FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time Scheduling on Heterogeneous Multiprocessors

Gurulingesh Raravi

Doctoral Programme in Electrical and Computer Engineering

Supervisor: Dr. Vincent Nélis

March 11, 2014

© Gurulingesh Raravi, 2014

Real-Time Scheduling on Heterogeneous Multiprocessors

Gurulingesh Raravi

Doctoral Programme in Electrical and Computer Engineering

March 11, 2014

Abstract

Embedded computing is one of the most important areas in computer science today, witnessed by the fact that 98% of all computers are embedded. Given that many embedded systems have to interact "promptly" with their physical environment, the scientific community has invested significant efforts in developing algorithms for scheduling the workload, which is generally implemented as a set of tasks, at the *right time* and in proving before run-time that all the timing requirements will be satisfied at run-time. This field of study is referred to as the *real-time scheduling theory*.

The scheduling theory for a unicore processor is well-developed; the scientific results are taught at all major universities world-wide and the results are widely-used in industry. Scheduling theory for multicores is emerging but the focus so far has been for multicores with identical processing units. This is unfortunate because the computer industry is moving towards *heterogeneous multicores* with a constant number of distinct processor types — AMD Fusion, Intel Atom and NVIDIA Tegra are some of the examples of such multicores.

This work deals with the problem of scheduling a set of tasks to meet their deadlines on *heterogeneous multiprocessors* with a constant number of distinct processor types. On heterogeneous multiprocessors, not all the processors are of the same type and further, the execution time of a task depends on the type of processor on which it executes. For such platforms, designing scheduling algorithms assuming that, tasks during their execution, can migrate between *processors of different types* is hard to achieve (if not impossible) for many practical systems since processors of different types typically vary in instructions sets, register formats, etc. Hence, designing algorithms in which either tasks cannot migrate between any two processors (referred to as *non-migrative*) or tasks can migrate between *processors of same type* (referred to as *intra-migrative*) is of greater practical significance.

For the *non-migrative* scenario, the scheduling problem can be solved in two steps: (i) assigning *tasks to individual processors* before run-time and (ii) once the assignment is done, scheduling the tasks on each processor using a uniprocessor scheduling algorithm at run-time. Scheduling tasks that are assigned to an individual processor is a well-studied problem and *optimal scheduling algorithms* exist for this problem, in the sense that, if there exists a schedule that meets all deadlines of the tasks then the optimal algorithms succeed in finding such a schedule as well. Hence, assuming that the tasks are scheduled during run-time on each processor using such optimal scheduling algorithms, this work focuses on designing algorithms for assigning tasks to individual processors on heterogeneous multiprocessors with a constant number of distinct processor types such that there exists a schedule that meets all deadlines for the assignment obtained by the algorithm.

Similarly, for *intra-migrative* scenario, the scheduling problem can be solved in two phases: first, *assigning tasks to processor types* (rather than to individual processors) before run-time and then *scheduling* the tasks on each processor type using a multiprocessor scheduling algorithm at run-time. Scheduling tasks that are assigned to a processor type is also a well-researched topic as this problem is equivalent to identical multiprocessor scheduling problem and *optimal schedul-*

ing algorithms exist for this problem. Hence, the focus of this work is to design algorithms for assigning tasks to processor types on heterogeneous multiprocessors with a constant number of distinct processor types assuming that the tasks are later scheduled using the optimal scheduling algorithms. Therefore, assuming that the tasks are scheduled during run-time on each processor type using such optimal scheduling algorithms, this work focuses on designing algorithms for assigning tasks to processor types such that there exists a schedule that meets all deadlines for the assignment output by the algorithm.

For the non-migrative scenario as well as the intra-migrative scenario, this work considers both the *two-type* heterogeneous multiprocessors in which the number of distinct processor types is *two* and the generic *t-type* heterogeneous multiprocessors in which the number of distinct processor types is $t \geq 2$.

For the task assignment problems under consideration, it is not possible to design an optimal algorithm with polynomial time-complexity unless P = NP. Hence, non-optimal algorithms with polynomial time-complexity are designed and a metric referred to as the *speed competitive ratio* is used to quantify the the performance of these algorithms. The speed competitive ratio of an algorithm A is the smallest number S, such that any task set that can be assigned by an optimal algorithm upon a particular platform before run-time to meet all deadlines when scheduled at run-time, can also be assigned by A to meet all deadlines if given a platform in which each processor is S times as fast.

For the problem of assigning *tasks to processor types*, polynomial time-complexity algorithms with finite speed competitive ratios are proposed for two-type and t-type heterogeneous multiprocessors. Similarly, several polynomial time-complexity algorithms with different speed competitive ratios are also proposed for the problem of assigning *tasks to individual processors* on two-type and t-type heterogeneous multiprocessors.

This work also studies the problem of scheduling tasks that share resources such as data structures and sensors on heterogeneous multiprocessors — referred to as the *shared resource scheduling* problem. Tasks must operate on such resources in a *mutually exclusive* manner while accessing the resource, that is, at all times, when a task holds a resource, no other task can hold that resource. For this problem, polynomial time-complexity algorithms with finite speed competitive ratios are proposed for two-type and t-type heterogeneous multiprocessors.

Overall, the proposed algorithms have the following advantages. The proposed algorithms for shared resource scheduling problem are first of their kind since no previous algorithm exists for this problem. The other algorithms proposed in this work either have a better speed competitive ratio and/or a better time-complexity and/or a better average-case performance compared to the existing algorithms in the literature.

Resumo

Hoje em dia, os sistemas embebidos são uma das áreas mais importantes no contexto das ciências da computação, visto que 98% dos computadores são desse tipo. A maior parte dos sistemas embebidos albergam aplicações de controlo que têm que interagir de uma forma quase "instantânea" (isto é, as tarefas que compõem este tipos de aplicações têm que disponibilizar os resultados da suas computações dentro de uma meta temporal) com o meio onde estão inseridos. Dada a importância das áreas onde estes sistemas são usados, eles têm sido objecto de um grande esforço por parte da comunidade científica na procura de algoritmos de escalonamento que permitam, por um lado, quando em execução, satisfazer os requisitos das aplicações e por outro, antes da execução, garantir que esses requisitos serão satisfeitos. Isto é designado de *teoria de escalonamento para sistemas de tempo real*.

A teoria de escalonamento para sistemas de tempo real baseados em sistemas uni-processador está bem desenvolvida; os resultados científicos dessa teoria são ensinados em universidades e são bastante usados na indústria. A teoria de escalonamento para sistemas de tempo real baseados em sistemas multi-processador é uma área emergente, no entanto, a maior parte dos trabalhos considera somente sistemas multi-processador compostos por processadores idênticos. Porém, a indústria de processadores está a dirigir a sua produção para sistemas com uma arquitectura heterogénea, isto é, sistemas multi-processador compostos por processadores heterogéneos. como são exemplo o AMD Fusion, o Intel Atom e o NVIDIA Tegra.

Este trabalho aborda o problema de escalonar conjuntos de tarefas por forma a cumprirem as metas temporais em sistemas com arquitecturas heterogéneas. Nestes sistemas a execução das tarefas depende to tipo de processador no qual está a ser executada. Para plataformas com este tipo de arquitecturas, criar algoritmos de escalonamento assumindo que as tarefas podem, durante a sua execução, migrar livremente entre processadores de diferentes tipos é bastante difícil de conseguir (senão impossível), porque diferentes tipos de processadores têm conjuntos de instruções diferentes assim como formatos de registos e etc. Neste contexto, torna-se imperativo do ponto de vista prático criar algoritmos de escalonamento nos quais as tarefas não possam migrar entre processadores (designadas de não-migratórias) ou que possam migrar somente entre processadores do mesmo tipo (designadas de intra-migratórias).

Assumindo uma aplicação em que todas as tarefas são não-migratórias, o problema pode ser resolvido em dois passos: (i) atribuir previamente as tarefas aos processadores e (ii) em execução escalonar as tarefas usando um algoritmo apropriado para sistemas uni-processador. Os algoritmos para sistemas uni-processador têm sido estudados durante as últimas décadas e são, hoje-em-dia, considerados bem desenvolvidos. Existem alguns algoritmos para sistemas uni-processador que são considerados óptimos (isto é, são capazes de escalonar qualquer conjunto de tarefas escalonável). Portanto, o foco deste trabalho é criar algoritmos para atribuição de tarefas aos processadores heterogéneos e depois estas tarefas são escalonadas de acordo com um qualquer algoritmo de escalonamento (preferencialmente óptimo) apropriado para sistemas uni-processador.

De uma forma semelhante, para aplicações compostas por tarefas intra-migratórias, o problema pode ser resolvido, também, em dois passos: (i) atribuir tarefas aos tipos de processadores (ao invés de atribuir aos processadores) e (ii) em tempo de execução escalonar essas tarefas de acordo com um algoritmo apropriado para sistemas multi-processador. Os algoritmos de escalonamento para sistemas multi-processador (assumindo sistemas com uma arquitectura idêntica, isto é, todos os processadores são iguais) têm sido estudados durante os últimos anos, e existem alguns que são considerados óptimos. Deste modo, este trabalho também endereça o problema de atribuição de tarefas a tipos de processadores heterogéneos. Em execução as tarefas são escalonadas usando algoritmos de escalonamento apropriados para sistemas com uma arquitectura idêntica.

Como estratégia de investigação foram assumidas as seguintes configurações: *2-tipos* e *t-tipos*. Na configuração *2-tipos* assume-se que o sistema multi-processador é composto por dois tipos de processadores heterogéneos enquanto que na configuração *t-tipos* assume-se que o sistema tem *t* (mais do que dois) tipos de processadores heterogéneos. Para o problema de atribuição de tarefas neste contexto, não é possível criar algoritmos de atribuição óptimos com uma complexidade temporal do tipo polinomial. Portanto, os algoritmos de atribuição de tarefas (a atribuição de tarefas é sempre feita antes da execução) desenvolvidos são considerados não-óptimos e é usado "speed competitive ratio" (SCR) como métrica para quantificar o seu desempenho. O SCR representa a relação da capacidade de processamento dos processadores. Assumindo que para um conjunto de tarefas existe um qualquer algoritmo de atribuição de tarefas para uma dada plataforma e que assegure que quando em execução, escalonadas de acordo com um algoritmo de escalonamento óptimo, todas as tarefas cumprem as metas temporais. O SCR de um algoritmo A é o menor número S (em que S representa a capacidade de processamento dos processadores) por forma a atribuir esse conjunto de tarefas e assegurar que esse conjunto de tarefas é escalonável.

Para o problema de atribuir tarefas aos tipos de processadores, neste trabalho são propostos dois algoritmos, um para os sistemas *2-tipos* e outro para os sistemas *t-tipos*. Ambos algoritmos apresentam um SCR finito. Para os sistemas *2-tipos*, o algoritmo apresenta baixa complexidade temporal do tipo polinomial enquanto que para os sistemas *t-tipos* apresenta complexidade temporal do tipo polinomial. Para o problema de atribuir tarefas aos processadores, neste trabalho também são propostos vários algoritmos para os sistemas *2-tipos* e um para os sistemas *t-tipos*. Todos algoritmos apresentam um SCR finito. Para os sistemas *2-tipos*, alguns algoritmos apresentam baixa complexidade temporal do tipo polinomial enquanto que para os sistemas *t-tipos* apresenta complexidade temporal do tipo polinomial.

