomposition d'une tâche en sous-tâches et les fenêtres d'exécution associées. Dans cette figure la tâche T_i a un facteur d'utilisation $T_i.u = 0.6$. Il y a 6 sous-tâches et donc 6 fenêtres. A titre d'exemple la fenêtre de la sous-tâche $T_{i.3}$ a les caractéristiques suivantes :

$$r(T_{i.3}) = \left\lfloor \frac{3-1}{6/10} \right\rfloor = 3$$
$$d(T_{i.3}) = \left\lceil \frac{3}{6/10} \right\rceil = 5$$

Comme indiqué précédemment les décisions d'ordonnancement dans *PFair* sont prises au début de chaque quantum. A ce moment les M tâches prêtes les plus priori-

Task	$T_i.p = T_i.d$	$T_i.e$
T_1	10	6
T_2	8	4
T_3	10	9

Table 10.1: Configuration utilisée pour la simulation de la figure 10.5

taires sont exécutées sur les M processeurs. Une sous-tâche$T_{i.k}$ est prête à l'instant t si elle satisfait les deux conditions :

- la sous-tâche précédente est terminée (condition non requise pour la première sous-tâche d'un job) ;
- la date doit être supérieure ou égal à la pseudo-date-de-réveil de la sous-tâche.

Les priorités des sous-tâches sont définies par un ensemble de règles explicitées ci-après pour des implémentations existantes. Il est à noter qu'un ordonnancement *PFair* n'impose rien quant à l'allocation des processeurs une fois les priorités des tâches prêtes définies.

10.2.3 Algorithmes PFair

PF, PD and PD^2 sont trois algorithmes *PFair* qui ont été prouvés optimaux [56]. Ils utilisent tous l'algorithme (“ early pseudo-deadline first ”)$(EPDF)$ pour trouver les sous-tâches les plus prioritaires parmi les sous-tâches prêtes, mais ils utilisent des règles différentes pour décider en cas d'égalité (“ tie breaking rules ”).

Des trois algorithmes PD^2 est considéré comme le plus efficace et les règles qu'il utilise sont présentées dans la version anglaise de la thèse.

10.2.4 Implémentation d'un algorithme PFair

Nous avons fait une implémentation de l'algorithme *PFair* PD^2 au cours de la thèse, afin de pouvoir le tester avec le simulateur STORM [1] . Dans cette implémentation nous avons réutilisé des travaux de Chandra et al. [19]. On montre sur la figure 10.5 les résultats de la simulation d'une configuration donnée à la table 10.1 sur 2 processeurs. Pour une durée de 50 unités de temps on voit les diagrammes de Gantt des 3 tâches et l'occupation des 2 processeurs.

10.2.5 Limitations de PFair

Un certain nombre d'objections ont été faites quant à l'utilisation pratique d'un tel principe. Elles sont rapidement résumées ci-après :

1. voir le chapitrer 5 pour d'autres détails au sujet de STORM. [2, 60]

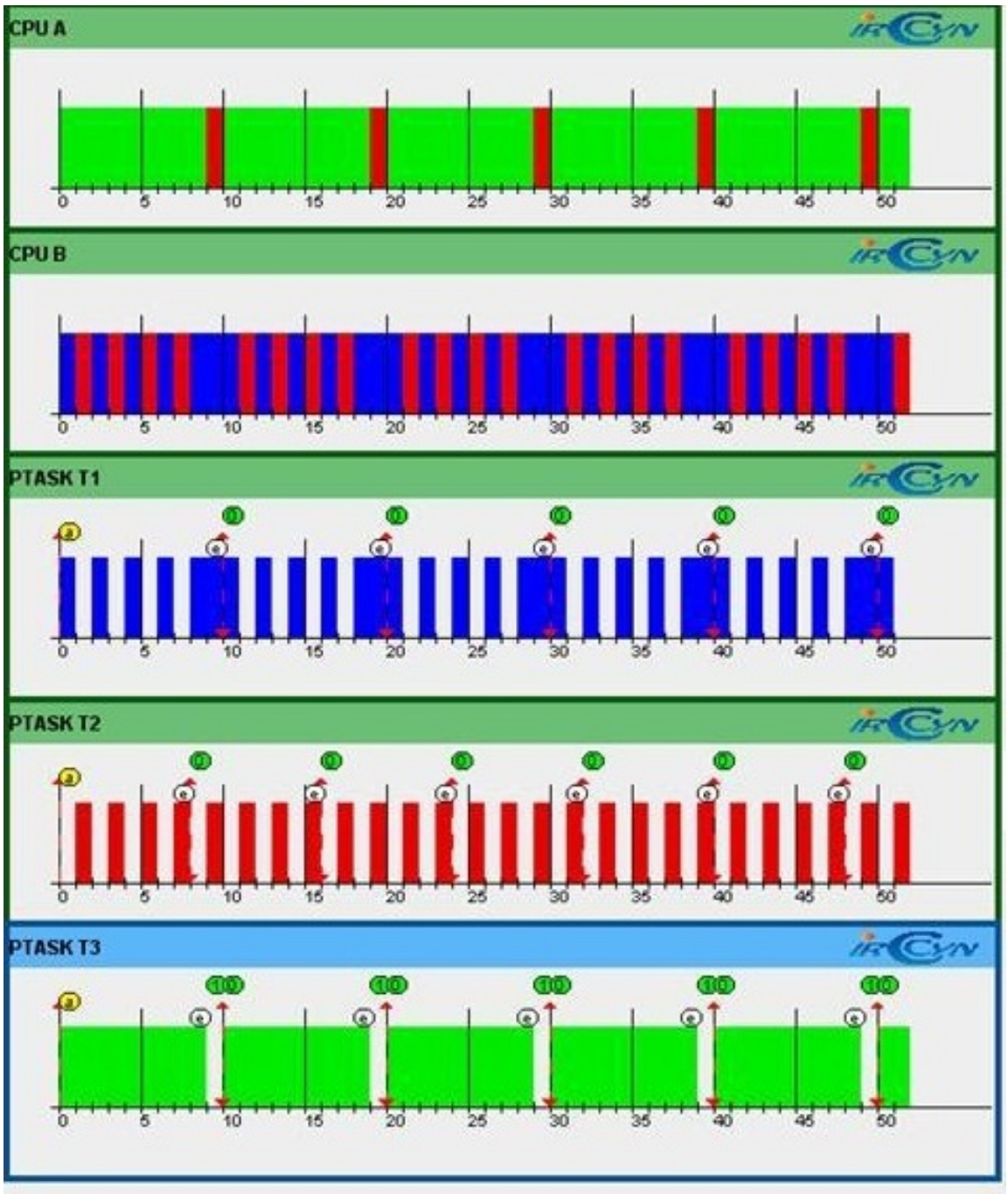


Figure 10.5: Un exemple d'exécution PFair d'une configuration

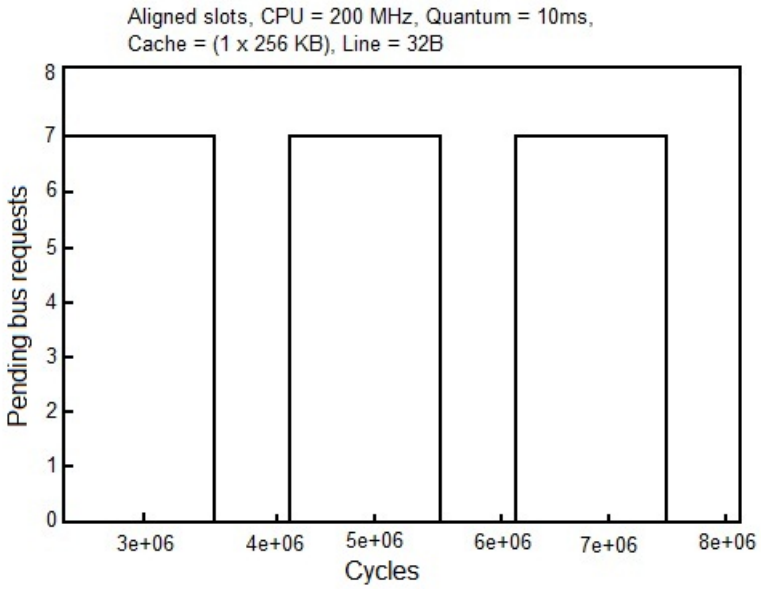


Figure 10.6: Charge d'un bus partagé par 8 processeurs aux points d'ordonnancement [37]

1. bien que *PFair* soit théoriquement optimal son principe même de fluidité de l'exécution, ou de l'équité dans l'attribution des ressources, engendre inévitablement d'innombrables préemptions et migrations des tâches [57]. Ceci ne peut que conduire à de mauvaises performances (surtout au niveau des caches de premier niveau des processeurs) et on peut donc penser que le principe de *PFair* est inutilisable en pratique ;
2. du fait de l'ordonnancement de l'ensemble des tâches à chaque quantum on peut penser qu'une architecture à mémoire partagée sur bus va être très surchargée à chaque quantum, après la prise des décisions d'ordonnancement, et une expérimentation rapportée dans [37] l'a effectivement démontré. La figure 10.6 [37] montre comment la concomitance des requêtes sur le bus peut engendrer une forte surcharge.
3. on peut également remarquer que si le temps d'exécution d'une tâche n'est pas son pire cas il y a aura perte de quantums, et il ne semble pas possible de les « récupérer »simplement.

Cependant, malgré ces remarques pessimistes :

1. des travaux ont montré que les résultats d'une expérimentation sous Linux étaient tout à fait acceptables (Calandrino et al. [18]) ;
2. les progrès sur les architectures matérielles peuvent permettre de limiter les effets

négatifs des migrations ;

3. des améliorations dans le principe, sans perte d'optimalité, peuvent se révéler efficaces (voir la version anglaise de la thèse).

10.2.6 Extensions de PFair

10.2.6.1 Le modèle Early Release Fair

Le principe de *PFair* a été étendu vers une version conservative connue sous le nom de *Early Release Fair* ou *ERFair*. Dans un ordonnancement *ERFair* les sous-tâches, hormis la première, n'ont pas de pseudo-date-de-réveil et elles peuvent s'exécuter dans la mesure où la sous-tâche précédente s'est exécutée. Dans cette version, qui de ce fait est conservative, la contrainte sur la valeur de la fonction lag est donnée par :

$$lag(T_i, t) < 1 \quad \forall\ t > 0\ and\ \forall\, i \tag{10.7}$$

La figure 10.7 donne un exemple d'un tel ordonnancement pour une tâche ayant un facteur d'utilisation de 0.6. Pour cette exécution, à titre d'exemple, la valeur de la fonction lag aux instants 2 et 5 est :

$lag(T_i, 5)$ = (0.6 $\times$ 5) - 5 = -2
$lag(T_i, 9)$ = (0.6 $\times$ 9) - 5 = 0.4

En terme de performances un ordonnancement *ERFair* présente un nombre moindre de préemptions comparé à la version non-conservative. Ceci est dû bien sûr à la possibilité d'enchainer l'exécution des sous-tâches.
PF, PD and PD^2 peuvent être adaptés pour donner une version conservative. Par exemple *ER-PD* a été prouvé plus efficace que sa version non-conservative et a même été étendu à la prise en compte des tâches sporadiques et intra-sporadiques (voir la version anglaise sur ce point) [36].

10.2.6.2 Le modèle PFair Staggered

Le problème de l'engorgement d'un bus partagé pour l'accès à mémoire a déjà été évoqué et des travaux ont cherché à éviter ce problème : c'est l'objet du modèle *PFair Staggered* proposé par Holman et Anderson [37, 53].

On peut facilement comprendre le principe de ce modèle sur la figure 10.8. Dans la partie (a) on retrouve les points d'ordonnancement, aux mêmes instants, au début de chaque quantum. Au contraire, dans la partie (b), on voit que chaque processeur décide de son nouvel ordonnancement sans synchronisation explicite avec les autres processeurs. Tous les points d'ordonnancement sont décalés les uns des autres, d'une durée

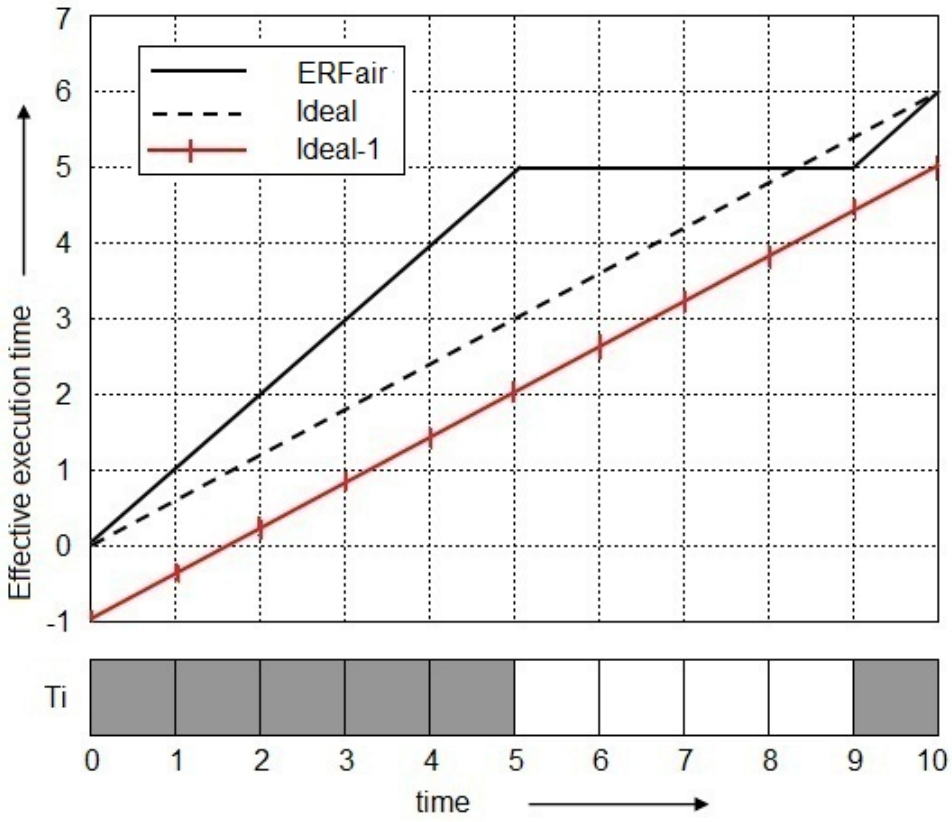


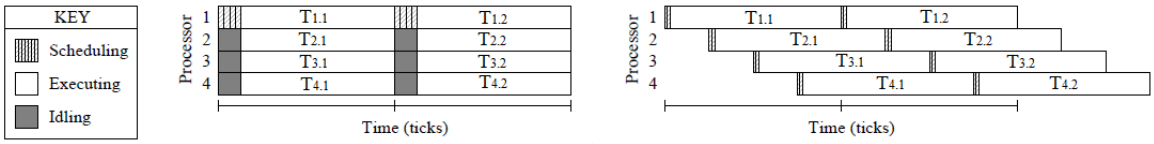
Figure 10.7: Exécution d'une tâche avec le principe *ERFair*

égale à un quart de la durée du quantum dans cet exemple à 4 processeurs. Les expérimentations menées ont montré une bien meilleure performance pour le bus partagé : la figure 10.9 montre la nouvelle charge du bus dans les mêmes conditions expérimentales que celles de la figure 3.6 déjà présentée.