Em muitos sistemas de computação, além da partilha do processador, as tarefas também partilham outros recursos como estruturas de dados, sensores e etc. Portanto, nestes casos as tarefas devem usar tais recursos de uma forma exclusiva. Isto é, um recurso só pode ser usada por uma tarefa de cada vez. Neste trabalho também foi estudado o problema associado a este tipo de tarefas (que partilham outros recursos além do processador) em sistemas multi-processador composto por processadores heterogéneos. Neste trabalho são propostos dois algoritmos para este tipos de tarefas, um para os sistemas *2-tipos* e outro para os sistemas *t-tipos*. Ambos algoritmos apresentam um SCR finito. Para os sistemas *2-tipos*, o algoritmo apresenta baixa complexidade temporal do tipo polinomial enquanto que para os sistemas *t-tipos* apresenta complexidade temporal do tipo polinomial.

Os algoritmos propostos apresentam as seguintes vantagens quando comparados com os algoritmos existentes: (i) alguns dos algoritmos propostos são pioneiros (nomeadamente os que se referem à atribuição de tarefas que partilham recursos); (ii) apresentam SCR melhor; (iii) apresentam uma complexidade temporal melhor e (iv) também apresentam um desempenho médio melhor.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Björn Andersson for his excellent guidance during the first two years of my PhD when he was at CISTER Research center. I thoroughly enjoyed the experience of working with him. Björn's commitment to the highest standards, his enthusiasm towards research inspired and motivated me. He always made me comfortable during the discussions which I might have abused a couple of times with my stupid remarks during some of our discussions in the middle of the nights.

I am deeply grateful to Vincent Nélis, my supervisor, for the guidance, motivation and invaluable support that he has provided over the last couple of years. Vincent is someone you will instantly like and never forget once you meet him. I think he is one of the friendliest supervisors that any student can wish for. It was really a pleasant experience to work with him as he gave me so much freedom and always encouraged me to make my own choices. Of course, our various coffee-table conversations on almost all the random topics in the world were always entertaining.

Thanks to all the people at CISTER for the great support at various levels. I would like to specially address a few of them. I am thankful to Eduardo Tovar for creating such a great environment at CISTER and for going out of his way sometimes to make my stay at CISTER and Porto a memorable one. I would like to thank some of my research collaborators, especially Konstantinos Bletsas and Geoffrey Nelissen, who have enhanced my enthusiasm and understanding of real-time systems. I would like to acknowledge Paulo Baltarejo Sousa for translating the abstract of the thesis in Portuguese. Thanks to all the students at CISTER, who have given me hours of helpful and enjoyable discussions; probably, the fact that most of us came from different parts of the world with different perspectives made it a memorable experience. My thanks also go to Sandra Almeida and Inês Almeida for the administrative support. Inês in particular has been extremely helpful with all the logistics during my initial days in Porto. Thanks to all the 'desi junta' for innumerable and unforgettable "Puerto Rico" moments; I will cherish those days/nights that we spent playing the game, those post-game discussions, arguments, wins and losses, and of course, the great food will stay with me for a long time.

Most importantly, none of this would have been possible without the love and patience of my family. My parents have been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to them. Finally, I would like to thank my best friend and wife, Dakshina for her understanding and love during the past few years. Her support, encouragement and care have helped me overcome setbacks and stay focused on my studies and her sense of humor and perspective about life has helped me stay sane through these years.

This work was partially supported by FCT (Fundação para a Ciência e Tecnologia) under the individual doctoral grant SFRH/BD/66771/2009.

List of Author's Publications

This is a list of papers and publications that reflects the results achieved during the development of the research work presented in this dissertation.

Journals

- Gurulingesh Raravi, Björn Andersson, Vincent Nélis and Konstantinos Bletsas, "Task Assignment Algorithms for Two-Type Heterogeneous Multiprocessors", *Real-Time Systems* (Accepted for publication).

- Gurulingesh Raravi, Björn Andersson and Konstantinos Bletsas, "Assigning Real-time Tasks on Heterogeneous Multiprocessors with Two Unrelated Types of Processors", *Real-Time Systems*, Volume 49, Number 1, pages 29–72, January, 2013.

Conferences

- Gurulingesh Raravi and Vincent Nélis, "A PTAS for Assigning Sporadic Tasks on Two-type Heterogeneous Multiprocessors", *In Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 117–126, San Juan, Puerto Rico, December 4–7, 2012.

- Gurulingesh Raravi, Björn Andersson, Konstantinos Bletsas and Vincent Nélis, "Task Assignment Algorithms for Two-Type Heterogeneous Multiprocessors", *In Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 34–43, Pisa, Italy, July 11–13, 2012. **(Outstanding Paper Award)**

- Gurulingesh Raravi, Björn Andersson and Konstantinos Bletsas, "Provably Good Scheduling of Sporadic Tasks with Resource Sharing on a Two-Type Heterogeneous Multiprocessor Platform", *In Proceedings of the 16th International Conference On Principles Of Distributed Systems*, pages 528–543, Toulouse, France, December 12–16, 2011.

- Björn Andersson, Gurulingesh Raravi and Konstantinos Bletsas, "Assigning Real-Time Tasks on Heterogeneous Multiprocessors with Two Unrelated Types of Processors", *In Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 239–248, Washington, DC, USA, November 30 – December 3, 2010.

Technical Reports

- Gurulingesh Raravi and Björn Andersson, "Real-Time Scheduling with Resource Sharing on Heterogeneous Multiprocessors", 2013.

- Gurulingesh Raravi and Björn Andersson, “Task Assignment Algorithm for Two-type Heterogeneous Multiprocessors using Cutting Planes”, 2013.
- Gurulingesh Raravi and Vincent Nélis, “Task assignment algorithms for heterogeneous multiprocessors”, 2013.

Contents

I Introduction **1**

1 Background on Real-Time Systems **3**
1.1 Introduction to real-time systems 5
1.2 Modeling real-time systems 6
1.3 Categorization of real-time systems 14
1.4 Real-time scheduling paradigms 16
1.5 Background on real-time scheduling theory 17

2 Overview of This Research **23**
2.1 Problem definition 23
2.2 Hardness of the problem 25
2.3 Why study heterogeneous multiprocessors? 26
2.4 Common assumptions 27
2.5 Performance metrics 28
2.6 Contributions and significance of this work 34
2.7 Organization of the report 36

II Two-type Heterogeneous Multiprocessors **39**

3 Intra-migrative Scheduling on Two-type Heterogeneous Multiprocessors **41**
3.1 Introduction 41
3.2 System model 44
3.3 The hardness of the intra-migrative task assignment problem 45
3.4 MILP-Algo: An optimal intra-migrative task assignment algorithm 49
3.5 LP-Algo: An intra-migrative task assignment algorithm 52
3.6 SA: An intra-migrative task assignment algorithm 57
3.7 Speed competitive ratio of SA algorithm 58
3.8 Average-case performance evaluations 68
3.9 Conclusions 71

4 Non-migrative Scheduling on Two-type Heterogeneous Multiprocessors **73**
4.1 Introduction 73
4.2 The hardness of the non-migrative task assignment problem 76
4.3 FF-3C algorithm and its variants 80
4.4 SA-P algorithm 118
4.5 Cutting plane algorithm 129
4.6 A polynomial time approximation scheme 154

4.7 Conclusions and Discussions 182

5 Shared Resource Scheduling on Two-type Heterogeneous Multiprocessors 187
5.1 Introduction 187
5.2 System model and assumptions 189
5.3 The hardness of the shared resource scheduling problem 190
5.4 Overview of our approach 192
5.5 Few notations and useful results 193
5.6 FF-3C-vpr algorithm and its speed competitive ratio 201
5.7 Conclusions 206

III T-type Heterogeneous Multiprocessors 209

6 Intra-migrative Scheduling on T-type Heterogeneous Multiprocessors 211
6.1 Introduction 211
6.2 System model 214
6.3 MILP-Algo: An optimal intra-migrative algorithm 215
6.4 An overview of our intra-migrative task assignment algorithm, LPG_{IM} 216
6.5 Step 1 of LPG_{IM}: Solving the LP formulation 217
6.6 Step 2 of LPG_{IM}: Forming the bi-partite graph 219
6.7 Step 3 of LPG_{IM}: Detecting and removing the circuits in the graph 221
6.8 Step 4 of LPG_{IM}: Integrally assigning the fractional tasks 226
6.9 Conclusions 235

7 Non-migrative Scheduling on T-type Heterogeneous Multiprocessors 237
7.1 Introduction 237
7.2 System model 240
7.3 LPG_{NM}: The non-migrative task assignment algorithm 241
7.4 Conclusions 244

8 Shared Resource Scheduling on T-type Heterogeneous Multiprocessors 245
8.1 Introduction 245
8.2 System model 247
8.3 Overview of our algorithm 251
8.4 The new algorithm, LP-EE-vpr 253
8.5 Speed competitive ratio of LP-EE-vpr algorithm 260
8.6 Discussion 280
8.7 Conclusions 282

IV Conclusions 285

9 Conclusions, Discussions and Future Directions 287
9.1 Summary of results 288
9.2 Implication of the results 289
9.3 Future directions 290
9.4 Concluding remarks 292

References 295

Part I

Introduction

Chapter 1

Background on Real-Time Systems

For the past 40 years, the transistor/semi-conductor density has roughly doubled every 18 months as observed by Gordon Moore [BC03]. Also, since 1980s, the processor clock speed has increased about 30% every year. The microprocessor vendors improved their silicon technology to go faster and faster until early 2000s. However, during this time, they realized that the clock speed has hit a wall since it started violating the principles of fundamental physics. Although computing power increases linearly with the clock speed, the power density increases with the square or cube, depending on the electrical model. It was seen that clock frequencies beyond about 5GHz could result in melting the chip unless cooled using exotic cooling technologies [LM08].

In order to overcome the limitations imposed by this clock speed wall and to continue leveraging Moore's Law to increase performance and reduce power, microprocessor vendors decided to go the *multiprocessor/multicore chip* way. Instead of increasing the clock rate of power-hungry monolithic cores, chip vendors adapted a design in which multiple slower processors are integrated on a single chip which collectively increase the computational capability while consuming lesser power. With this design choice, the number of processing cores per chip started doubling with each new generation of microprocessor chips, a trend which will continue for the foreseeable future [LM08], which in turn is leading to a significant increase in the number of applications being deployed on such chips.

Subsequently, computer controlled systems have percolated in all aspects of our daily lives. From mobile phones to nuclear reactor controllers, computer controlled systems have reached all aspects of human life in merely a few decades. In near future, it is expected that day-to-day activities such as cleaning the house, driving the car, etc. will be performed by computer controlled systems without human intervention. Actually, research prototypes that can perform such activities already exist and a few of these are even commercially available. The complexity of such computer-controlled systems is continuously increasing due to the increase in the functionalities, the increase in the interactivity between different functionalities and the increase in the responsiveness requirement. Hence, many of these computer-controlled systems demand more and more performance from the processors that implement these functionalities and multicores have emerged as an inevitable choice for such systems due to their high performance, low cost and

low power consumption characteristics and also due to the aggressive marketing of multicores by all the chip vendors.

Most of the computer controlled systems these days are *embedded systems*. An embedded system is defined as a computer-controlled system (sometimes, it can even be a simple monitoring system) in which the hardware and software is specifically designed for a particular functionality. For example, mobile phones, set-top boxes, cruise control systems in cars and autopilot system in an airplane. It is estimated that over 98% of all computing devices are "embedded" in nature and hence they do not even look like computers [RM09, Tan07, Zha03]. In other words, computers are moving away from the well-known desktop and can be found in everyday devices like credit cards, microwaves, mobile phones and cars. Let us consider an example of such an embedded system listed in [And03].

Example 1. *Consider a hypothetical car where a computer in the car is given a street address and the computer automatically drives to that address with no human intervention (research prototypes that can do things similar to this exist [JPKA95, Spe] but are not commercially available). Think about yourself as being the computer in the car.*

You are driving your car and approach a crossing. You see that there is no pedestrian there (a sensor reading) so you close your eyes for a few seconds and listen to the radio while your car approaches the intersection, and after those seconds you conclude that you can drive straight ahead without any need to slow down (an action). If, during those seconds, a pedestrian starts to walk at the crossing, an accident may occur, neither because your sensor reading was incorrect nor because you inferred an incorrect action based on your sensor reading, but because your action was based on a sensor reading that was too old. If you had monitored your environment with periodic sampling of a high enough rate, an accident would not have occurred. Let us assume that you woke up in time to see the pedestrian (a sensor reading) so you conclude that you should break or steer away (an action). Although you computed the right answer (that is, made the right decision) this is not sufficient for successful driving; you need to apply a break force quickly enough, that is, before a deadline. This deadline depends on states in the environment, for example, the speed of your car and the distance from your car to the pedestrian.

Observe that in the embedded system described in Example 1, it is not only essential to perform the computations or actions in a logically correct manner but it is also important to perform them at the *right time*. For example, failing to detect the pedestrian at the right time or failing to apply the brakes with the right force at the right time may lead to an undesired output. Such embedded systems in which the correctness of the output not only depends on the value but also depends on the time of delivery, are referred to as real-time systems which is the topic of next section.

Organization of the chapter. The rest of the chapter is organized as follows. Section 1.1 introduces the concept of real-time systems. Section 1.2 discusses how such systems are modeled. Section 1.3 discusses the classification of real-time systems. Section 1.4 introduces the design space of real-time scheduling and finally a brief background on real-time scheduling theory is provided in Section 1.5.

1.1 Introduction to real-time systems

A real-time system is a system in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. As mentioned earlier, in Example 1, if the pedestrian is not detected at the *right time* and/or if the brakes are not applied at the right time with the right force to bring the car to a halt then it may lead to an accident. Similarly, all real-time systems have *temporal requirements* which need to be guaranteed in order to avoid an undesirable behavior from the system.

The temporal requirements in a real-time system generally have their origin in the design process [And03]. Specifically, the designer specifies how the computer-controlled system in general should behave. For example, consider Adaptive Cruise Control (ACC) in a car. The objective of an ACC system is to ensure that the car approximately maintains a *safe distance* (typically set by the driver) from any *leading vehicle* (i.e., a vehicle in front of the car in the same lane). Once the behavior is defined, the designer derives the timing constraints such that, if these constraints are satisfied, the behavior is said to be as desired. These derived timing requirements depend on the state of the environment. For example, in an ACC system, the temporal delay that can be tolerated between detecting the leading vehicle (sensing) and taking actions in order to maintain the car at a pre-set safe separation distance from the leading vehicle depends on (i) the tolerable error margin, (ii) the dynamics of the car, (iii) the current distance of separation between the car and the leading vehicle, the speed of the leading vehicle, etc.

The consequence of violating the temporal requirements of a real-time system can depend on the environment. In some cases, not satisfying the temporal requirements may lead to a catastrophe such as severe damage to the equipment or even loss of human lives — such systems are referred to as *hard real-time systems*. For example, failing to maintain the safe distance between the car and the leading vehicle may lead to an accident which in turn may lead to loss of human lives. In other cases, the consequence may not be that harmful and instead the user experiences a degradation in the quality-of-service provided by the system which does not endanger the integrity of the user or the equipment or the environment — such systems are referred to as *soft real-time systems*. For example, in a multimedia application, failing to decode a video frame on time once in a while is acceptable for the user as long as certain quality of service is ensured. (More discussion about hard and soft real-time systems is provided later in the chapter.) Therefore, it is very essential to ensure that all the timing requirements are met for hard real time systems and to ensure that an acceptable quality-of-service is provided for soft real-time systems. This can be achieved using schedulability analysis.

Scheduling and Schedulability Analysis. An important aspect in the process of designing real-time systems is to ensure before run-time that the timing requirements are met at run-time. In order to do this, the *entities* which perform computations (such as reading the sensor data, computing the current distance of separation between the car and the leading vehicle based on the sensor readings, computing the speed of the car) need to be *scheduled* using an algorithm so as to meet all the timing requirements. In other words, these entities need to be *allocated* sufficient *resources*

such that they finish their execution before certain time thereby meeting all the timing requirements of the system. The process of verifying whether the timing requirements of the real-time system will be met or not when entities are scheduled using an algorithm on the computing resources, is referred to as the *schedulability analysis* of the algorithm. For hard real-time systems, the analysis needs to be rigorous and performed before run-time to provide a guarantee that all the timing requirements will be met during the run time. Whereas for soft real-time systems, some kind of stochastic analysis is sufficient as such systems need not meet *all* the timing requirements and may afford to miss *some* of its timing requirements as long as the system continues to provide an acceptable quality-of-service.
We now describe how the entities that perform computations (also referred to as the *workload*) are modeled and the resources that need to be allocated to these entities (also referred to as the *computing platform*) are modeled. Overall, modeling of real-time systems is discussed next.

1.2 Modeling real-time systems

First, the description of modeling the real-time workload is provided and then the modeling of the computing platform on which the workload is executed is discussed.

1.2.1 Modeling real-time workload

This section describes how the real-time workload is modeled using the notion of *job* and *task*.

1.2.1.1 Job and its characterization in real-time systems

Definition 1 (Job). *A unit of work (say, a set of instructions) that performs some computations and which needs to be scheduled and executed is referred to as a job.*

Informally, a job is a set of instructions in the context of the application that executes sequentially and provides a given application-relevant logical result. In real-time systems, each job is characterized by the following parameters:

- **Release time of a job:** The time instant at which a job becomes available for execution is referred to as the release time of the job. So, a job can be scheduled and executed any time instant at or after its release time[1].
- **Deadline of a job:** The time instant by which the execution of a job needs to be completed is referred to as the deadline of the job. The deadline can be expressed in two ways:
  - **Relative deadline of a job:** It is the maximum allowable *response time* of the job, where the response time of the job is the duration of time interval spanning from the release time of the job to its completion time.

[1] Note that there may be other constraints as well such as data dependency and control dependency that need to be considered before scheduling and executing the job.

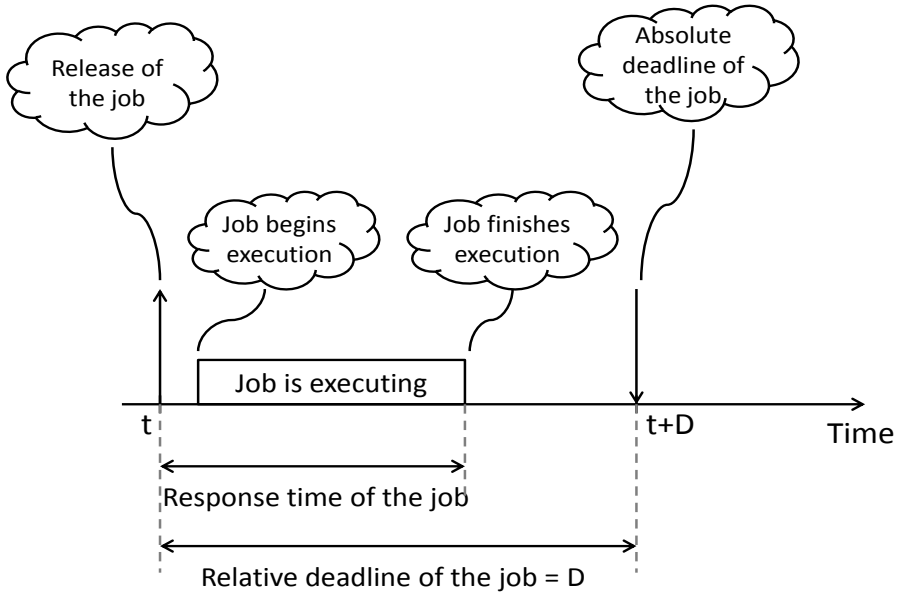


Figure 1.1: The parameters used to characterize a job in real-time systems. In this example, the release time of the job is t and its relative deadline is D which implies that the absolute deadline of the job is $t + D$.

– **Absolute deadline:** It is the absolute time instant by which the job is required to finish its execution. In other words, it is given by the release time of the job plus the relative deadline of the job.

Figure 1.1 illustrates the above discussed terms. As shown in the figure, the job has a release time of t and a relative deadline of D.

1.2.1.2 Task and its characterization in real-time systems

This section defines a task, lists different kinds of tasks and parameters used to characterize the tasks in the context of real-time systems.

Definition 2 (Task). *A collection of related jobs which jointly provide some system function is referred to as a task. Stated another way, a task releases a (potentially infinite) sequence of jobs. A task is generally denoted by τ_i where i is the task index.*

Remark about notation. From the definition of the job and the task, it can be seen that, a task is an abstract entity which releases many jobs over a period of time and every job of a task executes on some processor. However, to avoid tedium, in the rest of the thesis, instead of saying "a job of a task executes", we say that, "a task executes".

A task is characterized by the following parameters.

- **Worst-case execution time of a task:** The worst-case execution time (WCET) of a task is defined as the maximum duration of time that the task (i.e., any job of this task) could

take to execute on a given processor. All the jobs of a task will have the same worst-case execution time which is that of the task. In other words, the WCET of a task is the upper bound on the execution duration of any job of this task and hence the actual execution times of some of the jobs of this task may be less than the WCET of the task. The WCET of a task τ_i is commonly denoted by C_i.

- **Relative Deadline of a task:** The relative deadline of a task is defined as the maximum allowable response time of any job of this task. Hence, all the jobs of a task have the same relative deadline which is that of the task. The deadline of a task τ_i is generally denoted by D_i.

Remark about notation. To avoid tedium, in the rest of the thesis, we refer to "Relative Deadline" as "Deadline".

Definition 3 (**Task set**). *A collection of tasks is referred to as a task set and is denoted by* τ.