10.2.6.3 Autres extensions

On trouvera dans la version anglaise de la thèse deux autres extensions décrites :

1. le modèle intra-sporadique [6, 57] qui applique le principe sporadique aux sous-tâches (date de réveil et échéance) ;
2. le concept de « supertasking »[49] dans lequel, pour diminuer les surcoûts, certaines tâches de la configuration qui sont attachées à des processeurs, sont regroupées dans une «super tâche »ordonnancée comme un tout.



(a) Aligned Model (b) Staggered Model

Figure 10.8: Modèle PFair et modèle PFair Staggered [37]

10.3 Ordonnancement Deadline Partitioned Fair

10.3.1 Principe

En réponse aux différentes critiques formulées précédemment par rapport aux surcoûts imposés par le principe même de *PFair*, le modèle *Deadline Partitioned Fair* [43] ou plus simplement *DP-Fair* propose une autre approche toujours optimale, ayant la même CNS d'ordonnançabilité que *PFair*. Dans ce modèle les intervalles d'ordonnancement ne sont plus constants et de durée unitaire comme dans *PFair* : ils correspondent aux échéances des tâches, ce qui diminue fortement le nombre de points d'ordonnancement. A l'intérieur d'un intervalle, l'ordonnancement des tâches suit toujours le principe de l'équité en attribuant la ressource proportionnellement aux besoins des tâches.

10.3.2 Division du temps en intervalles

On peut remarquer qu'avec des algorithmes comme EDF ou LLF il y a une méconnaissance des contraintes des tâches les unes par rapport aux autres : l'objectif de chaque tâche est son échéance ; ceci explique d'ailleurs pourquoi ces algorithmes ont de mauvaises performances en cas de surcharge. Le fait de faire apparaître des instants d'ordonnancement plus petits, avec une échéance ou pseudo-échéance commune à toutes les tâches dans cet intervalle est un moyen de remédier à l'inconvénient mentionné. Dans *DP-Fair* un intervalle d'ordonnancement est défini par les dates d'échéance des tâches (ou ce qui revient au même les dates de réveil, puisque le modèle est à échéances implicites), comme le montre l'exemple de la figure 10.10. Chaque date d'un intervalle est appelé une borne (boundary). Ainsi dans l'exemple de la figure 10.10 avec les trois tâches T_1, T_2 and T_3 les intervalles sont délimités par les bornes b_0, b_1, b_2, etc. Il est évident au vu de l'exemple que la longueur d'un intervalle n'est, a priori, pas constante. Le problème suivant est donc de résoudre l'ordonnancement à l'intérieur d'un intervalle (parfois appelé également un noeud), de telle manière que l'ordonnancement global (sur l'hyperpériode) soit valide et respecte bien les échéances de toutes les tâches.

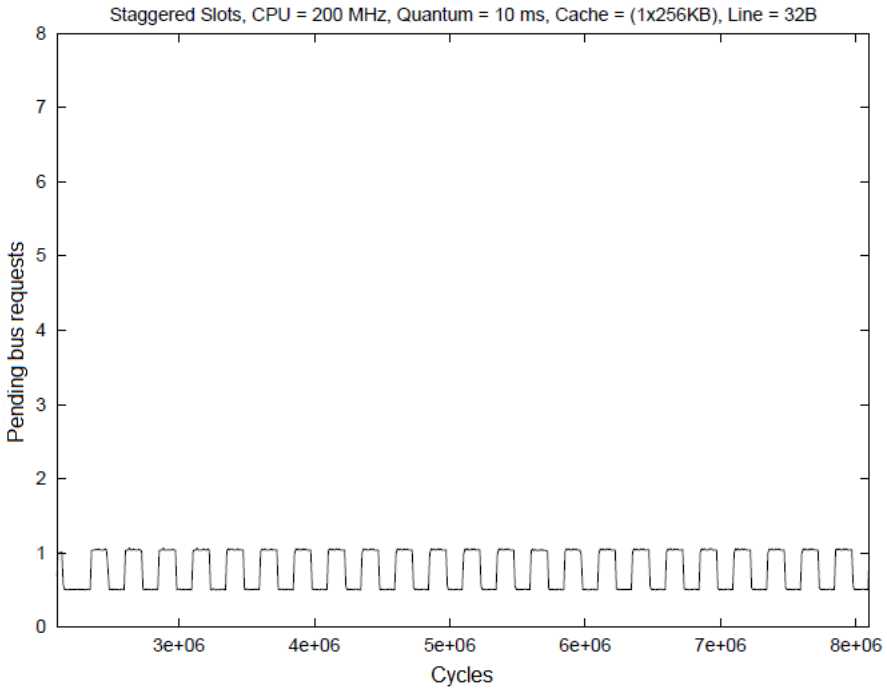


Figure 10.9: Charge d'un bus partagé par 8 processeurs aux points d'ordonnancement avec le modèle PFair Staggered [37]

10.3.3 Ordonnancement dans un intervalle

Comme dit précédemment la base de *DP-Fair* reste l'ordonnancement fluide, et à l'intérieur d'un intervalle les jobs se voient allouer une part des ressources proportionnelle à leurs besoins. Pour l'ordonnancement dans un intervalle, deux aspects doivent être considérés :

– en premier, le calcul des unités de temps à allouer à chaque job ;

– ensuite, l'allocation temporelle des processeurs aux jobs tout au long de l'intervalle.

Afin de présenter les concepts, nous allons nous appuyer sur une représentation graphique proposée par Funaoka et al. [28] : le *TN-plane* (Time and Nodal remaining execution time plane). Un exemple de *TN-plane* est montré sur la figure 10.11. En abscisse on trouve l'intervalle considéré débutant en b_k et se terminant en b_{k+1}, et en ordonnée le temps alloué à la tâche en début d'intervalle ($T_i.l$). La ligne pointillé montre ce que serait une exécution fluide de la tâche T_i, en respectant son échéance en b_{k+1}. Un exemple d'exécution réelle est également montré. Deux lignes sont également mentionnées : 1- *(NNRH)* (No Nodal Remaining execution time Horizon) est l'abscisse : si la trajectoire d'une tâche l'atteint c'est qu'elle a consommé toutes les unités de temps qui lui étaient attribuées ; 2- *(NNLD)* (No Nodal Laxity Diagonal) est l'hypoténuse du triangle rectangle isocèle formé en reportant en ordonnée la longueur de l'intervalle : si la trajec-

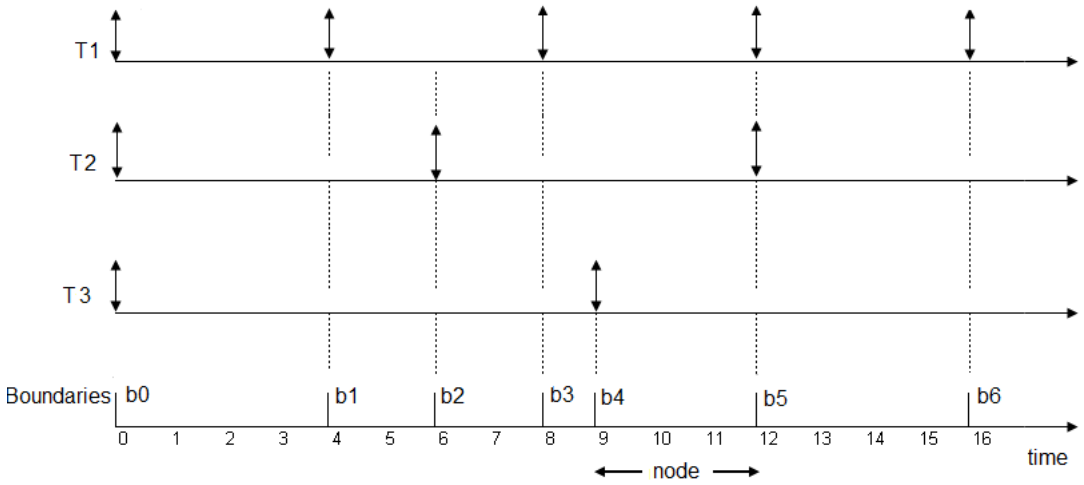


Figure 10.10: Définition des intervalles d'ordonnancement dans *DP-Fair*

toire d'une tâche l'atteint c'est qu'elle n'a plus aucune laxité et qu'elle doit s'exécuter continument jusqu'à la fin de l'intervalle, si elle ne veut pas manquer son échéance locale (ou réelle).

La figure 10.11 ne considérait qu'une seule tâche. La figure 10.12 montre les *TN-plane* associés individuellement à 3 tâches, ainsi que les *TN-plane* résultant pour chaque intervalle en superposant les exécutions fluides des tâches. Comme on le voit nettement pour la tâche T_1 entre les instants b_k et b_{k+2}, les triangles isocèles des *TN-plane* des 2 intervalles peuvent être positionnés de manière à montrer l'exécution fluide sur la période de la tâche T_1.

La figure 3.13 donne un exemple d'ordonnancement à l'intérieur d'un *TN-plane* sur un intervalle unique entre b_k et b_{k+1}. L'exécution des 4 tâches de cet exemple, avec 2 processeurs, peut y être observée grâce aux mouvements des jetons (cercles) montrés par les flèches. L'allocation initiale a donné à chaque tâche un nombre d'unités de temps ($T_i.l$). Entre les instants b_k et t_1 les tâches T_1 et T_2 s'exécutent sur les processeurs $P1$ et $P2$, et T_2 a consommé ses unités de temps à t_1 (elle a atteint la ligne $NNRH$) ; elle est alors remplacée par T_3. A l'instant t_2, le jeton de T_4 (qui ne s'est toujours pas exécutée) atteint la ligne *(NNLD)*, et cette tâche a alors une laxité nulle : elle doit impérativement s'exécuter jusqu'à la fin de l'intervalle. On peut définir la *laxité nodale* d'une tâche, qui représente le temps qu'une tâche peut attendre sans manquer son échéance locale à l'intervalle, par :

$$T_i.laxity(t) = b_{k+1} - t - T_i.l(t) \;\; à\, tout\, instant\; t \tag{10.8}$$

avec$T_i.l(t)$ le temps d'exécution restant pour le noeud à l'instant t.

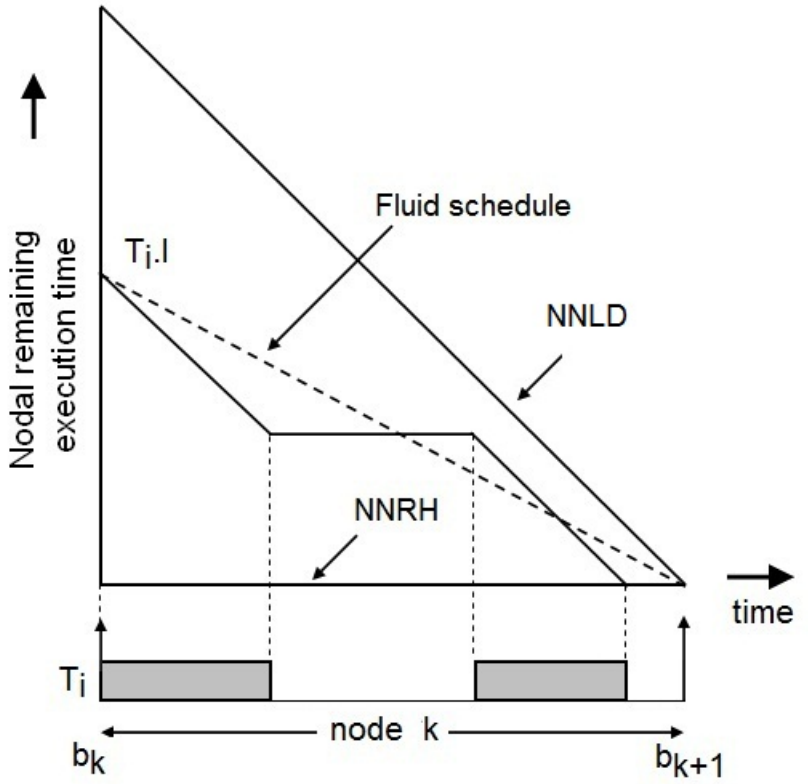


Figure 10.11: Un exemple de TN-plane

On peut noter sur la figure 10.13 l'indication de la laxité (maximale) de la tâche T_2 à l'instant b_k : $b_{k+1} - b_k - T_2.l$.