In real-time systems, tasks can be classified into three categories depending on the job release pattern:

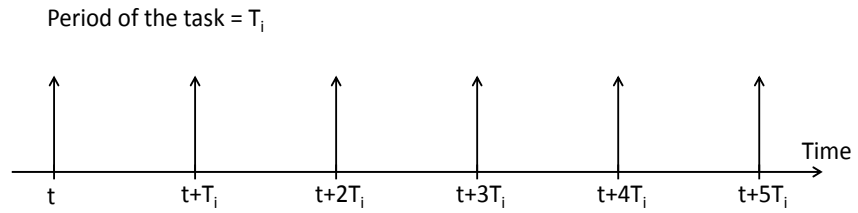
- **Periodic:** These tasks generate jobs *periodically*, separated by a fixed time interval, in the sense that, after the arrival of the first job at any time instant, the arrival of subsequent jobs are separated by a fixed time interval which is given by the *period* of the task. The period of such a task τ_i is generally denoted by T_i.

- **Sporadic:** In this task model, jobs arrive *sporadically*, i.e, after the arrival of the first job at any time instant, the subsequent jobs of this task may arrive at any time once a *minimum inter-arrival time* has elapsed since the arrival of the previous job of the same task. The minimum inter-arrival time of such a task τ_i is generally denoted by T_i.

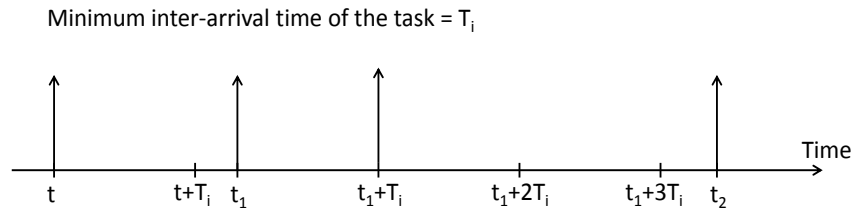
- **Aperiodic:** The jobs of these tasks may arrive at any time instant, in the sense that, no information about their arrival pattern is given.

Figure 1.2 illustrates the above discussed three categories of tasks. Figure 1.2a shows a periodic task with a period of T_i. As shown in this figure, the first job of the task is released at time t and then subsequent jobs of this task are released exactly T_i time units apart, i.e., the second job is released at $t + T_i$, the third job is released at $t + 2T_i$ and so on. Figure 1.2b shows a sporadic task with a minimum inter-arrival time of T_i. As shown in this figure, release of two consecutive jobs is always separated by a time duration of at least T_i units; for example, the first job is released at t, the second job is released at $t_1 \geq t + T_i$, the third job is released at $t_1 + T_i$, the fourth job is released at time $t_2 \geq t_1 + 2T_i$ and so on. Figure 1.2c shows an aperiodic task for which nothing can be said about the job release pattern.

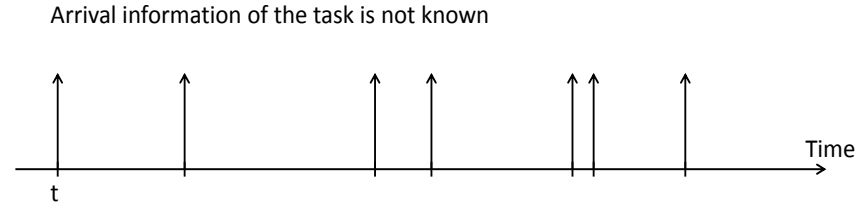
A periodic task is characterized by its worst-case execution time, period and deadline. A sporadic task is characterized by its worst-case execution time, minimum inter-arrival time and deadline. An aperiodic task is characterized by its worst-case execution time and deadline.



(a) Periodic task with a period of T_i — the duration between the arrival of any two consecutive jobs is exactly equal to T_i units.



(b) Sporadic task with a minimum inter-arrival time of T_i — the duration between the arrival of any two consecutive jobs is either equal to or greater than T_i units.



(c) Aperiodic task with no information on release pattern — the duration between the arrival of any two consecutive jobs is unknown.

Figure 1.2: A visualization of different categories of tasks — an upward arrow indicates the arrival of a job of a task.

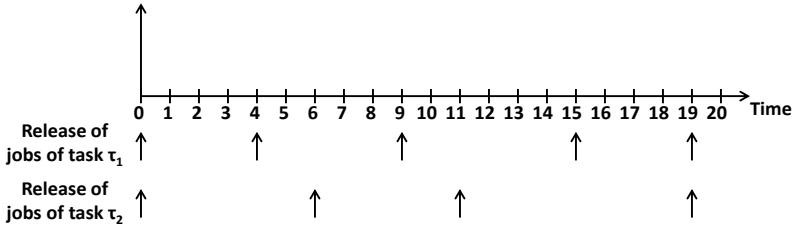
This research considers sporadic tasks.

We now define valid job arrival pattern for a sporadic task set.

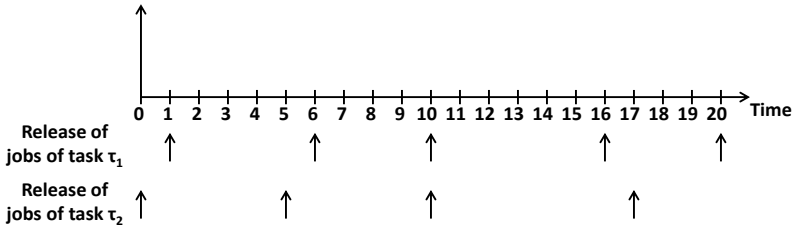
Definition 4 (Valid job arrival pattern of a sporadic task set.). *A job arrival pattern of a sporadic task set is said to be valid if every task in the task set respects its minimum inter-arrival time while releasing the jobs. For a given sporadic task set, there can be multiple valid job arrival patterns.*

The following example illustrates the concept of valid job arrival pattern for a given sporadic task set.

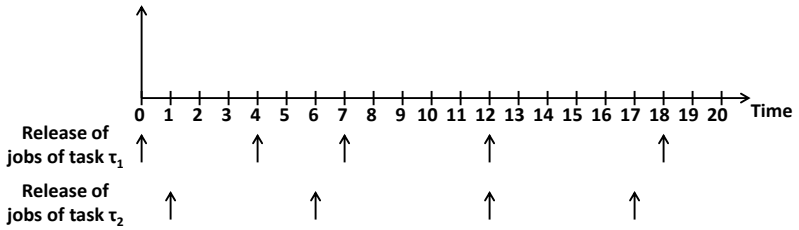
Example 2. *Consider a sporadic task set $\tau = \{\tau_1, \tau_2\}$. Let the minimum inter-arrival time of task τ_1 be given by $T_1 = 4$ and let the minimum inter-arrival time of task τ_2 be given by $T_2 = 5$. Figure 1.3a and Figure 1.3b show a valid job arrival pattern each for this task set. Note that, it is*



(a) A valid job arrival pattern.



(b) Another valid job arrival pattern.



(c) An invalid job arrival pattern.

Figure 1.3: Examples of valid and invalid job arrival patterns for the task set of Example 2.

easy to construct many more valid job arrival patterns for this task set in a similar manner. For the sake of completeness, Figure 1.3c shows an invalid job arrival pattern for this task set (since the release time of second and third job is not separated by at least 4 *units).*

The sporadic task sets can be classified into three categories depending on the relation between the deadlines and the minimum inter-arrival times of every task in the task set.

- **Implicit-deadline:** In implicit-deadline sporadic task set, every task has its deadline *equal to* its minimum inter-arrival time, i.e., $\forall \tau_i \in \tau : D_i = T_i$.

- **Constrained-deadline:** In such a task set, every task has its deadline *no greater than* its minimum inter-arrival time, i.e., $\forall \tau_i \in \tau : D_i \leq T_i$.

- **Arbitrary-deadline:** In arbitrary-deadline sporadic task set, there is no relation between deadlines and minimum inter-arrival times of tasks. In other words, in such task sets, the deadline of every task may be less than or equal to or greater than its minimum inter-arrival time.

This work considers implicit-deadline sporadic task sets.

For the sake of completeness and future references, we now formally define an implicit-deadline sporadic task set.

Definition 5 (**Implicit-deadline sporadic task set**). *In an implicit-deadline sporadic task set τ, each task $\tau_i \in \tau$ is characterized by a worst-case execution time (WCET) and a minimum inter-arrival time T_i (which is equal to its deadline). Each task τ_i releases a (potentially infinite) sequence of jobs, with the first job released at any time and subsequent jobs released at least T_i time units apart. Each job released by task τ_i has to complete its execution within T_i time units from its release.*

We now define a parameter that is used to characterize an implicit-deadline sporadic task and a parameter that is used to characterize a set of implicit-deadline sporadic tasks. These parameters are extensively used while performing the schedulability analysis of algorithms in real-time literature and also in subsequent chapters of this document.

Definition 6 (**Utilization of an implicit-deadline sporadic task**). *For an implicit-deadline sporadic task, the ratio of its worst-case execution time and its minimum inter-arrival time is referred to as the utilization of the task. The utilization of a task τ_i is denoted by u_i and is formally defined as: $u_i \stackrel{\text{def}}{=} \frac{C_i}{T_i}$.*

Definition 7 (**Utilization of an implicit-deadline sporadic task set**). *It is the sum of utilizations of all the tasks in the task set. The utilization of a task τ is denoted by U_τ and is formally defined as: $U_\tau \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i$.*

1.2.2 Modeling the computing platform

This section describes how the computing resources, also referred to as computing platform, on which the workload needs to be allocated and executed, is modeled. In the context of the work, we are interested in modeling the computing platform as a set of *processing elements* which has a limited computational capacity. This is sufficient for the purpose of this work since the end objective is to assign and execute the workload on these processing elements such that on each processing element (a group of processing elements, respectively), the total computational demand of the workload assigned on the processing element (the group of processing elements, respectively) should not exceed the capacity of that processing element (that group of processing elements, respectively) which in turn guarantees that all the timing requirements of the workload will be met — more details are provided in the next chapter. For this reason, we ignore modeling the other architectural features of the computing platform such as caches, memory, system bus, etc. and hence do not consider the impact of sharing such hardware resources on the execution behavior of the tasks.

If the computing platform on which the real-time workload needs to be executed consists of a single processor then it is referred to as a *uniprocessor* platform. If the computing platform consists of multiple processors then it is referred to as a *multiprocessor* platform or a *multicore* platform. The real-time scheduling theory for uniprocessor system is well-understood. However,

the same cannot be said about the multiprocessor real-time scheduling theory (more details are given in the subsequent parts of this chapter). Hence, in this work, we focus on multiprocessor systems.

Remark about notation. In this dissertation, very often, the terms "processor" and "core" are interchangeably used and they both refer to the processing element on which the tasks are assigned and scheduled.

This work considers multiprocessor computing platforms.

Multiprocessor systems can be categorized into three groups as follows:

- **Identical:** In identical multiprocessors, all the processors are identical, in the sense that, they all have exactly the same computing capacity. Hence, the rate of execution of all tasks is the same on all processors. These multiprocessors are sometimes referred to as *homogeneous* multiprocessors. Some example of such multiprocessors are: Intel Core 2 Quad Processor [Int13e], Intel Core 2 Extreme Processor [Int13d], AMD Dual Core Processors [AMD13b] and ARM Cortex-A9 MPCore [ARM13a].

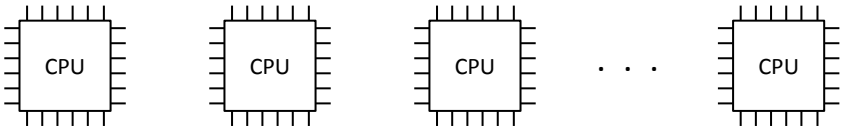
- **Uniform:** By contrast, in uniform multiprocessors, each processor is characterized by its own speed or computing capacity. Hence, the rate of execution of a task depends on the speed of the processor. Thus, a processor of speed $s > 1$ will execute all tasks s times faster than a processor of speed 1. Some examples of uniform multiprocessors are: ARM big.LITTLE Processing [ARM13b] and Samsung Exynos 5 Octa [Sam13b].

- **Heterogeneous:** These are multiprocessors in which the processors are of different *types*. For example, in a heterogeneous multiprocessor, some processors can be of type Central Processing Units (CPUs), some processors can be of type Graphics Processing Units (GPUs), some other processors can be of type network processors and so on. The processors of different types generally differ in their instruction sets, register formats, etc. Hence, the rate of execution of a task depends on both the processor type and the task. Indeed, not all tasks may be able to execute on all processors. These multiprocessors are sometimes referred to as *unrelated* multiprocessors. Some examples of heterogeneous multiprocessors are: AMD Fusion [AMD13a], Intel Atom [Int13c], Nvidia Tegra 3 [Nvi12].

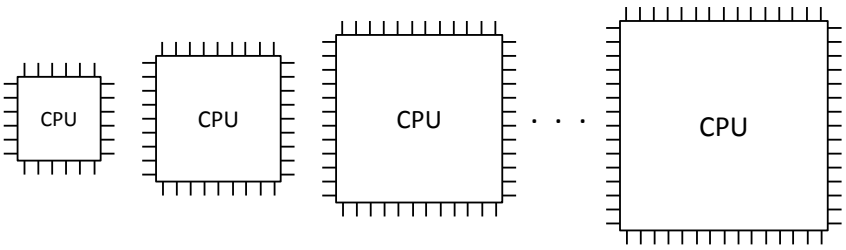
Figure 1.4 illustrates the above discussed three categories of multiprocessors.
A special case of heterogeneous multiprocessor is a t-type heterogeneous multiprocessor which is defined as follows.

Definition 8 (**t-type heterogeneous multiprocessor**). *A heterogeneous multiprocessor in which the number of distinct types of processors is a constant, $t \geq 2$, is referred to as a t-type heterogeneous multiprocessor. It is also referred to as a* ***t-type platform****.*

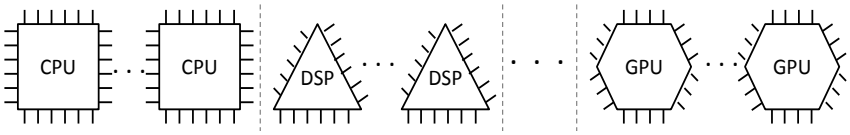
A special case of t-type heterogeneous multiprocessor is a two-type heterogeneous multiprocessor which is defined as follows.



(a) Identical multiprocessors: all the processors have the same speed.



(b) Uniform multiprocessors: processors have different speeds — size is proportional to the speed.



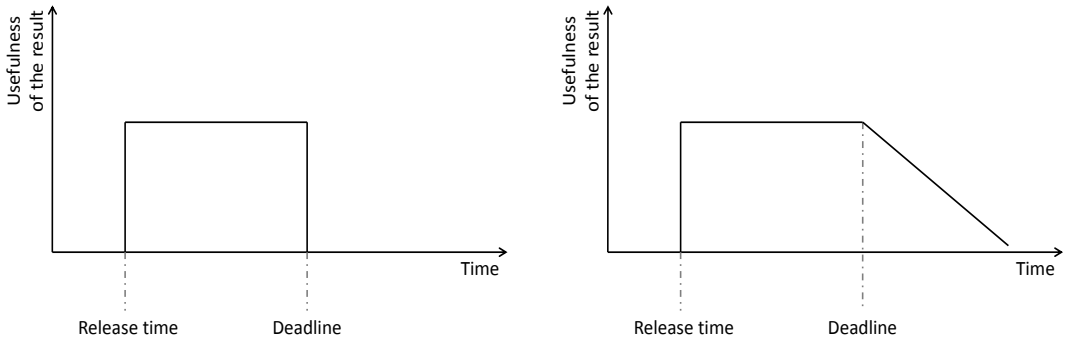
(c) Heterogeneous multiprocessors: processors are of different types; processors of different types can differ in their instruction sets, register formats, etc. — different shapes correspond to different processor types.

Figure 1.4: A visualization of different categories of multiprocessor systems.

Definition 9 (Two-type heterogeneous multiprocessor). *A two-type heterogeneous multiprocessor is a special case of t-type heterogeneous multiprocessor in which there are only two distinct types of processors, i.e., $t = 2$, and thus each processor in the system belongs to one of these types. It is also referred to as a* ***two-type platform****.*

This research considers both two-type and t-type heterogeneous multiprocessors.

The reason for studying heterogeneous multiprocessor systems is that the heterogeneous multiprocessor model is more generic than identical or uniform multiprocessor model, in terms of the systems that it can accommodate. Hence, the results obtained for this model are also applicable to identical or uniform multiprocessor models. The reason for studying heterogeneous multiprocessors with a constant number of distinct types of processors (i.e., both two-type and t-type) is that, (i) in practice, most of the heterogeneous multiprocessors are of this nature since many chip manufacturers offer chips having a constant number of distinct types of processors, especially two types of processors, for example, see [AMD13a, Int13b, Nvi12, App13, Qua13, Sam13a, ST 12, Tex13] and (ii) studying the generic t-type heterogeneous multiprocessors (in which $t \geq 2$) helps in understanding the problem better and solutions to such generic models may cater to complex systems in near future.



(a) Hard real-time job — usefulness of the result drops to zero (or even becomes negative) when tardiness exceeds zero.

(b) Soft real-time job — usefulness of the result degrades gracefully as the tardiness increases.

Figure 1.5: A visualization of hard and soft jobs in real-time systems.

1.3 Categorization of real-time systems

With the understanding of a job and a task, we now discuss the earlier mentioned two categories of real-time systems in detail. In literature, the real-time systems are generally categorized as either hard or soft real-time systems [Kop11, Liu00, Bur91]. One way the distinction is done is based on the usefulness of the result measured using the *tardiness* of jobs [Liu00].

Definition 10 (**Tardiness of a job**). *The tardiness of a job indicates how late the job execution is compared to its deadline. The tardiness of a job is zero if it completes execution on or before its deadline; otherwise, the tardiness of a job is given by the difference between its completion time and its deadline.*

Based on the usefulness of the result measured using tardiness, the jobs in real-time systems are categorized as follows:

- **hard real-time job:** the usefulness of the result (in the scope of the application) of a hard real-time job falls abruptly and even may become negative (i.e., may be harmful or catastrophic) when the tardiness of such a job exceeds zero. The implication of this is that missing the deadline of a hard real-time job can lead to a catastrophe and hence the user requires a rigorous validation by provably correct and efficient procedures to ensure that such jobs always meet their deadlines.

- **soft real-time job:** the usefulness of the result of a soft real-time job decreases gradually as the tardiness of such a job increases. The implication of this is that missing some deadlines of a soft real-time job is acceptable. The user in this case only requires a demonstration that the job meets some statistical constraint, for example, average number of deadlines that are met using techniques such as performance profiling.

The above defined hard and soft real-time jobs are illustrated in Figure 1.5.

Note that the above definitions can be extended for a task to obtain the terms *hard real-time task* and *soft real-time task*.

Definition 11 (**Hard real-time system.**). *A system that consists of only hard real-time tasks is referred to as hard real-time system.*

For hard real-time systems, the designer of the system has to prove rigorously before run-time that all the deadlines will be met and as a result system will not exhibit any undesired behavior at run-time. We now give an example of a hard real-time system [Liu00].

Example 3. *we consider an automatically controlled train. It cannot stop instantaneously. When the signal is red (stop), its braking action must be activated a certain distance away from the signal post at which the train must stop. This braking distance depends on the speed of the train and the safe value of deceleration. From the speed and safe deceleration of the train, the controller can compute the time for the train to travel the braking distance. This time in turn imposes a constraint on the response time of the jobs which sense and process the stop signal and activate the brake. This system is a hard real-time system as failing to meet the timing constraints here may lead to a catastrophe as it can cause loss of human lives and and/or significant damage to the equipment (i.e., train infrastructure) and hence this system needs to be formally validated at design time to guarantee that all timing requirements will be met.*

Definition 12 (**Soft real-time system.**). *A system with all soft real-time tasks is referred to as soft real-time system.*

For soft real-time systems, the designer of the system is rarely required to prove rigorously that the system meets its real-time performance objectives. However, in many systems, a statistical based guarantee (for example, the average number of deadlines that are met) needs to be provided. The following example of a soft real-time system is listed in [Liu00].

Example 4. *Let us consider multimedia systems that provide the user with services of "guaranteed" quality. For example, a frame of a movie must be delivered every thirtieth of a second, and the difference in the times when each video frame is displayed and when the accompanied speech is presented should be no more than* 80 *msec. In fact, it is common to subject each new video stream to be transmitted by a network to an acceptance test. If the network cannot guarantee the satisfaction of timing constraints of the stream without violating the constraints of existing streams, the new stream is rejected, and its admission is requested again at some later time. However, the users are often willing to tolerate a few glitches, as long as the glitches occur rarely and for short lengths of time. At the same time, they are not willing to pay the cost of eliminating the glitches completely. For this reason, we often see timing constraints of multimedia systems guaranteed on a statistical basis, (e.g., the average number of late/lost frames per minute is less than* 2*). Moreover, users of such systems rarely demand any proof that the system indeed honor its guarantees. The quality-of-service guarantee is soft, the validation requirement is soft, and the timing constraints defining the quality are soft.*

This work focuses on hard real-time systems.

1.4 Real-time scheduling paradigms

In real-time systems, and especially in hard real-time systems, an important aspect while building such systems is to schedule the tasks on the computing platform so as to meet all the deadlines. Scheduling is an act of allocating resources (especially, processors) between various tasks with the objective of meeting the deadlines of all the tasks. In real-time systems, there are at least two different ways to schedule the tasks [RS94].