En résumé, à l'intérieur d'un intervalle il s'agit d'allouer les processeurs aux tâches en utilisant les deux événements importants que sont l'arrivée d'un jeton sur les lignes $NNRH$ (la tâche n'a plus besoin de son processeur) ou $NNLD$ (la tâche doit se voir attribuer un processeur à temps plein jusqu'à la fin de l'intervalle). Dans ces conditions on peut montrer que l'ordonnancement est faisable sur un intervalle (toutes les échéances locales ou réelles sont satisfaites à la fin de l'intervalle), et par extension que l'ordonnancement sur les *TN-planes* de l'hyperpériode est faisable [12, 36, 43].

10.3.3.1 Calcul du temps d'exécution alloué sur un intervalle

Afin d'obtenir un ordonnancement faisable sur un intervalle, les conditions suivantes doivent être satisfaites pour le calcul des temps d'exécution alloués aux jobs :

– le temps d'exécution alloué à chaque job est inférieur ou égal à la durée de

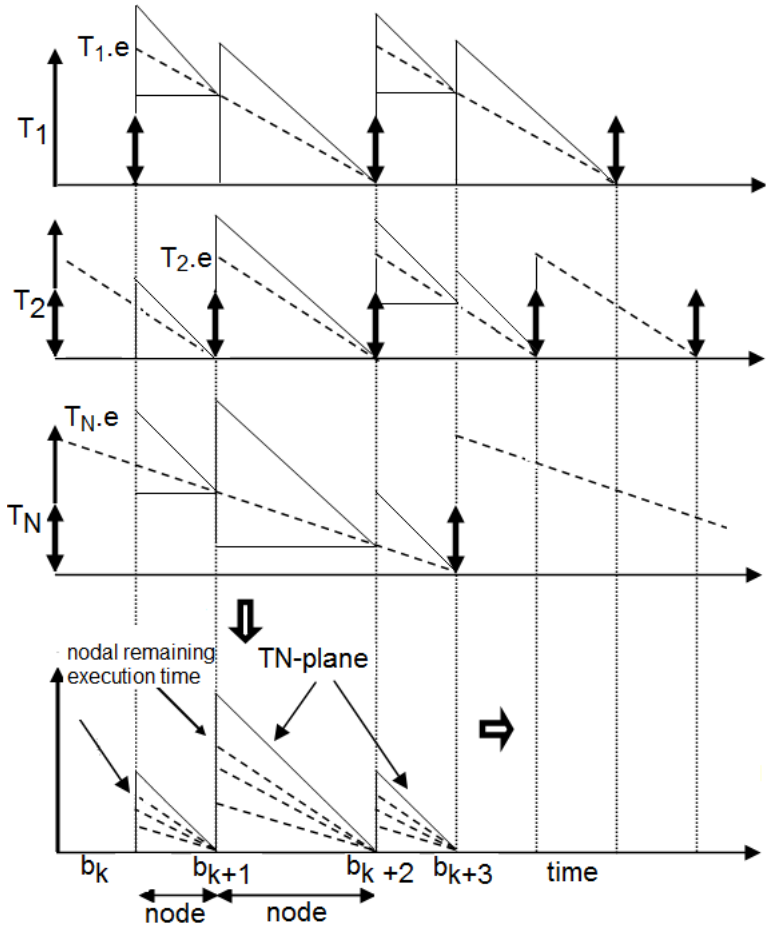


Figure 10.12: TN-planes consécutifs.

l'intervalle : $T_i.l \leq (b_{k+1} - b_k)\ \forall i\ 1 \leq i \leq N$;

- la somme des temps alloués aux jobs est inférieure ou égale à la capacité totale des processeurs de l'architecture : $\sum_{i=1}^{N}(T_i.l) \leq M * (b_{k+1} - b_k)$;
- pour n'importe quelle tâche T_i, la somme de tous les temps d'exécution $T_i.l$ alloués sur sa période est égale à son pire temps d'exécution. Si l'on note $T_i.j$ le j^{th} job de la tâche T_i, b_k la date de début d'un intervalle et b_q son échéance, alors l'équation suivante doit être satisfaite :

$$\sum_{x=k}^{q-1}(T_i.l^x) = T_i.e\ \ \forall i \tag{10.9}$$

avec $T_i.l^x$ le temps d'exécution alloué, calculé à b_x.

Toutes ces conditions étant satisfaites dans l'allocation des temps d'exécution, ainsi que la CNS d'ordonnançabilité et un ordonnancement correct à l'intérieur d'un inter-

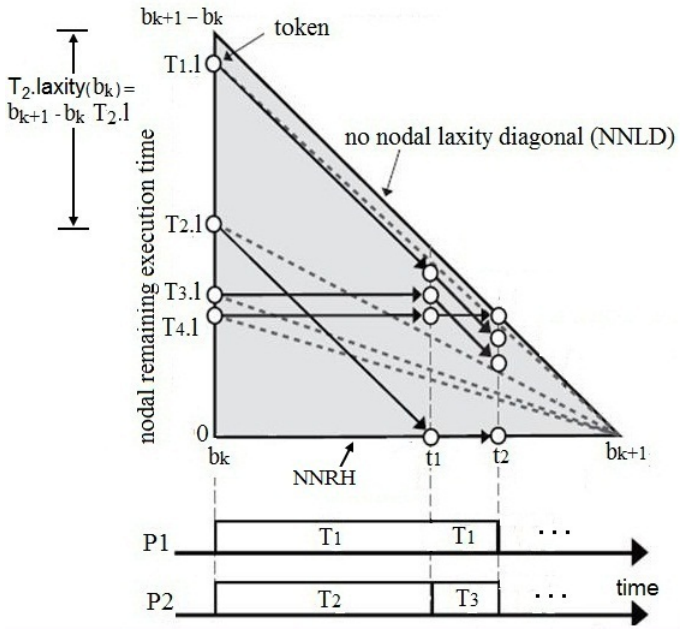


Figure 10.13: Ordonnancement à l'intérieur d'un TN plane}

valle (voir paragraphe suivant), alors les échéances des tâches de la configuration seront respectées.
Dans la version anglaise de la thèse quelques éléments sont donnés sur l'approche de Megel et al. [48] qui utilisent une technique différente pour l'allocation des temps d'exécution.

Le nombre d'unités de temps $T_i.l.$ alloué à un job de la tâcheT_i pour un intervalle est, a priori, une valeur réelle :$T_i.u$*$(b_{k+1} - b_k)$. Ceci pose évidemment un problème pratique d'implémentation où l'on ne peut raisonner qu'avec des valeurs multiples de "ticks"d'horloge du système d'exploitation support d'exécution. Une option consiste alors à calculer le nombre d'unités de temps alloué comme une valeur entière qui est soit $\lfloor (b_{k+1} - b_k) * T_i.u \rfloor$, soit $\lceil (b_{k+1} - b_k) * T_i.u \rceil$. Ceci sera évoqué plus loin.

10.3.3.2 Allocation des tâches aux processeurs dans un intervalle

L'allocation des tâches aux processeurs à l'intérieur d'un intervalle consiste à trouver un cheminement pour les jetons tel que tous atteignent la ligne *NNRH* sans traverser la ligne *NNLD*. On peut imaginer une allocation statique, donc pré-calculée, ou bien dynamique, donc calculée à l'exécution. Nous nous plaçons dans ce dernier cas. Les règles suivantes doivent alors être respectées pour l'allocation des tâches dans tout intervalle entre b_k et b_{k+1} :

– au plus *M* jobs peuvent être exécutés à tout instant ;

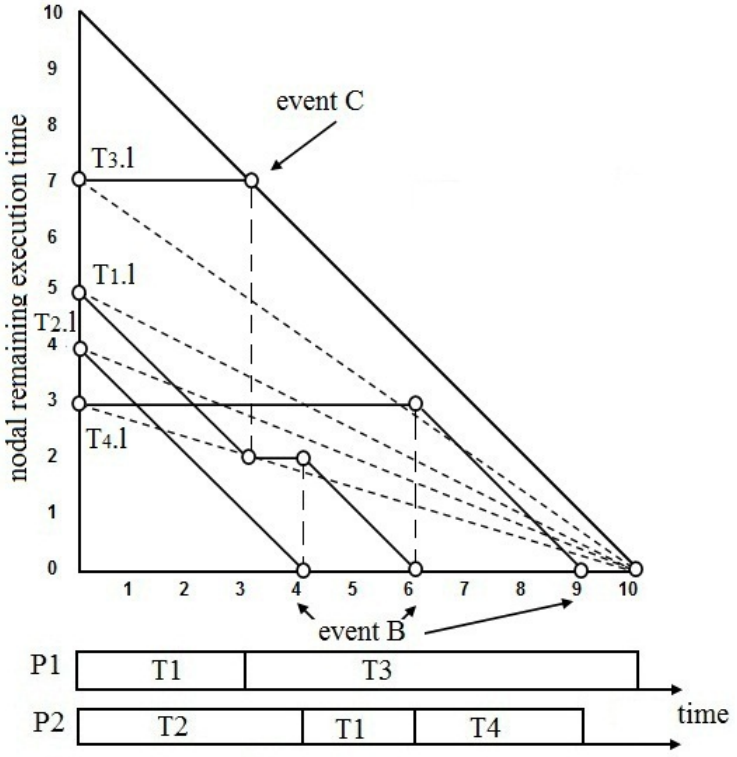


Figure 10.14: Exemple d'exécution de jobs dans un TN-Plane

– un job qui atteint la ligne *NNLD* (laxité nulle) devient prioritaire. Si aucun processeur n'est libre il devra préempter un autre job ayant une laxité non nulle. Cette situation est représentée sur la figure 10.14 par l'événement C pour T_3 ;
– quand un job a consommé les unités de temps allouées il est arrêté et son processeur est réalloué. Cette situation est représentée sur la figure 10.14 par l'événement B pour les tâchesT_2,T_1 et T_4 ;

Ces événements B et C sont appelés des points secondaires d'ordonnancement, par opposition au point primaire d'ordonnancement qu'est l'instant b_k.

On peut déjà remarquer qu'il peut exister plusieurs trajectoires satisfaisant à un ordonnancement faisable dans l'intervalle. Celui qui minimisera les surcoûts en termes de préemptions et migrations sera sûrement le plus avantageux.

En guise de résumé la figure 10.14 montre un ordonnancement possible pour la configuration donnée dans la table 10.2, avec 2 processeurs. On peut d'abord remarquer que l'ordonnancement est faisable : charge de 19 unités de temps pour une capacité de 20 unités de temps (intervalle de 10 unités de temps et 2 processeurs). Le mouvement

$Task$	$T_i.l$
T_1	5
T_2	4
T_3	7
T_4	3

Table 10.2: Configuration de 4 tâches avec $M = 2$ et $(b_{k+1} - b_k) = 10$ pour la figure 10.14

Name of algorithm	WC/ NWC	$T_i.l$ (Real / integer)	Dispatching type
BFair [Zhu et al. 2003]	NWC	Integer	Static
LLREF [Ravindran et al. 2006]	NWC	Real	Dynamic
LRE-TL [Funk et al.2009]	NWC	Real	Dynamic
DP-Wrap [G. Levin et al. 2010]	NWC	Real	Static
BFair/LRE-TL[Cho et al. 2010]	NWC	Integer	Dynamic
E-TNPA [Funakao et al.1 2008]	WC	Real	Dynamic
TRPA [Funakao et al.2 2008]	WC	Real	Dynamic

Table 10.3: Classification d'algorithmes DP-Fair

des jetons au gré des événements B et C montrent l'ordonnancement des jobs qui, comme on peut le voir sur T_1 est bien de nature non-conservative (la possibilité de faire un ordonnancement conservatif dans l'intervalle sera présentée plus loin).

10.3.4 Classification des algorithmes DP-Fair

Les algorithmes conçus selon les principes de base de *DP-Fair* peuvent être classés en plusieurs catégories selon les critères suivants :

- l'ordonnancement est fait en mode conservatif (WC) ou bien non-conservatif (NWC) ;
- le nombre d'unités de temps allouées à un job dans un intervalle est une valeur entière ou réelle ;
- l'allocation des tâches aux processeurs dans un intervalle est faite statiquement ou bien dynamiquement.

Cette possibilité de variation sur les critères est représentée dans la figure 10.15.

Les algorithmes *DP-Fair* présents dans la littérature appartiennent à l'une des catégories, ce que montre la table 10.3. On peut noter que des combinaisons sont absentes ce qui laisse la possibilité de trouver, peut-être, des algorithmes ayant des propriétés encore plus intéressantes.

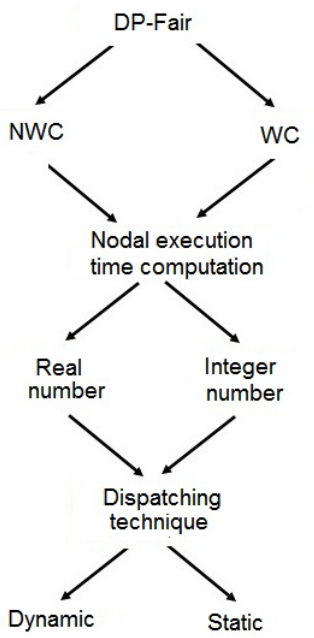


Figure 10.15: Classification des algorithme DP-Fair

10.3.5 Algorithmes avec ordonnancement non-conservatif

La partie anglaise de la thèse présente de manière assez détaillée plusieurs algorithmes. Nous nous contentons ici de les citer dans la classification proposée.

- Nombre réel d'unités de temps allouées & allocation dynamique
  - *LLREF* : Largest Local Remaining Execution time First [36] ;
  - *LRE-TL* : Local Remaining Execution TL-plane [30].
- Nombre réel d'unités de temps allouées & allocation statique
  - *DP-Wrap* [43].
- Nombre entier d'unités de temps allouées & allocation statique
  - BFair : Boundary Fair [61].
- Nombre entier d'unités de temps allouées & allocation dynamique
  - *BFair/LRE-TL* : Boundary Fair/Local Remaining Execution TL-plane [21].