- **Table-driven scheduling:** In table-driven scheduling, a table is generated before run-time with the time slots in which each task must be executed. During this process, it is ensured that, on a processor, at any time, at most one task is executing. Later, at run time, this table is used to execute the tasks in their respective time slots.
- **Priority-driven scheduling:** In priority-driven scheduling, a number referred to as *priority* is assigned to each task often based on the task parameters. For example, the priority assignment can be based on periods, say a task with smaller period has a higher priority compared to the priority of a task with a higher period. Then, among the tasks that are ready to execute, the task with the highest priority is executed. The priority-driven scheduling techniques can in turn be categorized as follows [DB11].
  - **Task-static priority scheduling.** Each task has a single static priority that applies to all of its jobs, i.e., every job of a task gets the same priority as that of the task. Rate Monotonic scheduling (RM) [LL73] is an example of such a scheduling.
  - **Job-static priority scheduling.** Different jobs of the same task may have different priorities, but each job has a single static priority. An example of such a scheduling technique is Earliest Deadline First (EDF) [LL73].
  - **Dynamic-priority scheduling.** A job of a task may have different priorities at different times. Least Laxity First (LLF) [Mok83a] is an example for this category.

This research considers job-static priority scheduling.

Every scheduling approach in general falls into one of the two categories: *preemptive* or *non-preemptive* as defined below.

- **Preemptive scheduling:** In preemptive scheduling, a task executing on a processor can be forced by the scheduler to relinquish the processor before it completes execution (i.e., *preempted*) in order to execute some other ready-to-run higher priority task. The task that is interrupted is resumed sometime later for execution.
- **Non-preemptive scheduling:** In non-preemptive scheduling, a task executing on a processor will not be preempted and will therefore execute until completion.

This work considers preemptive scheduling.

Another way to categorize the scheduling techniques, especially the ones intended for multiprocessors is based on whether a task is allowed to migrate from one processor to another or not. This is describe next and is illustrated in Figure 1.6.

- **Fully-migrative scheduling.** In fully-migrative scheduling, tasks are not assigned/pinned to individual processors and all the processors are scheduled using a single algorithm. This scheduling technique is sometimes also referred to as *global scheduling*. Depending on the granularity of migration, fully-migrative scheduling approaches are categorized as follows:
  - **Task-level migration:** Different jobs of the same task may execute on different processors; however, each job must execute entirely on a single processor.
  - **Job-level migration:** Here, even a job may migrate during its execution and continue to execute on a different processor; however, the job cannot execute in parallel on more than one processor, i.e., a job can only execute on at most one processor at any time.
- **Non-migrative scheduling.** In non-migrative scheduling, each task is assigned to a single processor, on which each of its jobs will execute. In this model, each processor is scheduled independently. This scheduling technique is also referred to as *partitioned scheduling*.

For heterogeneous multiprocessor systems, we define another category, namely "intra-migrative" scheduling, as follows.

Definition 13 (**Intra-migrative scheduling.**). *In intra-migrative scheduling for heterogeneous multiprocessors, each task is statically assigned to a processor type and the jobs of that task will execute only on those processors that belong to the processor type to which the task is assigned. In other words, tasks/jobs assigned to a processor type can only migrate between processors of same type. Each processor type is scheduled independently.*

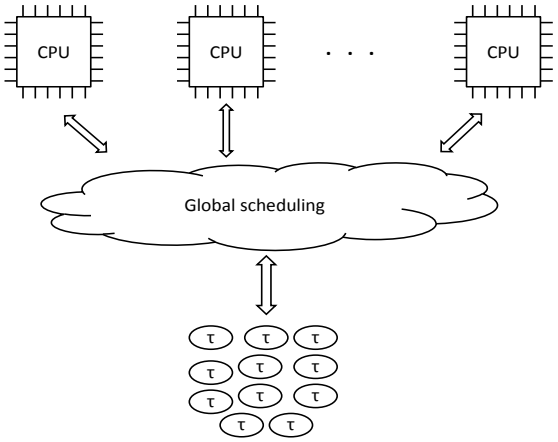
Figure 1.7 illustrates intra-migrative scheduling approach in heterogeneous multiprocessors.

This research focuses on both intra-migrative and non-migrative scheduling.

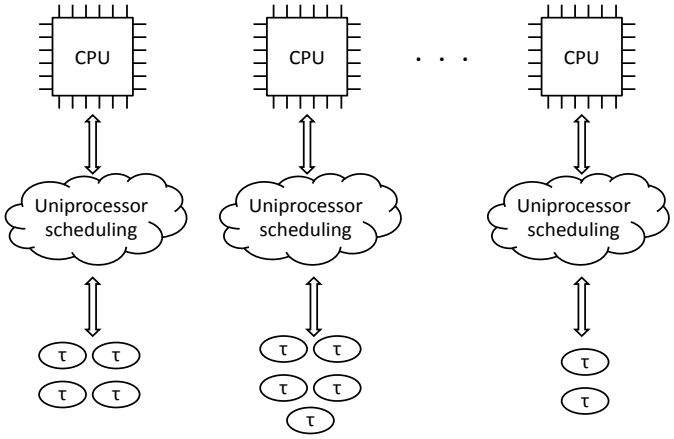
1.5 Background on real-time scheduling theory

The real-time scheduling theory has its origin as early as 1960s during the Apollo space mission in US. It was during this time, the first couple of real-time scheduling algorithms were designed, analyzed and were used to schedule the real-time workload on the on-board computer in the first manned space mission to the moon [LL73, Liu69].

The paper by Liu and Layland [LL73] is regarded as the foundational and most influential work in real-time scheduling theory. The paper addressed the problem of scheduling implicit-deadline periodic tasks with hard deadlines on a uniprocessor system. The paper presented two historical algorithms — one on *task-static priority scheduling* and another on *job-static priority scheduling*. These scheduling algorithms were preemptive in nature and made a couple of assumptions about



(a) Fully-migrative scheduling — no static assignment of tasks to processors; tasks/jobs can migrate between processors.



(b) Non-migrative scheduling — static assignment of tasks to processors; no migration is allowed.

Figure 1.6: A visualization of different categories of scheduling approaches.

the workload and the computing platform; some of the assumptions were: (i) tasks should be implicit-deadline periodic, (ii) tasks should be independent and hence not share any resources (except the processor) and (iii) computing platform should have a single processor.

The task-static priority algorithm proposed in [LL73] is referred to as Rate Monotonic (RM) algorithm and it assigns the priority based on the periods of the tasks — the priority of a task is inversely proportional to its period. Specifically, the task with the shortest period is given the highest priority and the task with the longest period is given the lowest priority. For this algorithm, following properties were shown. First, for any task set τ comprising n tasks, if it holds that the utilization of the task set $U_\tau \leq n \times \left(2^{\frac{1}{n}} - 1\right)$ then upon scheduling such a task set with RM guarantees that all deadlines are met. Observe that as n approaches infinity, the value of $n\left(2^{1/n} - 1\right)$ approaches $\ln(2)$ which is approximately 69%. Second, it was shown that RM is an *optimal* task-static priority algorithm with the interpretation that, if a task set can be scheduled to

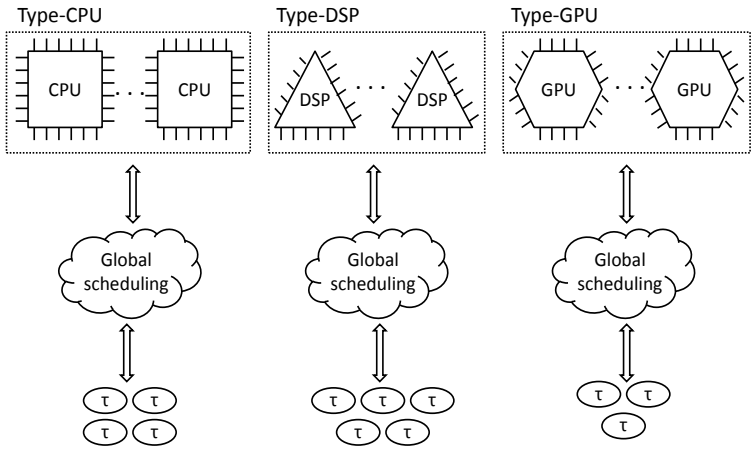


Figure 1.7: Intra-migrative scheduling on heterogeneous multiprocessors — static assignment of tasks to processor types; tasks/jobs can migrate between processors of same type.

meet all deadlines by any task-static priority algorithm then scheduling the same task set with RM will also guarantee that all deadlines are met.

The job-static priority algorithm proposed in [LL73] is referred to as Earliest-Deadline First (EDF) and it assigns the priority based on the deadlines of the tasks. The priorities assigned to tasks are inversely proportional to the deadlines of the ready-to-run jobs. For this algorithm, it was shown that, for any task set τ, if it holds that the utilization of the task set $U_\tau \leq 1$ then upon scheduling such a task set with EDF guarantees that all deadlines are met. Also, it has been shown that [LL73, Der74] EDF is an *optimal* job-static priority algorithm, in the sense that, if a task set can be scheduled to meet all deadlines by any job-static priority algorithm then scheduling the same task set with EDF will also guarantee that all deadlines are met.

Since then the real-time scheduling theory has evolved gradually by relaxing the assumptions made by Liu and Layland and imposing more constraints as per the requirements/demands of the new systems (e.g., embedded systems), applications (e.g., electronic software in cars, mobiles, etc) and architectures (e.g., multi-cores).

After that seminal paper, lot of research has been done on real-time scheduling on uniprocessor systems (e.g., [KAS93, BTW95, Leh90, SRL90, BMR90, BRH90, LW82, LSD89, SAA+04]). For example, the work in [KAS93, BTW95] provided a methodology for considering the preemption overhead into schedulability analysis that arises due to context switching, task queue manipulation and interrupt handling; the authors of [Leh90] provided an analysis technique for arbitrary-deadline tasks; the work in [SRL90] provided analysis techniques for systems where tasks can communicate/synchronize with each other and so on. Today, uniprocessor scheduling techniques are considered mature, taught in undergraduate courses worldwide and they are also adapted by the industry as well.

Although the multiprocessor real-time scheduling theory originated at the same time as that of uniprocessor (i.e., in the late 1960s and early 1970s), unfortunately, it did not experience similar

success. In 1969, Liu [Liu69] observed that: "*Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.*"

Dhall and Liu [Dha77], [DL78] observed that popular uniprocessor algorithms such as RM and EDF will not scale to multiprocessor when used as global scheduling algorithms and suffer from the so called "*Dhall effect*" leading to deadline misses even when the computational capacity requested by the workload from the underlying computing platform is nearly 0%. The following example illustrates this scenario.

Example 5. *Consider a system with n tasks and m processors ($n > m$) as shown in Table 1.1.*

Tasks	$\mathbf{C_i}$	$\mathbf{T_i = D_i}$
τ_1	2ε	1
τ_2	2ε	1
...	...	1
τ_{n-1}	2ε	1
τ_n	1	$1+\varepsilon$

Table 1.1: An example to illustrate *Dhall effect* — deadlines can be missed on multiprocessors even though the computational capacity requested is nearly 0%.

If the task set is globally scheduled using either RM or EDF algorithms, the task τ_n with period $1+\varepsilon$ will miss the deadline. (The exact number of processors is irrelevant here since task τ_n will miss deadline as long as the number of processors m is less than the number of tasks m). Note that the total utilization of this task set is: $U_\tau = (n-1) \times 2\varepsilon + \frac{1}{1+\varepsilon}$ and as $\varepsilon \to 0$, $U_\tau \to 1$. Thus, even with a multiprocessor system, i.e., $m > 1$, a task set with a utilization just above 1 may not be schedulable to meet all deadlines.