10.3.6 Algorithmes conservatifs

On rappelle qu'un algorithme est dit *conservatif* s'il ne laisse jamais un processeur au repos tant qu'il y a au moins une tâche prête à s'exécuter. Bien sûr lorsque le facteur total d'utilisation est égal à la capacité des processeurs ($U = M$), il n'y a pas de différence pour ces algorithmes optimaux entre un algorithme conservatif et un algo-

rithme non-conservatif. C'est pourquoi les études ne seront pas menées dans ce cas-là mais avec un facteur total d'utilisation $U < M$. La version conservative des algorithmes basés sur le modèle *DP-Fair* opère, comme la version non-conservative en deux étapes :

– le calcul des temps d'exécution alloués aux tâches sur l'intervalle ;
– l'ordonnancement de l'exécution des tâches sur les processeurs sur l'intervalle.

Deux algorithmes vont être présentés dans cette partie : *NVNLF* et *NNLF*.

10.3.6.1 NVNLF basé sur les TN-plane étendus

NVNLF [29] est l'acronyme de « *NoVirtual Nodal Laxity Diagonal First* ». Il est sommairement présenté selon les deux étapes mentionnées précédemment.

Calcul des temps d'exécution sur l'intervalle Dans cette technique, en plus de l'allocation «conventionnelle» pour chaque tâche (proportionnelle à son facteur d'utilisation), on peut rajouter des unités temporelles inutilisées dans l'intervalle (on rappelle que le facteur total d'utilisation est inférieur à la capacité des processeurs). Ce rajout, pour une tâche, est tel qu'il ne peut être supérieur à la longueur de l'intervalle, et également au temps total restant d'exécution sur la période. Si l'on appelle $T_i.l^*$ la valeur «conventionnelle» allouée et $T_i.a$ les unités additionnelles allouées, alors le temps d'exécution alloué à une tâche T_i, entre b_k et b_{k+1} vaut :

$$T_i.l = T_i.l^* + T_i.a \tag{10.10}$$

et le temps minimum libre L' dans l'intervalle b_k et b_{k+1} est calculé par :

$$L' = (M - U) * (b_{k+1} - b_k) \tag{10.11}$$

La manière de répartir ce temps libre entre les tâches est expliqué dans l'algorithme donné dans la version anglaise. On notera simplement que le nombre d'unités allouées $T_i.a$ peut être positif, nul ou négatif. Ce dernier cas se rencontre dans la situation où la tâche s'est vu allouer trop d'unités dans les intervalles précédents, par rapport à son temps d'exécution restant sur la période. Les auteurs de cet algorithme se sont appuyés sur la représentation graphique des *TL-plane* afin d'avoir une visualisation de cette allocation d'unités temporelles, en plus ou en moins, ce qui a donné lieu aux *TN-plane* étendus. On en trouvera un exemple dans la version anglaise, ainsi qu'un exemple numérique de calcul des allocations sur un intervalle.

Règles d'ordonnancement sur l'intervalle Les tâches sont ordonnancées dynamiquement sur l'intervalle comme cela était fait dans la technique non-conservative. Ainsi :

- quand une tâche arrive à une valeur nulle pour sa laxité, elle préempte une tâche pour laquelle la laxité n'est pas nulle ; c'est l'événement *C* déjà rencontré ;
- quand une tâche a consommé tout le temps d'exécution qui lui était alloué, elle s'arrête et libère son processeur ; c'est l'événement *B* également déjà rencontré.

10.3.6.2 NNLF basé sur les TR-plane

NNLF [42] est l'acronyme de «No Nodal Laxity First». Comme précédemment, à chaque intervalle, l'algorithme se déroule en deux étapes.

Calcul des temps d'exécution sur l'intervalle Dans cet algorithme, le temps d'exécution alloué aux tâches l'est de telle manière que l'ordonnancement pratique coïncide avec l'ordonnancement fluide à l'instant de terminaison de l'intervalle. Cela est illustré sur la figure 10.16. On y trouve une période complète de la tâche depuis l'instant $T_i.r$ jusqu'à l'instant $T_i.d$, cette période étant divisée en quatre intervalles d'ordonnancement (par les autres tâches du système). Une ligne pointillée montre l'ordonnancement fluide de la tâche sur sa période à partir de $T_i.e$. On peut également observer trois valeurs $T_i.s$, une par intervalle. La valeur $T_i.s$ est calculée au début de l'intervalle par la formule :

$$T_i.s = T_i.e - T_i.u * (b_{k+1} - T_i.r) \tag{10.12}$$

Avec $T_i.r$ l'instant de réveil de la tâche T_i. Ensuite le temps d'exécution $T_i.l$ alloué à la tâche sur l'intervalle est calculé par :

$$T_i.l = max\{T_i.R - T_i.s, 0\} \tag{10.13}$$

Avec $T_i.R$ le temps total d'exécution restant pour la tâche à l'instant b_k. On pourra vérifier sur la figure 10.16 que ce temps est bien la différence entre la valeur correspondant à l'ordonnancement fluide et la valeur de $T_i.s$ en tout instant de début d'intervalle.

Règles d'ordonnancement sur l'intervalle Après le calcul des temps d'exécution alloués à chaque tâche, M tâches sont affectées aux processeurs. Pour définir les règles d'ordonnancement sur un intervalle les auteurs ont introduit la notion de tâche «sûre (safe)» et de tâche «non-sûre (unsafe)». Une tâche est dite $sûre$ quand elle a consommé le temps d'exécution qui lui était alloué, elle est dite $non - sûre$ dans le cas contraire. Les règles d'ordonnancement sur l'intervalle sont alors les suivantes. Chaque tâche allouée à un processeur s'exécute à moins que :

- la tâche ait consommé son temps d'exécution alloué ;
- un événement C se produise (laxité nulle pour une autre tâche) et c'est elle qui est préemptée, sa laxité étant non-nulle ;

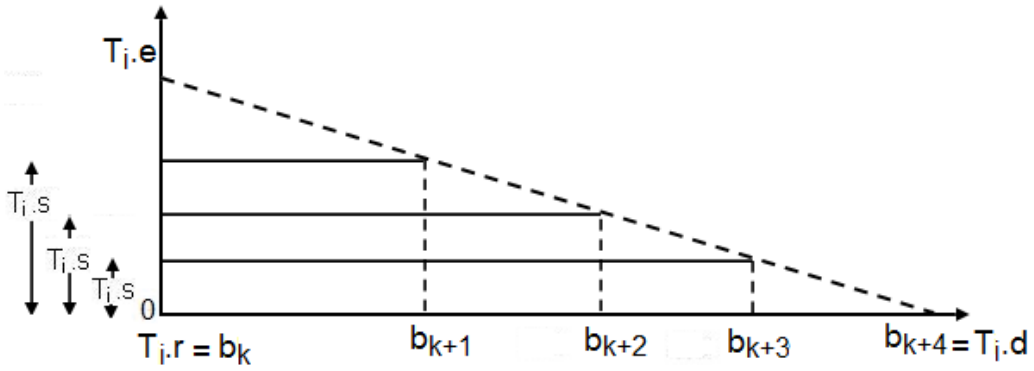


Figure 10.16: Les multiples TR-planes pour une tâche

– la somme des temps d'exécution restants sur l'intervalle pour toutes les tâches $non-s\hat{u}res$ à un instant donné (soit U_F cette somme) devienne égale à la capacité restante des processeurs à cet instant (soit t_F cet instant). C'est alors l'événement F.
Alors l'équation suivante est vraie :

$$U_F \geq M \tag{10.14}$$

Dès que l'événement F se produit toutes les tâches $s\hat{u}res$ sont arrêtées et seulement les tâches $non - sures$ peuvent être exécutées, si l'on veut respecter les échéances locales à l'intervalle.
Comme précédemment les auteurs se sont attachés à avoir une représentation visuelle de l'allocation sur un intervalle, c'est le *TR-plane* dont on trouvera une illustration dans la version anglaise.

10.4 RUN (Reduction to a UNiprocessor)

RUN est un algorithme global optimal qui a été proposé en 2011 [55]. Du fait de cette date de publication il n'a pu être expérimenté durant cette thèse. Il se distingue des algorithmes étudiés précédemment qui respectaient tous l'idée d'équité, par le fait qu'il y a à la fois du partitionnement et de l'équité ; les auteurs d'ailleurs le qualifient de «partitionned proportionate fair». Ses auteurs ont montré que cet algorithme apportait une réduction très significative du nombre de préemptions par job. Comme son nom le suggère *RUN* opère hors-ligne en réduisant le problème de l'ordonnancement multiprocesseur à une série d'ordonnancements monoprocesseur. En ligne c'est le concept de serveur qui est utilisé pour ordonnancer les tâches qui lui sont attribuées. Une présentation succincte de la manière dont l'ordonnancement est calculé, puis ensuite exécuté, est faite dans la version de référence de la thèse, en langue anglaise.

11

Techniques de maîtrise des surcoûts

Les surcoûts principaux liés à la technique d'ordonnancement global multiprocesseur sont ceux des migrations et préemptions des tâches. Ils peuvent engendrer de fortes pénalités temporelles mettant en cause le respect des échéances. L'analyse qui est proposée dans ce chapitre s'appuie sur une architecture matérielle multiprocesseur symétrique, chaque processeur étant équipé d'un cache de premier niveau (L1), le cache de deuxième niveau (L2) ou bien la mémoire principale étant communs à l'ensemble des processeurs. Le rôle des caches n'est pas étudié ici, mais il est évident que la taille du cache L1 est un élément très important pour les surcoûts engendrés par rechargement du cache lors d'une migration.

11.1 Evénements durant l'exécution d'une tâche

Les préemptions et migrations pouvant survenir lors de l'exécution d'une tâche sont mis en évidence dans la figure 11.1. Cette figure montre l'exécution d'une tâche T_i sur deux processeurs, et elle met en évidence les événements suivants :

- B est l'événement correspondant à l'activation d'un job. Deux situations sont possibles :
  - $B1$ est l'événement dans le cas où le job redémarre sur le même processeur que celui de la précédente activation ;
  - $B2$ est l'événement dans le cas où le job redémarre sur un processeur différent de celui de la précédente activation ;
- E est l'événement correspondant à la terminaison du job ;
- P est l'événement correspondant à la préemption ou la suspension du job ;
- R est l'événement qui représente la reprise d'un job. Trois situations peuvent être distinguées :
  - $R1$ est l'événement dans le cas où le job reprend sur le même processeur que celui de sa dernière exécution ;
  - $R2$ est l'événement dans le cas où le job reprend sur un processeur différent de celui de sa dernière exécution ;
  - $R3$ est l'événement dans le cas où le job continue immédiatement son exécution sur un processeur différent du précédent .

11.2 Preemption

Une tâche est préemptée lorsqu'elle est temporairement suspendue par l'ordonnanceur. Dans un contexte général ce sont par exemple les situations suivantes qui peuvent amener à cette décision :

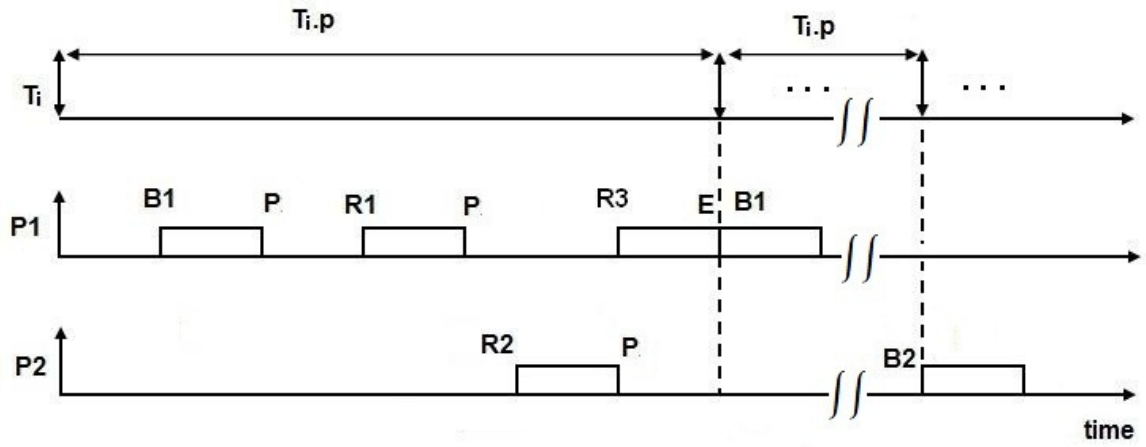


Figure 11.1: Events along a task execution

- une tâche de plus forte priorité vient d'être réveillée dans le système ;
- l'algorithme d'ordonnancement est de nature non-conservative ;
- la durée temporelle allouée à la tâche en cours (slot time) est terminée dans le cas d'un ordonnancement Round-Robin.

En général les ordonnanceurs utilisés dans les systèmes d'exploitation temps réel sont préemptifs car cela présente beaucoup d'intérêt :

- la possibilité de préempter les tâches favorise l'ordonnançabilité d'une configuration ;
- la possibilité de préempter les tâches diminue le temps de réponse des tâches les plus critiques.

Le coût d'une préemption est celui d'une sauvegarde de contexte, nécessaire pour la reprise ultérieure de la tâche. Dans la table 11.1 on donne une estimation des coûts pour chaque événement (cette table a été laissée en langue anglaise pour conserver le formatage initial).

11.3 Migration

Une migration se produit lorsqu'une tâche change de processeur au cours de son exécution. Elle est la conséquence d'une décision d'ordonnancement. On peut distinguer le changement éventuel à chaque réveil de la tâche ou bien à l'intérieur d'un job comme cela a déjà été présenté. Les coûts associés ne sont bien sûrs pas les mêmes.