Therefore, in those days, the research efforts were mostly focused on the partitioned approaches where tasks are assigned to individual processors and then a well-known uniprocessor scheduling algorithm is used on each processor to schedule the respective tasks. However, from 1997 onwards, when Phillips et. al. [PSTW97] showed that the "Dhall effect" was more to do with task sets having some tasks with high utilization than the nature of global scheduling algorithms itself, there was a renewed interest in global scheduling policies [RSS90, BCPV96, AJ03, ABJ01, GFB03, BCL05, BCL09] along with the continued interest in partitioned scheduling [OS95, LBOS95, LGDG03, LDG04, FBB06, BF07b]. See the surveys [CFH+04, DB11] for a more comprehensive list of contributions in multiprocessor scheduling.

Most of the multiprocessor scheduling theory has been developed for identical [FBB06, LDG04, BCL05, GFB03] and uniform multiprocessors [FB03, CG06, DJ06, HS86] and only a few results are available for heterogeneous multiprocessors [HS76, LST90, Bar04c, Bar04b, Bar04a, CSV12, WBB13]. This is unfortunate because many chip manufacturers offer heterogeneous multicores

these days [AMD13a, Int13b, Nvi12, App13, Qua13, Sam13a, ST 12, Tex13]. Hence, in this work, we focus on heterogeneous multiprocessor scheduling.

In a heterogeneous multiprocessor system, fully-migrative scheduling algorithms that allow tasks to migrate between processors of different types [Bar04a] are hard to realize (if not impossible as shown in [DVT12]) since processors with different functionalities (i.e., processors of different types) typically have different instructions sets, register formats, etc. Hence, the problem of assigning tasks to processors (to processor types, respectively) and then scheduling them with an uniprocessor scheduling algorithm (an identical multiprocessor scheduling algorithm, respectively) is of much greater practical significance. This however requires that the following two sub-problems be solved: (i) assigning tasks to processors (to processor types, respectively) and (ii) once tasks are assigned, performing uniprocessor scheduling on each processor (identical multiprocessor scheduling on each processor type, respectively). The latter problem is well-understood: it can be performed with an optimal uniprocessor scheduling algorithm such as EDF [LL73] for example (an optimal identical multiprocessor scheduling algorithm such as DP-Fair [LFS$^+$10], respectively). But assigning tasks to processors is the difficult part (to avoid tedium, we will only speak about assigning tasks to processors in the rest of this paragraph). Several approaches for assigning real-time tasks to processors are available but for achieving provably good performance, only the following classes are known

- **Bin-packing schemes:** Bin-packing schemes are popular for assigning tasks to processors but unfortunately the proof techniques used on identical multiprocessors do not easily translate to heterogeneous multiprocessors and consequently, no bin-packing schemes exist for assigning real-time tasks on heterogeneous multiprocessors. Hence, in this work, we propose task assignment techniques that based on bin-packing heuristics and show how to prove the performance of these algorithms for heterogeneous multiprocessor model.

- **Integer-Linear-Programming (ILP) modeling:** The problem of assigning tasks to processors is modeled as Zero-One Integer-Linear-Programming (ILP). Such a formulation can be solved directly but it has the drawback of having a large computational complexity; the decision problem ILP is NP-complete. Through relaxation of the ILP formulation to LP and the use of certain tricks [Pot85], it is possible however to design an approximation scheme [LST90, Bar04b, Bar04c] which runs in polynomial time. It must solve linear-programming formulations so the degree of the polynomial is unfortunately high. Hence, we propose algorithms with low-degree polynomial time-complexity that do not rely on ILP modeling techniques. Also, we propose algorithms based on ILP modeling and its relaxation to LP but for such algorithms, we show how to prove a better performance guarantee (as compared to state-of-the-algorithms [LST90, Bar04b, Bar04c]).

- **Dynamic programming techniques:** Using dynamic programming techniques, it is possible to design algorithms that can assign the tasks in polynomial time, to any desired degree of accuracy [WBB13, HS76] — referred to as *polynomial time approximation schemes*.

However, the practical significance of such algorithms is severely limited since these polynomial time approximation schemes generally incur a very high time-complexity as the constants in the run-time expression for these algorithms are excessively large [CB11]. Hence, in this work, by combining the dynamic programming technique and the bin-packing heuristic, we show how to obtain a polynomial time approximation scheme which is efficient to be usable in practice.

Overall, in this work, we aim to design (low-degree) polynomial time-complexity algorithms with provably good performance for the problem of task assignment on both t-type and two-type heterogeneous multiprocessors, which are becoming increasingly relevant [AMD13a, Int13b, Nvi12, App13, Qua13, Sam13a, ST 12, Tex13] for the reasons stated earlier.

Chapter 2

Overview of This Research

This chapter gives an overview of this research. It formally defines the problem considered and briefly discusses its hardness. Then, some of the assumptions that are common across all the algorithms proposed here are stated. Also, the performance metrics used to evaluate the proposed algorithms are discussed. Finally, the contributions and significance of this work is summarized.

Organization of the chapter. Section 2.1 defines the problems studied in this work and Section 2.2 discusses the hardness of these problems. Section 2.3 describes why studying heterogeneous multiprocessors with a constant number of distinct processor types (i.e., both two-type and the generic t-type) is relevant. Section 2.4 lists some of the assumptions and Section 2.5 defines the metrics that are used to quantify the performance of the newly proposed algorithms. Section 2.6 lists the contributions and significance of this work and finally Section 2.7 gives an overview on how the rest of the thesis is organized.

2.1 Problem definition

This work considers the following problem:

Problem Definition [Task scheduling problem]. *Given a set of implicit-deadline sporadic tasks and a heterogeneous multiprocessor platform with a constant number of distinct processor types (i.e., both 2-type and generic t-type), design efficient algorithms to schedule the given tasks on the given platform so as to meet all the deadlines.*

As mentioned earlier, in heterogeneous multiprocessors, achieving task migration between different types is hard to achieve (if not impossible as discussed by [DVT12]) as different processor types typically differ in their register formats, instruction sets, etc. Hence the problem of assigning tasks to processors (to processor types, respectively) and then scheduling them with an optimal uniprocessor scheduling algorithm (optimal identical multiprocessor scheduling algorithm, respectively) is of much greater practical significance. Hence, in this work, we consider non-migrative and intra-migrative scheduling. With these approaches, the scheduling problem translates to task assignment problem as described next.

In the *non-migrative* scheduling (sometimes referred to as *partitioned* scheduling in the literature), every task is statically assigned to a processor before run-time and all its jobs must execute only on that processor at run-time. The challenge is to find, before run-time, a *task-to-processor* assignment such that, at run-time, on each processor, the given uniprocessor scheduling algorithm meets all deadlines of the tasks assigned on that processor. Scheduling tasks to meet deadlines on single processor systems is a well-understood problem. One may use Earliest Deadline First (EDF) [LL73] on each processor, for example. EDF is an *optimal* scheduling algorithm on a uniprocessor system [LL73, Der74], with the interpretation that, for every *valid* job arrival pattern, if a schedule exists that meets all deadlines then EDF always succeeds to construct such a schedule in which all the deadlines are met. Therefore, assuming that such an optimal scheduling algorithm is used on every processor, the challenging part is to find a task-to-processor assignment such that, *there exists* a schedule that meets all deadlines — such an assignment is said to be *feasible* assignment hereafter. Hence, in non-migrative model, the problem of scheduling the tasks translates to the problem of assigning tasks to individual processors.

In the *intra-migrative* scheduling, every task is statically assigned to a *processor type* before run-time, rather than to an individual processor. Then, the jobs of each task can migrate at run-time from one processor to another as long as these processors are of the same type (to which the task is assigned). Similar to the non-migrative model, once tasks are assigned to processor types, scheduling them to meet all deadlines under the intra-migrative model is well-understood, e.g., one may use an optimal identical multiprocessor scheduling algorithm, such as, ERfair [AS00], DP-Fair [LFS+10] or U-EDF [NBN+12]. Once again, assuming that such an optimal scheduling algorithm is used for scheduling tasks on processors of each type, the challenging part is to find a *feasible task-to-processor-type* assignment such that, *there exists* a schedule that meets all deadlines. Hence, in intra-migrative model, the problem of scheduling the tasks translates to the problem of assigning tasks to processor types.

With this information, the task scheduling problem can be re-written as follows:

Rewriting the Problem Definition [Task assignment problem]. *Given a set of implicit-deadline sporadic tasks and a heterogeneous multiprocessor platform with a constant number of distinct processor types (either two-type or generic t-type), design efficient algorithms to assign tasks to processors (to processor types, respectively) such that* "there exists" *a schedule (for every valid job arrival pattern) that meets all deadlines. The problem of assigning tasks to processors is referred to as* ***non-migrative task assignment problem*** *and the problem of assigning tasks to processor types is referred to as* ***intra-migrative task assignment*** *problem.*

We also study a variant of the task scheduling problem which is referred to as *shared resource scheduling* problem. In this variant, tasks share some resources such as data structures, sensors, etc. in addition to sharing processors. Tasks must operate on such resources in a *mutually exclusive* manner while accessing the resource, that is, at all times, when a job of a task holds a resource, no other job of any task can hold that resource. The problem is stated below.

New Problem Definition [Shared resource scheduling]. *We consider the problem of scheduling a task set of implicit-deadline sporadic tasks to meet all deadlines on a heterogeneous multiproces-*

sor platform (either two-type or generic t-type) where a task may access multiple shared resources in a mutually exclusive manner. In other words, this is the "task assignment problem" (defined earlier) with an additional constraint that the tasks may share some resources in a mutually exclusive manner.

To summarize, with the larger goal of understanding the real-time scheduling on heterogeneous multiprocessors, we study the intra-migrative and non-migrative task assignment problems along with the shared resource scheduling problem on both two-type and t-type heterogeneous multiprocessors.

2.2 Hardness of the problem

The problems under consideration are shown to be intractable — informally, these problems are shown to be "hard" to solve. In other words, for both the task assignment problems (i.e., assigning tasks to processor types and assigning tasks to individual processors) as well as the shared resource problem, it turns out that that, it is not possible to design optimal algorithms with polynomial time-complexity unless P=NP. So, if one wishes to design an optimal algorithm for any of these problems, it is indeed possible but such an algorithm will have an exponential time-complexity. Such exponential time-complexity algorithms are not scalable, in the sense that, for problem instances with large number of tasks and processors, these algorithms may take ages to complete their execution. Hence, for such "hard" problems, it is desirable to design (non-optimal) *polynomial-time* complexity algorithms and also to provide a bound on how much worse it performs, compared to an optimal scheme. Therefore, it is essential to understand whether the problems under consideration fall in this (intractable) class. Depending on the hardness, a problem can be categorized in one of the many well-defined classes [KV06]. The problems under consideration can be categorized as follows. The intra-migrative task assignment problem on two-type heterogeneous multiprocessors can be categorized as *NP-Complete* and all the other problems (intra-migrative task assignment problem on t-type, non-migrative task assignment problem on both two-type and t-type and shared resource scheduling problem on both two-type and t-type) can be categorized as *NP-Complete in the strong sense*. Informally, (i) both the classes of problems (i.e., NP-Complete and NP-Complete in the strong sense) are difficult, (ii) no optimal algorithm with a polynomial time-complexity can be designed for both the classes of problems unless P=NP and (iii) NP-Complete in the strong sense is more difficult than NP-Complete. There has been a significant effort in studying problems of these classes and other classes (for example, see [GJ78, GJ79, Sip96, KV06, AB09]) — interested reader can refer to such works which also define these terms more precisely.