Dans un cadre plus général, la migration, même si elle a cet inconvénient, présente tout de même des avantages :

- elle accroît l'ordonnançabilité d'une configuration et permet d'ordonnancer des configurations qui ne le seraient pas sans cette possibilité (un exemple a déjà été donné) ;
- la migration ;
- globalement la migration améliore les temps de réponse du système.

Les coûts associés à la migration sont estimés dans la table 11.1 (événements $R2$ et $R3$).

11.4 Estimation des surcoûts

La table 11.1 présente une estimation des coûts à partir des événements identifiés dans la figure 11.1. Pour chaque événement on décompose le coût estimé entre celui lié au contexte d'exécution, celui lié à l'exécution du service du système d'exploitation, celui de l'ordonnancement, volontairement séparé du coût du service, et enfin celui de la microarchitecture, c'est-à-dire principalement celui lié à un rechargement du cache.

Il est très difficile de calculer la valeur exacte des délais de cache car cela dépend de la séquence temporelle des tâches et donc de l'ordonnancement. C'est un sujet de recherche très actuel dans la communauté de l'analyse statique des pires temps d'exécution des programmes, qui tente de prendre en compte les effets des caches (Cache Related Preemption Delay (CRPD) [3]). Une autre approche, celle de l'analyse dynamique peut aussi permettre d'estimer ces délais sans toutefois prétendre à déterminer le pire cas (voir l'outil Harmless de l'équipe Systèmes Temps réel de l'IRCCyN [1]). Dans cette thèse nous ne pouvons pas déterminer ces délais. Aussi, bien qu'ils soient analysés, nous ne les prenons pas en compte et de ce fait les événements $R2$ et $R3$ pourraient être confondus.

Dans notre étude nous utilisons des algorithmes d'ordonnancement dirigés par le temps, même si l'intervalle entre deux activations de l'ordonnanceur est variable. Les surcoûts système doivent donc être pris en compte à chaque réveil. Cependant, comme cela a été montré dans l'étude des algorithmes, des décisions d'ordonnancement sont aussi prises entre deux instants « principaux » de réveil de l'ordonnanceur. On distingue donc ci-après ce que l'on doit faire (et donc les coûts associés) au début d'un intervalle et à l'intérieur d'un intervalle d'ordonnancement. A chaque début d'un intervalle nous devons :

- gérer les jobs (coût service RTOS et coût important d'ordonnancement) ;
- préempter certains jobs (événement P, coût d'une sauvegarde de contexte) ;
- reprendre certains jobs (événement R, coût d'une restauration de contexte).

Event type	Comment	Specific cost	RTOS cost	Micro architecture cost
B1	Release of a job on the same processor as for the previous release	Initialize context	Executed service (activateTask or Time Triggered) Scheduling	Variable (depend on the history of caches)
B2	Release of a job on a different processor as for the previous release	Initialize context	Executed service (activateTask or Time Triggered) Scheduling	Update at least first level cache for the job
E	End of a job	No cost	Executed service (terminateTask) Scheduling	No cost
P	Preemption of a job	Save context	No cost of service (preemption is the consequence of another event)	No cost
R1	Resume type 1	Restore context	No cost of service (resumption is the consequence of another event)	Variable (depend on the history of caches since the preemption of the job)
R2	Resume type 2 which is a migration	Restore context	No cost of service (resumption is the consequence of another event)	Update at least first level cache for the job
R3	Resume type 3 which is a migration	Restore context	No cost of service (resumption is the consequence of another event)	Update first level cache for the job (no misses on second level cache)

Table 11.1: Évenéments associés à l'exécution des tâches

A l'intérieur d'un intervalle nous devons :

- item gérer les jobs (coût service RTOS et coût faible d'ordonnancement) ;
- préempter certains jobs (événement P, coût d'une sauvegarde de contexte) ;
- reprendre certains jobs (événement R, coût d'une restauration de contexte).

Nous utilisons la notation cost *(# Ev)* pour désigner le surcoût dû à un nombre*(#)* d'occurrences de l'événement *Ev*.. Ainsi le surcoût total, pour une tâche T_i, sur une hyperpériode d'ordonnancement (*HP*) peut être estimé, hors coût microarchitecture, par :

$$Overhead = cost((HP/T_i.p) * (cost(B) + cost(E))) + cost(\#P) + cost(\#R) \quad (11.1)$$

Le nombre d'événements *B* et *E* est facile à déterminer sur une hyperpériode pour la tâche T_i. Par contre pour déterminer le nombre d'événements *P* et *R* de cette même tâche, nous devons les compter pour cette tâche durant la simulation de la configuration, sur une hyperpériode.

Il est évident que, pour une tâche, le nombre d'événements B et E est constant sur une hyperpériode ; on ne peut pas le réduire. Le seul moyen de réduire les surcoûts associés est de tenter de réduire, par des décisions d'allocation, le nombre d'événements de type B_2 car ce dernier est, a priori, plus couteux que l'événement de type B_1. Le nombre de préemptions et celui de migrations dépend uniquement des décisions d'ordonnancement. Il faut chercher à les réduire et en premier celui des migrations, a priori plus coûteux de par les effets de la migration sur les caches. C'est l'objet des heuristiques qui sont proposées dans cette thèse que de réduire autant que possible les préemptions et les migrations.

11.5 Heuristiques de maîtrise des surcoûts

Comme cela a été indiqué en conclusion du chapitre 3 nous utilisons pour notre modèle DP-Fair :

- la technique BFair pour le calcul des valeurs entières de temps d'exécution allouées aux tâches sur un intervalle ;
- la technique dynamique d'allocation utilisée dans *LRE-TL* pour l'allocation des tâches aux processeurs.

Notre modèle se nomme *BFair/LRE-TL*.

Les règles de calcul des temps d'exécution alloués aux tâches sur un intervalle ainsi que les règles de base d'ordonnancement des tâches sur l'intervalle, telles que définies dans les algorithmes précités laissent «de la place »pour améliorer les stratégies d'allocation des tâches et d'ordonnancement de base, afin de réduire les préemptions

et migrations, sans nuire à l'optimalité de l'algorithme de base. Nous décrivons donc ci-après des heuristiques ayant cet objet et opérant en complément de l'algorithme de base *BFair/LRE-TL*.

Les algorithmes proposés dans la littérature utilisent l'hypothèse de coût nul pour les préemptions et les migrations, et donc les règles d'allocation des tâches aux processeurs sont sans importance. L'idée de base de la première heuristique proposée est donc de faire une allocation « intelligente » des tâches sur les processeurs.

De même le choix des M tâches sélectionnées parmi les tâches prêtes pour s'exécuter n'est pas important pourvu qu'elles ne soient pas à laxité nulle et qu'elles aient encore besoin de s'exécuter [30]. L'idée de base de la deuxième heuristique qui est proposée est donc de faire une sélection « intelligente » des tâches à exécuter parmi les tâches prêtes [30].

Ces deux heuristiques ne sont pas spécifiques à *BFair/LRE-TL* ; elles peuvent être utilisées pour n'importe quel algorithme d'ordonnancement global de type *DP-Fair*.

11.5.1 Heuristique de maîtrise des migrations (*MCH*)

Cette heuristique a pour objectif de maîtriser autant que faire se peut les migrations des tâches. Elle sera répertoriée dans la suite par l'acronyme *MCH* (Migration Control Heuristic). Habituellement les tâches à exécuter sont assignées aux processeurs sans considérer leur histoire. L'heuristique *MCH* conserve la trace d'exécution des tâches, créant ainsi une relation d'affinité tâche/processeur. Ainsi, dans la mesure du possible, l'heuristique essaye d'allouer à une tâche le même processeur que celui utilisé lors de la dernière exécution. Cette heuristique est exécutée, en complément de l'algorithme de base *BFair/LRE-TL* aux points primaires d'ordonnancement, ceux qui correspondent au début d'un intervalle, ainsi qu'à chaque point secondaire d'ordonnancement, ceux qui correspondent aux événements B et C d'un *TN-Plane*. La complexité calculatoire de l'heuristique *MCH* est en O(M). Son algorithme est détaillé dans le chapitre 4 de la version anglaise de la thèse.

11.5.2 Heuristique de maîtrise des préemptions (*PCH*)

Cette heuristique a pour objectif de maîtriser autant que faire se peut les préemptions des tâches. Elle sera répertoriée dans la suite par l'acronyme *PCH* (Premption Control Heuristic). Lors de l'exécution de l'algorithme d'ordonnancement à un point primaire d'ordonnancement toutes les tâches prêtes ont la même priorité et M parmi elles peuvent être choisies arbitrairement pour s'exécuter immédiatement. L'idée de l'heuristique *PCH* est de permettre aux tâches qui étaient en exécution juste avant le point d'ordonnancement de continuer à s'exécuter, pourvu qu'elles soient effectivement prêtes. Ainsi on peut éviter des préemptions inutiles. L'heuristique *PCH* n'est utilisée

Nom	Début d'intervalle	Dans un intervalle
MCH	OUI	OUI
PCH	OUI	NON
Hybrid	PCH+MCH	MCH

Table 11.2: Activation des heuristiques

qu'aux points primaires d'ordonnancement. Sa complexité calculatoire est en O(M). Son algorithme est détaillé dans le chapitre 4 de la version anglaise de la thèse.

11.5.3 Heuristique hybride : maîtrise des migrations et des préemptions (*Hybrid*)

On peut utiliser conjointement les heuristiques *MCH* et *PCH* avec *BFair/LRE-TL*. Cette nouvelle heuristique, appelée *Hybrid*, apporte en principe les avantages des deux heuristiques en réduisant les préemptions et les migrations. En un point primaire d'ordonnancement on exécute alors l'heuristique *PCH* puis l'heuristique *MCH*. En un point secondaire d'ordonnancement on exécute seulement l'heuristique *MCH*.

La table 11.2 résume les conditions d'exécution des heuristiques selon le cas.

12

Mise en place de l'expérimentation

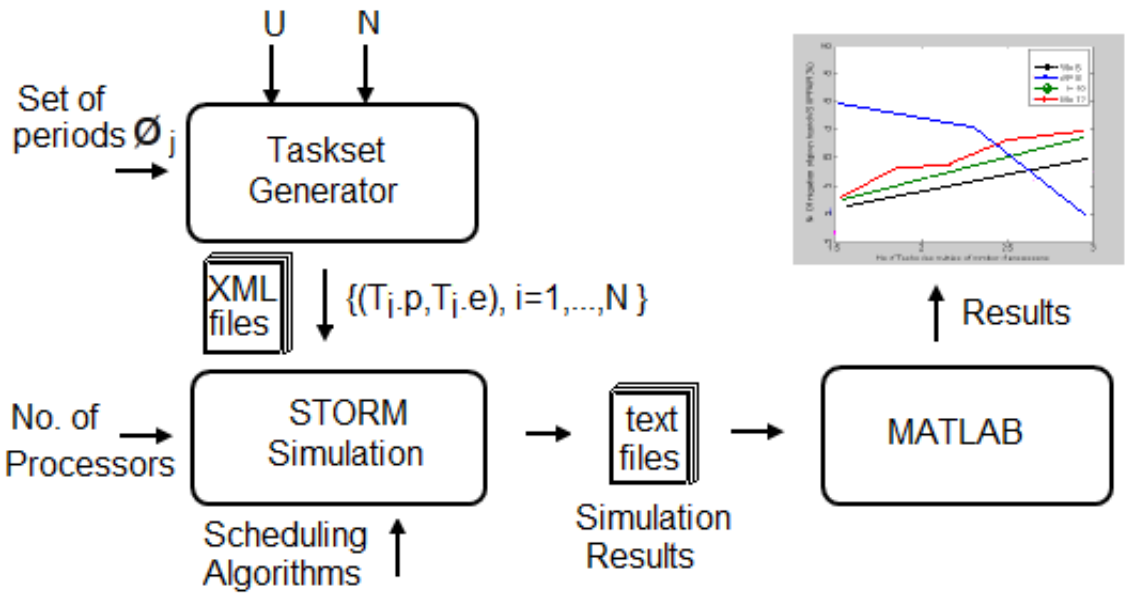


Figure 12.1: Mise en place de l'expérimentation

12.1 Introduction

L'étude des performances des heuristiques a été menée via une technique statistique. La technique expérimentale mise en oeuvre est montrée sur la figure 12.1. On y trouve un générateur de données de test fournissant les configurations à tester, élaborées à partir des données utilisateur ; un simulateur qui simule les configurations d'entrées sur l'architecture multiprocesseur, pour divers algorithmes d'ordonnancement ; un outil d'analyse et de mise en forme des résultats de simulation.

12.2 Générateur de données

Le générateur de données a reçu une attention particulière car il est essentiel dans une campagne de tests statistiques de maîtriser les configurations à tester ; il ne sert à rien de tester des milliers de configurations très proches car les résultats ne sont pas significatifs.

La figure 12.2 (a) montre le bloc diagramme du générateur de données. Les entrées, pour une configuration, sont le facteur total d'utilisation U, le nombre de tâches N et un jeu de valeurs pour les périodes des tâches ϕ_j. En sortie on obtient N couples $(T_i.e,\ T_i.p)$ tels que $\sum_{i=1}^{N}(\frac{T_i.e}{T_i.p})$ est très proche de U (mais toujours inférieure). Les valeurs des périodes des tâches et des temps d'exécution sont des entiers.
En faisant varier les valeurs des entrées on peut obtenir une grande variété de données.

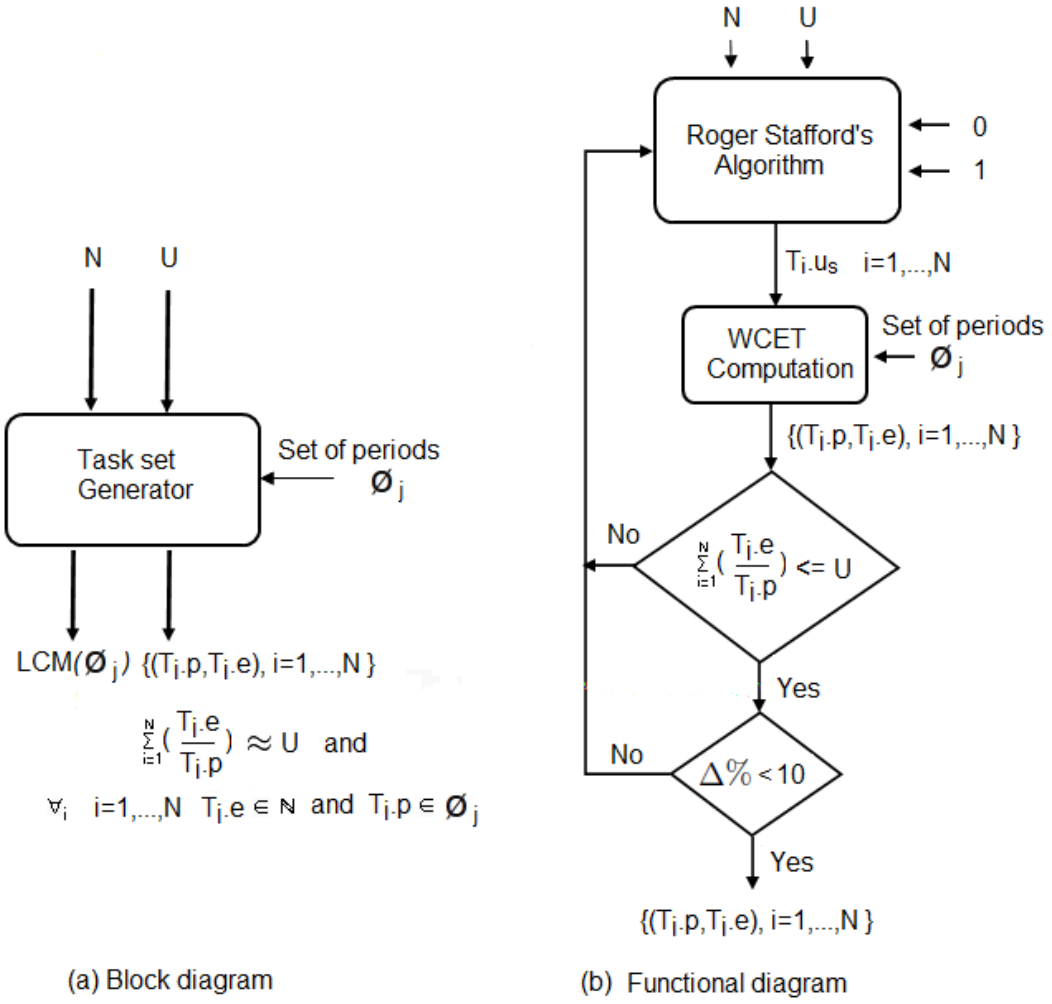


Figure 12.2: Générateur de configuration de tâches

Ainsi :

1. on peut faire varier N à U constant, ou bien faire varier U à N constant. Ainsi, selon la valeur du rapport U/N on peut obtenir des configurations de tâches lourdes ($T_i.u >= 0.5$) ou légères ($T_i.u < 0.5$) ;
2. avec plusieurs jeux de valeurs de périodes on peut obtenir des durées d'intervalles d'ordonnancement de différentes longueurs avec une distribution particulière ;
3. le nombre de processeurs n'est pas une donnée d'entrée du générateur. Ainsi une configuration générée pour une valeur donnée de N et une valeur donnée de U peut être simulée ensuite avec plusieurs valeurs de M.

Le diagramme fonctionnel du générateur de configurations est montré sur la figure 12.2 (b). L'élément central est l'utilisation de l'algorithme Randfixedsum de Roger Stafford's

[59]. Il permet de générer un nombre donné (N) de valeurs aléatoires réelles dans un intervalle [a, b], telles que leur somme soit égale à un nombre réel spécifié (U) ; ce sont donc les facteurs d'utilisation $T_i.u_s$ des N tâches qui sont générés en utilisant l'intervalle [0, 1]. Pour obtenir ensuite les temps d'exécution des N tâches nous utilisons un jeu de périodes prédéfinies ϕ_j avec une technique Round-Robin (le nombre de tâches n'est pas forcément un multiple du nombre de périodes). La valeur de $T_i.e$ est calculée par $T_i.e = T_i.u_s * T_i.p$ et elle est convertie en l'entier le plus proche. L'algorithme complet est donné dans la version anglaise de la thèse. On y détaille notamment la technique de prise en compte des erreurs successives de troncatures, et le rejet d'une configuration générée qui aurait une différence jugée excessive par rapport à la distribution d'origine.

12.2.1 Propriétés des jeux de périodes

Nous avons utilisé 4 jeux de périodes, chacun comprenant 5 valeurs. Les valeurs choisies permettent d'obtenir des intervalles d'ordonnancement variés en durée et en nombre sur l'hyper période ; elles permettent également d'obtenir une valeur de l' hyperpériode conduisant à une durée de simulation raisonnable. Les valeurs des 4 jeux sont données ci-après, les caractéristiques complètes étant détaillées dans la version anglaise de la thèse.

1. ϕ_1 = {30, 36, 40, 45, 50}; Hyperpériode = 1800 ; 168 intervalles de durées entre 2 et 30 ;
2. ϕ_2 = {30, 35, 40, 50, 100} ; Hyperpériode = 4200 ; 336 intervalles de durées entre 5 et 30 ;
3. ϕ_3 = {30, 45, 90, 150, 200}; Hyperpériode = 1800 ; 86 intervalles de durées entre 5 et 30 ;
4. ϕ_4 = {30, 35, 60, 70, 90} ; Hyperpériode = 1260 ; 72 intervalles de durée constante de valeur 12.

12.2.2 Distribution des facteurs d'utilisation des tâches

La distribution des facteurs d'utilisation des tâches à l'intérieur de la configuration est une caractéristique importante de cette dernière. On peut facilement obtenir des configurations équilibrées, ou au contraire à tâches majoritairement lourdes ou au contrainte à tâches majoritairement légères, en jouant sur le rapport entre U et N. Les graphes des figures 12.3, 12.4 et 12.5 le montrent. Chaque valeur dans le graphe représente une moyenne effectuée sur 30 configurations, chaque configuration ayant 100 tâches. A U/N=0.5 on obtient une distribution quasi uniforme entre 0 et 1 des facteurs d'utilisation ; à U/N=0.25 les tâches légères sont majoritaires alors que ce sont les tâches lourdes qui dominent à U/N=0.75.

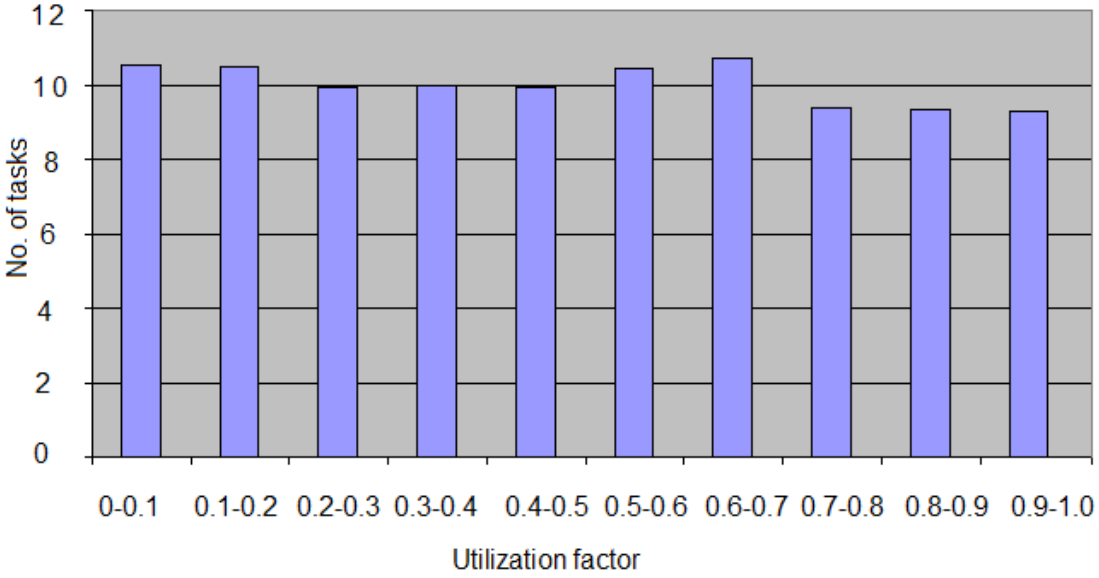


Figure 12.3: Distribution des facteurs d'utilization factor pour U/N = 0.5

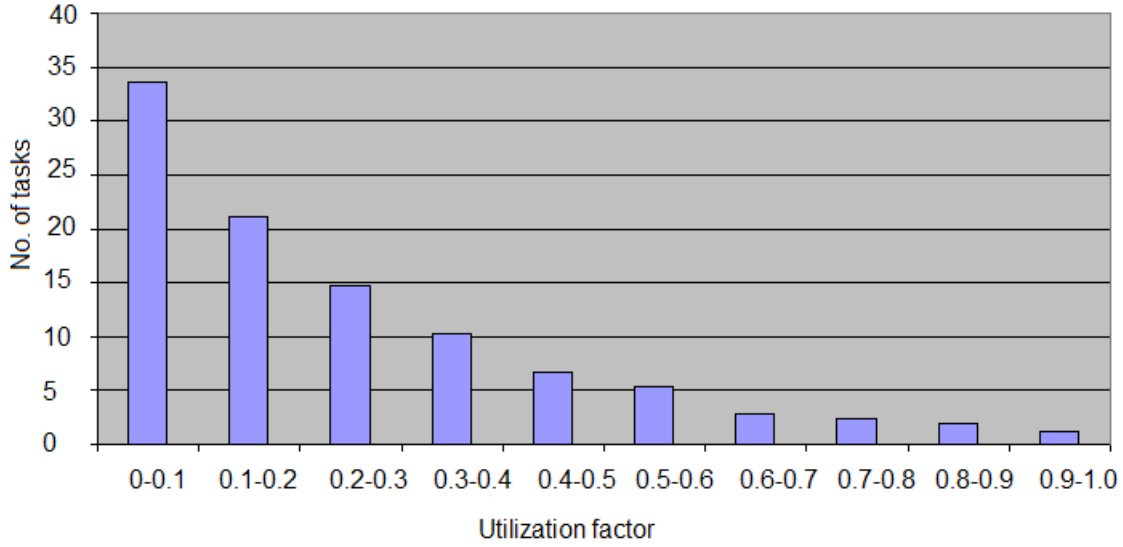


Figure 12.4: Distribution des facteurs d'utilization factor pour U/N = 0.25

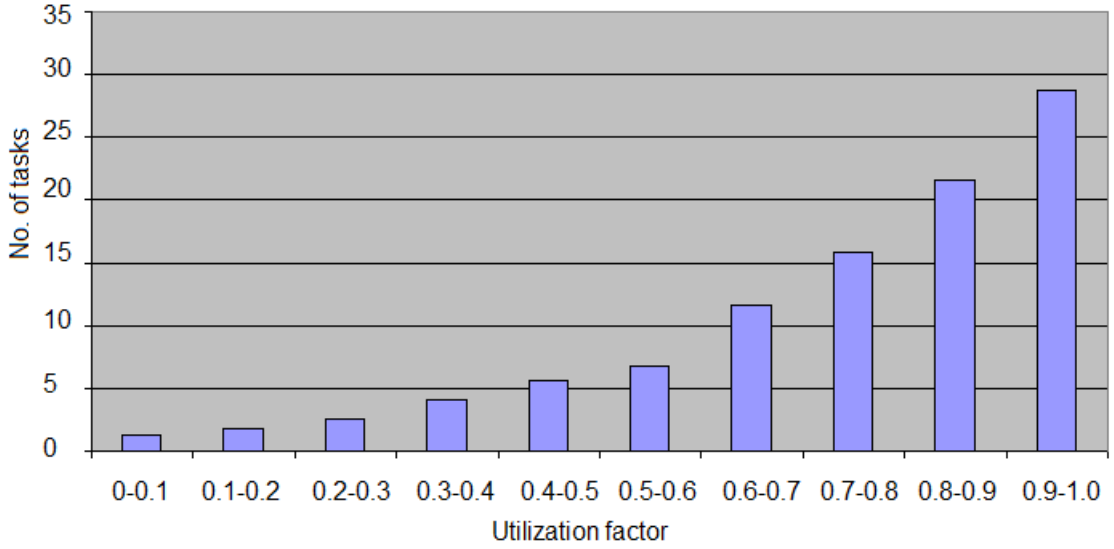


Figure 12.5: Distribution des facteurs d'utilization factor pour U/N = 0.75

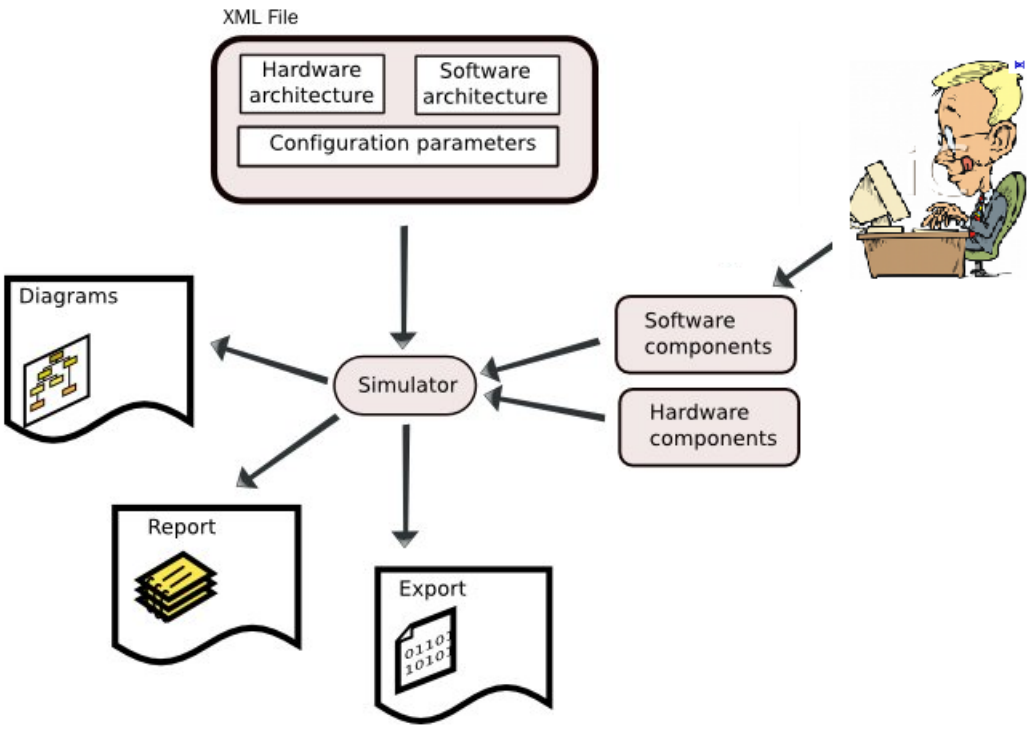


Figure 12.6: Architecture de STORM

12.3 Simulateur

12.3.1 Présentation rapide de STORM

Afin de mener à bien la phase expérimentale nous utilisons le simulateur STORM "Simulation TOol for Real time Multiprocessor scheduling", [2, 60], (http://storm.rts-software.org) développé dans l'équipe Systèmes Temps réel de l'IRCCyN. Il permet de simuler le comportement temporel d'une configuration sur une architecture multiprocesseur symétrique et pour un algorithme d'ordonnancement. Durant la simulation un ensemble de grandeurs peuvent être capturées permettant de construire, en ligne ou hors-ligne, des métriques de caractérisation du comportement. La vue générale de STORM est donnée à la figure 12.6.

Comme on le voit sur cette figure STORM prend en entrée un fichier XML définissant l'ensemble des caractéristiques de la configuration à simuler et analyser, Un exemple de fichier est montré sur la figure 12.7. On y trouve :

- durée de la simulation ;
- type d'ordonnanceur ;

```
<SIMULATION duration="840">
<SCHED name="sched" className="storm.Schedulers. xxx .SchedulingAlgoName"
<CPUS>
 <CPU className="storm.Processors.CT11MPCore" name="CPU A" id="1" />
 <CPU className="storm.Processors.CT11MPCore" name="CPU B" id="2" />
</CPUS>
<TASKS>
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T1" id="3" period="8" deadline="8" activationDate="0" WCET="4" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T2" id="4" period="12" deadline="12" activationDate="0" WCET="3" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T3" id="5" period="15" deadline="15" activationDate="0" WCET="10" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T4" id="6" period="12" deadline="12" activationDate="0" WCET="1" />
 <TASK className="storm.Tasks.PTask_WAM" name="PTASK T5" id="7" period="14" deadline="14" activationDate="0" WCET="7" />
</TASKS>
</SIMULATION>
```

Figure 12.7: Exemple de fichier d'entrée pour STORM

- nombre et type des processeurs ;
- nombre et types des tâches ;
- paramètres de chaque tâche (nom, période, échéance, durée d'exécution.

STORM propose en standard un bon nombre d'ordonnanceurs mais il est possible d'en ajouter, ce que nous avons fait pour l'analyse des algorithmes traités dans cette thèse. Un tel ordonnanceur est écrit en langage Java et il s'interface avec STORM au travers d'une API. Les résultats de simulation peuvent prendre plusieurs formes : diagrammes de Gantt pour les tâches et les processeurs, diagrammes de puissance, fichiers texte de caractéristiques enregistrés au cours de la simulation et destinés à être traités par un outil approprié. Pour plus de détails le lecteur est invité à se reporter à la documentation du simulateur. (http://storm.rts-software.org).

12.3.2 Algorithmes implémentés durant la thèse

Algorithmes d'ordonnancement

Les algorithmes d'ordonnancement suivants ont été implémentés pour STORM :

1. *DP-Fair/LRE-TL* (non conservatif)
2. *DP-Fair/LRE-TL* (conservatif)
3. *DP-Fair/LRE-TL* + *MCH* (Migration control heuristic)
4. *DP-Fair/LRE-TL* + *PCH* (Preemption control heuristic)
5. *DP-Fair/LRE-TL* + *PCH* + *MCH* (*Hybrid*)

Observateur des surcoûts

Un autre programme, *« l'observateur »*, interfacé à STORM via l'API générale,

a également été écrit pour permettre la mesure d'événements spécifiques, donc non mesurés directement par STORM, lors de la simulation. Son algorithme est présenté dans la version anglaise de la thèse.

12.4 Processus d'expérimentation

Etape 1: espace d'exploration des données

A partir de la variation des paramètres d'entrée du générateur nous avons engendré un ensemble de configurations destinées à être ensuite simulées.

Pour chaque jeu de périodes ϕ_j :

1. l'expérimentation est conduite avec un facteur total d'utilisation variable (U=M, U=0.75M and U= 0.5M) ;
2. pour chaque valeur du facteur total d'utilisation, les expérimentations sont faites avec un nombre variable de processeurs (M=2, 4, 6, 8 10 et 12) ;
3. pour valeur de M, 4 valeurs différentes du nombre de tâches N sont utilisées (N = 1.5M, N = 2M, N = 2.5M and N = 3M) ;
4. pour chaque valeur de N, 30 configurations sont utilisées.

En résumé l'arbre général des données générées est donné à la figure 12.8.

1. pour un quadruplet $\{\phi_i, \frac{U}{M}, M, N\}$, 30 configurations sont générées ;
2. pour le triplet $\{\phi_i, \frac{U}{M}, M\}$, 120 configurations sont générées avec les variations sur N ;
3. pour un doublet $\{\phi_i, \frac{U}{M}\}$, 720 configurations sont générées avec les variations sur N et M ;
4. pour un jeu de périodes $\{\phi_i\}$, 2160 configurations sont générées avec les variations sur N, M et $\frac{U}{M}$;
5. au total nous générons 8640 configurations.

Etape 2: la simulation

Les 8640 fichiers de configurations différentes sont simulés avec STORM pour chacun des algorithmes d'ordonnancement. A chaque groupe de 30 configurations caractérisées par le quadruplet $\{\phi_i, \frac{U}{M}, M, N\}$, on associe un unique fichier de résultats.

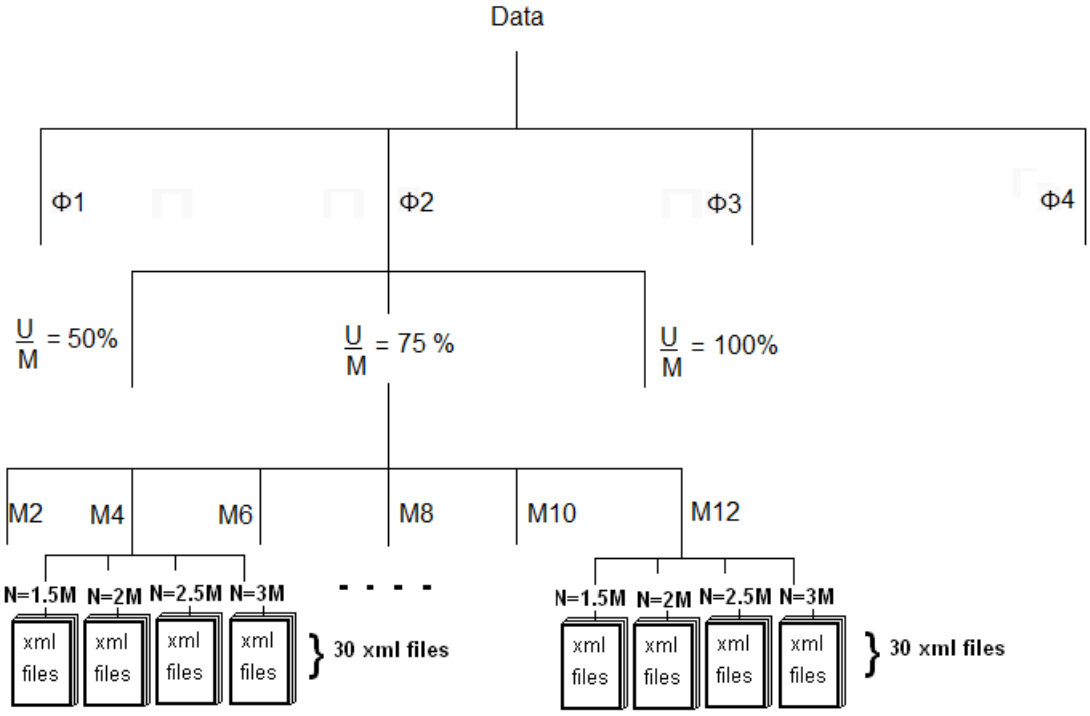


Figure 12.8: Arbre de génération des données

Etape 3: Analyse des résultats expérimentaux

Les résultats bruts de simulation ont été traités par un programme Matlab, et différentes métriques ont été élaborées conduisant au tracé de graphes divers :

1. maîtrise des surcoûts des différents algorithmes pour des valeurs variables du rapport U/M pour une valeur donnée de N ;

2. maîtrise des surcoûts de l'algorithme «Hybrid » pour des valeurs variables de U et M pour une valeur donnée de N ;

3. maîtrise des surcoûts de l'algorithme «Hybrid » pour une valeur variable du rapport N/M pour une valeur donnée de U ;

4. maîtrise des surcoûts de l'algorithme "Hybri" pour une valeur variable du rapport U/N, pour une valeur donnée de U, comme dans la table 12.1.

U/M	1				0.75				0.5			
N/M	1.5	2	2.5	3	1.5	2	2.5	3	1.5	2	2.5	3
U/N	0.67	0.5	0.4	0.33	0.5	0.38	0.3	0.3	0.33	0.25	0.2	0.17

Table 12.1: Données générées en function de U/N

13

Résultats expérimentaux

On présente dans ce chapitre les résultats obtenus en suivant le processus expérimental présenté au chapitre précédent. C'est le programme *observer* qui mesure les nombres de migrations et préemptions lors de la simulation d'une configuration sur une durée égale à l'hyperpériode. Afin de pouvoir calculer l'efficacité des heuristiques les mesures sont effectuées, pour la même configuration, d'une part avec l'algorithme *DP-Fair* qui va servir de référence, et d'autre part avec nos heuristiques. Les résultats sont ensuite formatés en fonction de différentes variables.

De très nombreux graphes ont été construits pour illustrer les résultats obtenus et il est sans intérêt de les reproduire deux fois dans le mémoire. C'est pourquoi nous ne les mettons pas dans cette version simplifiée en français, mais seulement, avec les commentaires associés, dans la version anglaise qui est la référence du travail (se référer au chapitre 6).

13.1 Résultats avec la version non-conservative des algorithmes (NWC)

Dans ce cas c'est l'algorithme *BFair/LRE-TL* qui sert de référence pour les versions incluant les heuristiques PCH (Preemption Control heuristic) et MCH (Migration Control Heuristic). Ainsi, au total, les mesures ont été faites sur quatre algorithmes :

- *BFair/LRE-TL*
- *BFair/LRE-TL + PCH = PCH*

- *BFair/LRE-TL + MCH = MCH*
- *BFair/LRE-TL + MCH + PCH = Hybrid-NWC*

Les résultats suivants ont été calculés et présentés sous forme de graphes :

13.1.1 Maîtrise des migrations (NWC)

- Résultats pour les différents algorithmes à *U=M, U=0,75M, U=0,5M* ;
- Résultats pour l'algorithme hybride avec différents jeux de périodes ;
- Résultats pour l'algorithme hybride avec un rapport variable *N/M* ;
- Résultats pour l'algorithme hybride en fonction de *U/N*.

13.1.2 Maîtrise des préemptions (NWC)

- Résultats pour les différents algorithmes à *U=M, U=0,75M, U=0,5M* ;
- Résultats pour l'algorithme hybride avec différents jeux de périodes ;
- Résultats pour l'algorithme hybride avec un rapport variable *N/M* ;
- Résultats pour l'algorithme hybride en fonction de *U/N*.

Pour conforter la validité des résultats un exemple de mesure d'écart type pour l'algorithme *Hybrid* est également fourni.

13.2 Résultats avec la version conservative des algorithmes (WC)

Pour une charge *U=M* il n'y a bien sûr pas de différence entre les résultats des versions non-conservative et conservative. Ce sont donc seulement les cas avec *U=0.75M* and *U=0.5M* qui ont été expérimentés. Dans ce cas c'est toujours l'algorithme *BFair/LRE-TL* qui nous sert de référence pour les versions incluant les heuristiques PCH et MCH. Ainsi, au total, les mesures ont été faites sur trois algorithmes :

- *BFair/LRE-TL (NWC)*
- *BFair-NNLF/LRE-TL= NNLF*
- *BFair-NNLF/LRE-TL+ PCH + MCH = Hybrid-WC*

Les résultats suivants ont été calculés et présentés sous forme de graphes :

13.2.1 Maîtrise des migrations (WC)

- Résultats pour les différents algorithmes à *U=0,75M*, *U=0,5M* ;

Item	Migrations			Préemptions		
U/M	1	0.75	0.5	1	0.75	0.5
Hybrid-NWC	52 %	47 %	47 %	65 %	94 %	98 %
Hybrid-WC	NA	24 %	15 %	NA	9 %	2 %

Table 13.1: Maîtrise des surcoûts par rapport à BFair/LRE-TL

– Résultats pour l'algorithme hybride avec différents jeux de périodes ;
– Résultats pour l'algorithme hybride avec un rapport variable *N/M* ;
– Résultats pour l'algorithme hybride en fonction de *U/N*.

13.2.2 Maîtrise des préemptions (WC)

– Résultats pour les différents algorithmes à *U=0,75M, U=0,5M* ;
– Résultats pour l'algorithme hybride avec différents jeux de périodes ;
– Résultats pour l'algorithme hybride avec un rapport variable *N/M* ;
– Résultats pour l'algorithme hybride en fonction de *U/N*.

Comme précédemment, pour conforter la validité des résultats un exemple de mesure d'écart type pour l'algorithme *Hybrid* est fourni.

13.3 Conclusion

On peut obtenir une vue d'ensemble des résultats obtenus dans la table 13.1. Les résultats sont donnés pour trois valeurs de *U/M*. Les chiffres indiqués représentent le pourcentage du nombre de migrations ou préemptions restant après application des heuristiques, par rapport au nombre engendré par l'algorithme de référence. Par exemple, dans le cas non-conservatif, l'algorithme *hybrid-NWC* permet d'obtenir 52% des migrations de l'algorithme de référence à *U/M=1* ; il apporte donc une décroissance de 48% sur ce nombre de migrations.

14

Conclusion générale

La problématique générale de la thèse était la maîtrise des surcoûts engendrés par les algorithmes globaux optimaux pour l'ordonnancement de systèmes temps réel multiprocesseur, pour un jeu de tâches périodiques à échéances implicites.

L'inconvénient majeur de ces algorithmes est la présence de très nombreux points d'ordonnancement, ce qui engendre des surcoûts à l'exécution et peut donc compromettre l'ordonnançabilité en pratique. Afin de réduire les surcoûts, c'est-à-dire pour nous les préemptions et les migrations, nous avons proposé un ensemble d'heuristiques simples ne remettant pas en cause l'optimalité de l'algorithme de base. Afin d'évaluer leurs performances nous avons réalisé une étude statistique sur des métriques caractérisant cet ordonnancement, aussi bien dans le cas non-conservatif que dans le cas conservatif.

Les conclusions générales que l'on peut tirer de ces travaux sont les suivantes.

Classification des algorithmes DP-Fair L'analyse des principes des algorithmes optimaux globaux pour architecture multiprocesseur symétrique, notamment ceux basés sur le partitionnement des échéances, nous a amené à proposer une classification fondée sur la manière dont ils procèdent pour l'allocation des durées d'exécution aux tâches prêtes pour un intervalle, ainsi que sur la manière dont ils organisent l'exécution des tâches prêtes à l'intérieur d'un intervalle. Ceci a mis en évidence des combinaisons, actuellement inutilisées, dont il faudrait analyser la pertinence.

Les heuristiques de maîtrise des surcoûts réduisent efficacement le nombre de préemptions et de migrations La baisse des surcoûts est très significative par rapport à l'algorithme de référence *BFair/LRE-TL*. Ceci est vrai en faisant varier le facteur total d'utilisation des configurations, les périodes des tâches, la nature de la configuration (tâches majoritairement légères, tâches équilibrées, tâches majoritairement lourdes) et le nombre de processeurs. La réduction est observée aussi bien dans le cas d'un algorithme non-conservatif que dans le cas d'un algorithme conservatif, ce dernier cas engendrant par nature moins de préemptions.

Extension de ces travaux Une principale extension envisageable pour ces travaux serait assez naturellement l'expérimentation des heuristiques sur une architecture réelle. Nous disposons en effet, dans notre équipe de recherche, d'un système d'exploitation temps réel open source (Trampoline) pour lequel une politique d'ordonnancement global est en cours d'expérimentation, dans le cadre du projet ANR RESPECTED. Ce serait donc l'occasion de vérifier l'efficacité de nos heuristiques en exécution.

Par ailleurs, même si l'on peut imaginer que les coûts des migrations sont très supérieurs au coût des préemptions, il serait très intéressant de pouvoir observer le comportement réel des caches lors de l'exécution et de mesurer les surcoûts réel engendrés (surcoûts système et surcoûts des préemptions et migrations). Même si une telle analyse sur cible ne peut prétendre à l'exhaustivité, elle peut être une contribution intéressante à la maîtrise des *CRPD* (Cache related Preemption Delays).

Bibliography

[1] harmless.rts-software.org. 89, 190

[2] http://storm.rts-software.org. 49, 106, 169, 200

[3] Sebastian Altmeyer and Claire Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57:707–719, 2010. 89, 190

[4] J. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real-time tasks on multiprocessors. 2005. 46

[5] J. Anderson and A. Srinivasan. Early-release fair scheduling. *In the Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000. 38, 46

[6] James Anderson and Anand Srinivasan. Pfair scheduling: Beyond periodic task systems. In *In Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000. 55, 173

[7] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA '00, pages 337–, Washington, DC, USA, 2000. IEEE Computer Society. 37

[8] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 33–40. IEEE, 2003. 37

[9] Björn Andersson, Sanjoy K. Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 193–202, 2001. 36, 37, 161

[10] Bjorn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 322–334, Washington, DC, USA, 2006. IEEE Computer Society. 39, 162

[11] N. Audsley and A. Burns. Real-time system scheduling. Technical report, Department of Computer Science, University of York, UK., 1990. 28

[12] K. Bletsas B. Andersson and S. K. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. *In the Proceedings of the Real-Time Systems Symposium*, pages 385–394, 2008. 35, 61, 160, 177

[13] Theodore P. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16:760–768, 2005. 37

[14] D. C. Blest and D. G. Fitzgerald. Scheduling sports competitions with a given distribution of times. *Discrete Applied Mathematics*, 22(1):9–19, 1988. 27, 156

[15] Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society. 38

[16] Almut Burchard, Jörg Liebeherr, Yingfeng Oh, and Sang H. Son. New strategies for assigning realtime tasks to multiprocessor systems. *IEEE Transaction on Computers*, 44(12):1429–1442, 1995. 33, 34, 159

[17] G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, 2004. 26, 155

[18] John M. Cal, Hennadiy Leontyev, Aaron Block, Umamaheswari C. Devi, and James H. Anderson. Litmus rt: A testbed for empirically comparing real-time multiprocessor schedulers ? *In the Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126, 2006. 52, 171

[19] Abhishek Chandra, Micah Adler, and Prashant Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. *In the Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–14, 2001. 49, 169

[20] Steve J. Chapin. Distributed and multiprocessor scheduling. *ACM Computing Surveys*, 1996. 38

[21] H. Cho, Binoy Ravindran, and E. Douglas Jensen. T-l plane-based real-time scheduling for homogeneous multiprocessors. *Journal of Parallel and Distributed Computing*, 70:225 – 236, 2010. 70, 182

[22] S. Davari and S.K Dhall. On a periodic real time task allocation problem. *Annual International Conference on System Sciences*, 1986. 33

[23] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. *Research report YCS-2009 University of York*, 2009. 53

[24] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974. 27, 28, 156, 157

[25] U.C. Devi and James H. Anderson. Desynchronized pfair scheduling on multiprocessors. page 332, 2005. 52

[26] Umamaheswari C. Devi and James H. Anderson. A schedulable utilization bound for the multiprocessor epdf pfair algorithm. *Real-Time Systems*, 38:237–288, 2008. 52

[27] S K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operation research*, 26:127–140, 1978. 16, 28, 30, 33, 34, 157, 159

[28] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Energy-efficient optimal real-time scheduling on multiprocessors. *In the Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 23–30, 2008. 58, 175

[29] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. *In the Proceedings of the 20th Euromicro Conference on Real-Time Systems ECRTS. Barcelona, Spain*, pages 13–22, 2008. 4, 71, 72, 73, 82, 183

[30] S. Funk and Vijaykant Nadadur. Lre-tl an optimal multiprocessing scheduling algorithm for sporadic task sets. *In the Proceedings of the 17th International Conferenece of Real-Time and Network systems, Paris*, pages 159–168, 2009. 38, 57, 65, 66, 67, 82, 91, 161, 182, 193

[31] Shelby Funk. Lre-tl: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. *Real-Time Syst.*, 46(3):332–359, December 2010. 66

[32] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. *In the Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 183–192, 2001. 17, 31, 158

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. 33, 34, 159

[34] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, September 2003. 37

[35] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on applied mathematics*, 17(2):416–429, 1969. 29

[36] B. Ravindran H. Cho and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *In the Proceedings of the IEEE International Real-Time Systems Symposium*, pages 101–110, 2006. 38, 52, 54, 61, 65, 66, 161, 172, 177, 182

[37] P. Holman and J. Anderson. The staggered model: Improving the practicality of pfair scheduling. *In the Proceedings of the 24th IEEE International Real-Time Systems symposium(RTSS'03)*, 2003. 9, 11, 51, 54, 55, 56, 171, 172, 174, 175

[38] Philip Holman and James H. Anderson. Group-based pfair scheduling. *Real-Time Systems*, 32:125–168, 2006. 57

[39] L. George J. Goossens I. Lupu, P. Courbin. Multi-criteria evaluation of partitioning schemes for real-time systems. *In the Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*, pages 1–8, 2010. 34

[40] J.L. Diaz D.F. Garcia J.M. Lopez, M. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, pages 5–28, 2003. 34

[41] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. pages 139–148, 2008. 39, 162

[42] Shinpei Kato Kenji Funakao and NobuyukiYamasaki. New abstraction for the optimal real time scheduling on multiprocessor. *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 357–364, 2008. 75, 184

[43] G. Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. *In the Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 3–13, 2010. 38, 42, 57, 61, 67, 161, 164, 174, 177, 182

[44] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. 23, 27, 28, 152, 156, 157

[45] C. L. Liu and James W. Layland. Readings in hardware/software co-design. pages 179–194, 2002. 156

[46] C.L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 37-60:28–31, 1969. 28

[47] R. McNaughton. Scheduling with deadlines and loss functions. *Management Sciences*, pages 1–12, 1959. 70

[48] Thomas Megel, Renaud Sirdey, and Vincent David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. *In the Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 37–46, 2010. 62, 179

[49] Mark Moir and Srikanth Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 294–303, Washington, DC, USA, 1999. IEEE Computer Society. 57, 173

[50] Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard–real–time environment. Technical Report MIT-LCS-TR-297, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1983. Ph.D. Thesis. 27, 28, 156, 157

[51] Yingfeng Oh, Yingfeng Oh, Sang H. Son, and Sang H. Son. Tight performance bounds of heuristics for a real-time scheduling problem. Technical report, 1993. 33, 34, 159

[52] Yingfeng Oh and Sang H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical report, Department of Computer Science, University of Virginia, 1995. 33, 34, 159

[53] J.H Anderson P Holman. Adapting pfair scheduling for symmetric multiprocessors. *Journal of embedded computing IOS press*, 1, Nov 2005. 54, 172

[54] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *STOC*, pages 140–149, 1997. 17, 31, 158

[55] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott A. Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS*, pages 104–115, 2011. 42, 78, 81, 82, 83, 164, 185

[56] N. C.G.Plaxton S.Baruah and D.Varvel. Proportionate progress:a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996. 17, 31, 37, 38, 42, 46, 48, 66, 158, 161, 164, 165, 166, 169

[57] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *In the Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, 2001. 49, 55, 56, 171, 173

[58] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, 84(2):93–98, October 2002. 37

[59] R. Stafford. Random vectors with fixed sum. *http://www.mathworks.com/matlabcentral/fileexchange/9700*, 2006. 98, 198

[60] R. Urunuela, A-M. Déplanche, and Y. Trinquet. Storm - a simulation tool for real-time multiprocessor scheduling evaluation. *In the Proceedings of IEEE International Conference on Emerging Technology and Factory Automation, ETFA, Bilbao*, pages 1–8, 2010. 49, 106, 169, 200

[61] D. Zhu, Daniel Mosse, and Rami Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary? *In the Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 142–151, 2003. 38, 57, 69, 82, 91, 161, 182

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

TITRE EN FRANÇAIS :

Maîtrise des surcoûts d'exécution de politiques d'ordonnancement global pour les systèmes temps réel multiprocesseur

Résumé et mots-clés en français :

Systèmes temps réel, ordonnancement global et optimal multiprocesseur, maîtrise des surcoûts, simulation.

En théorie, les algorithmes optimaux d'ordonnancement global permettent d'obtenir une meilleure utilisation des ressources processeur que les algorithmes d'ordonnancement partitionnés, mais pratiquement ils sont considérés comme inférieurs, car ils provoquent une grande quantité de surcoûts d'exécution. Cette surcharge est due à des points d'ordonnancement fréquents, ainsi que les migrations et les préemptions pour les tâches. Dans cette thèse, nous avons choisi une classe d'ordonnancement optimal connu sous le nom de DP-Fair et nous avons mis au point quelques techniques pour maîtriser la surcharge sans affecter l'optimalité. Nous avons proposé deux heuristiques, une contrôle le nombre de préemptions et la seconde le nombre de migrations. Nous avons utilisé une approche statistique pour évaluer la performance de nos heuristiques. Les résultats obtenus sont très encourageants et montrent une réduction significative de la surcharge.

TITRE EN ANGLAIS :

Overhead control in optimal scheduling algorithms for real-time multiprocessor systems

Résumé et mots-clés en anglais :

Real-time systems, global multiprocessor scheduling, overhead control, simulation

Theoretically the optimal global scheduling algorithms achieve higher utilization of processors resource than partitioned scheduling algorithms but practically they are considered inferior because they incur a large amount of overhead. This overhead constitutes frequent scheduling points, migrations and preemptions. In this thesis we have chosen an optimal scheduling class known as DP-Fair and have devised a few techniques to control the overhead without affecting the optimality. We have proposed two heuristics, one controls the number of preemptions and the second controls the number of migrations. We have used a statistical approach to evaluate the performance of our heuristics. The results obtained are very encouraging and show a significant reduction in the overhead.

L'UNIVERSITÉ NANTES ANGERS LE MANS