Coming back to the problems under consideration, the above mentioned claims about the hardness of the problems are formally proven in subsequent chapters. Here, we only give the intuition.

Intra-migrative task assignment problem on two-type platforms. This problem is equivalent to the problem of assigning tasks to two processors, each of different type, such that each processor is used at most 100% of its capacity. Even the simpler instance of this problem,

in which tasks must be assigned to *two* identical processors, is known to be *NP-Complete* (Theorem 18.1 in [KV06], p. 426). So, this result continues to hold for two-type platforms as well. This is formally proven in Chapter 3.

Non-migrative task assignment problem on two-type platforms. Even in the simpler case of identical multiprocessors, the problem of assigning tasks to individual processors is shown to be *NP-Complete in the strong sense* [Joh73]. So, this result continues to hold for two-type platforms as well. This is formally proven in Chapter 4.

Shared resource scheduling on two-type platforms. It can be seen that a restricted version of this problem in which tasks do not share any resources is equivalent to the problem of assigning tasks to individual processors which is *NP-Complete in the strong sense* (as mentioned above). Hence this result continues to hold for the shared resource problem as well. This is formally proven in Chapter 5.

Intra-migrative task assignment problem on t-type platforms. Even in the simpler case, in which each processor type has only one processor, finding a feasible task-to-processor-type assignment is *NP-Complete in the strong sense* (since the restricted version of this special case in which all the processors are identical is NP-Complete in the strong sense [Joh73]). Hence, this result continues to hold for t-type platforms having one or more processors of each type as well. This is discussed in Chapter 6.

Non-migrative task assignment problem on t-type platforms. Even in the simpler case of two-type heterogeneous multiprocessors, the problem of assigning tasks to individual processors is *NP-Complete in the strong sense* (as discussed earlier). So, this result continues to hold for t-type heterogeneous multiprocessors as well. This is discussed in Chapter 7.

Shared resource scheduling on t-type platforms. Even in the simpler case of two-type heterogeneous multiprocessors, this problem is known to be *NP-Complete in the strong sense*. Hence this result continues to hold for t-type heterogeneous multiprocessors as well. This is formally proven in Chapter 8.

2.3 Why study heterogeneous multiprocessors?

The heterogeneous multiprocessor computing platform is a more generic computing platform than identical and uniform platforms. In other words, identical and uniform multiprocessors are special case of heterogeneous multiprocessors. Hence, it is interesting to study heterogeneous multiprocessors since a solution designed for heterogeneous multiprocessors can also be applied to identical and uniform multiprocessors. In practice, many chip makers offer chips having a constant number of distinct types of processors. For example, AMD [AMD13a], Apple [App13], Intel [Int13c, Int13b], NVIDIA [Nvi12], Qualcomm [Qua13], Samsung [Sam13a], ST Microelectronic [ST 12], TI [Tex13] offer such chips. Traditionally, processors of the first type were

meant for general purpose computations and processors of the second type (respectively, third type, fourth type and so on) were meant for special purpose computations such as graphics processing (respectively, signal processing, network processing and so on), hence task assignment was trivial. Today though, designers use processors of the second type (and third type and so on) for wide range of computations and this makes task assignment non-trivial [Gee05]. Unfortunately, the literature does not provide any algorithm that takes advantage of this special structure. Hence, in this work, we consider heterogeneous multiprocessors with a constant number of distinct processor types (i.e., both two-type and generic t-type). Also, we believe that studying the generic t-type heterogeneous multiprocessors (in which $t \geq 2$) provides a better understanding of the problem and the solutions to such a generic model will cater to complex systems in near future.

2.4 Common assumptions

This section lists some of the assumptions that this work makes and hence these assumptions hold for the rest of the discussion in this thesis. This list of assumptions is not comprehensive and some of the assumptions specific to each proposed algorithm are listed in the respective chapters. The assumptions that are common for all the chapters are listed below.

- *Implicit-deadline sporadic tasks.* The tasks considered in this work are implicit-deadline sporadic tasks, that is, for each task, the deadline of the task is equal to its minimum inter-arrival time.

- *Heterogeneous multiprocessors.* The computing platform considered in Part II (i.e., Chapters 3-5) is two-type heterogeneous multiprocessor in which each processor is either of type-1 or of type-2. The computing platform considered in Part III (i.e., Chapters 6-8) is t-type heterogeneous multiprocessor in which each processor belongs to one and only one of the $t \geq 2$ types.

- *No parallel execution.* A task cannot execute in parallel, i.e., at any time instant, it can be executing on at most one processor.

- *Impact of shared hardware resources.* The shared hardware resources such as shared cache and memory bus are not modeled as part of the computing platform. Hence, the impact of such shared hardware resources on the execution behavior of a task is ignored.

- *(In)dependent tasks.* Majority of this work assumes that the tasks are independent, i.e., they do not share any resources such as data structures and do not have any data dependency. It is only in Chapter 5 and Chapter 8 that we consider dependent tasks, i.e., tasks that share resources such as data structures, sensors, etc.

2.5 Performance metrics

This section describes the performance metrics used in this work to quantify the performance of the newly proposed algorithms.

Commonly, the performance of a real-time scheduling algorithm is characterized using the notion of *utilization bound* [LL73]. The utilization bound of an algorithm is a number such that if the utilization of the task set (see Definition 7 in Chapter 1 on page 11) is no greater than this number then all deadlines will be met when the task set is scheduled using this algorithm. More formally, it is defined as follows.

Definition 14 (**Utilization bound of an algorithm**). *An algorithm $\mathscr{A}$ is said to have an utilization bound of* UB *only if it is capable of scheduling any task set with an utilization of up to* UB *such that all deadlines are met.*

This metric has been used to evaluate scheduling algorithms on uniprocessor [LL73], *identical* multiprocessors [ABJ01, GSYY10, OB98] and *uniform* multiprocessors [DJ06]. However, it does not translate to algorithms on *heterogeneous* multiprocessors. This is because on heterogeneous multiprocessors, each task is characterized by as many utilizations as the number of distinct processor types and hence the syntax and semantics of *utilization of a task set* is not clear in this context as of today. Hence we rely on the *speed competitive ratio* (also referred to as *resource augmentation* [PSTW97] and *speedup factor* [Bar13, WBB13] in literature) to characterize the performance of the algorithm under design.

2.5.1 Performance metric: Speed competitive ratio

The *speed competitive ratio* is an alternative method of comparing the performance of an algorithm $\mathscr{A}$ with that of an optimal algorithm or class of algorithms. In this context, we first define the term "*adversary*" as follows.

Definition 15 (**Adversary**). *The adversary is the optimal algorithm or the class of optimal algorithms against which the performance of an algorithm $\mathscr{A}$ is evaluated.*

We define the speed competitive ratio of an algorithm as follows.

Definition 16 (**Speed competitive ratio of a task assignment algorithm**). *We define the speed competitive ratio* $\text{SCR}_{\mathscr{A}}$ *of an algorithm $\mathscr{A}$ against an adversary, as the lowest number such that, for every task set and computing platform, it holds that: if it is possible for the adversary to meet all deadlines of the task set on the computing platform then algorithm $\mathscr{A}$ succeeds to output an assignment of tasks that meets all deadlines of the task set as well but given a platform, in which every processor is* $\text{SCR}_{\mathscr{A}}$ *times faster than the corresponding processor in the platform used by the adversary.*

A low speed competitive ratio indicates high performance; the best achievable is *one* (which reflects the optimal algorithm for a given problem). If a scheduling algorithm has an infinite

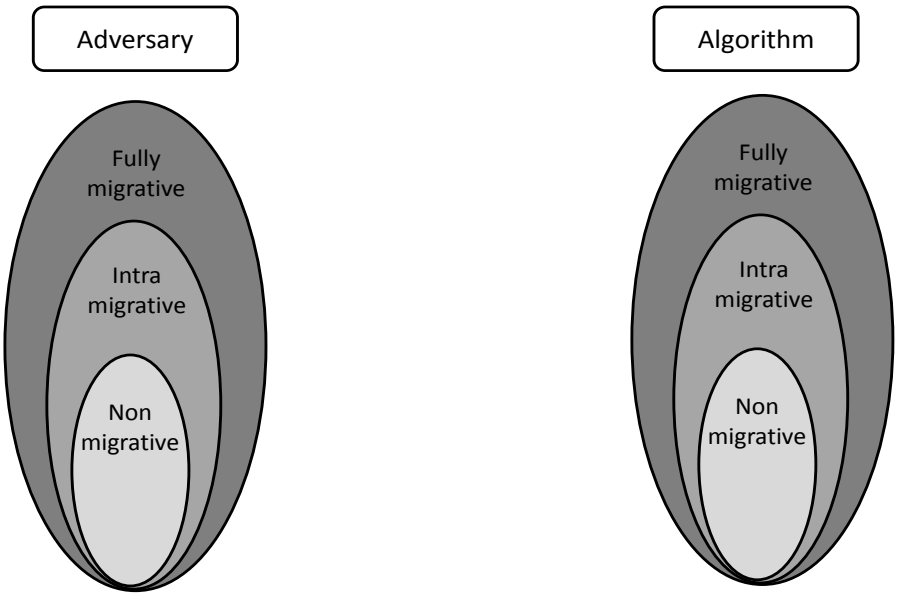


Figure 2.1: Different adversaries that are referred in the subsequent parts of this report. Also, corresponding categories of algorithms that can be designed are also listed. In this research, we will not discuss any fully-migrative algorithms. However, we will use all the three categories of adversaries shown here to quantify the performance of the algorithms.

speed competitive ratio then a task set exists which could be scheduled (by adversary) to meet deadlines but would miss deadlines with the actually used algorithm even if processor speeds were multiplied by an "infinite" factor. Therefore, a scheduling algorithm with a *finite* (ideally small) speed competitive ratio is desirable because it can ensure the designer that deadlines will be met by using faster processors. Consequently, the real-time systems community has embraced the development of scheduling algorithms with finite speed competitive ratio, for example, see [AT07b, BF07b, DRBB09].

This research uses the speed competitive ratio as one of the performance metrics.

The task assignment algorithm $\mathscr{A}$ can be non-migrative or intra-migrative or fully-migrative. Similarly, the adversary can be non-migrative or intra-migrative or fully-migrative as well. This is shown in Figure 2.1. As shown in the figure, the class of non-migrative algorithms is strictly contained in the class of intra-migrative algorithms which in turn is strictly contained in the class of fully-migrative algorithms. This is because, every non-migrative assignment is also an intra-migrative assignment (but vice versa is not true) which in turn is also a fully-migrative assignment (but vice versa is not true).

Relative Powerfulness of the adversaries: We say that the fully-migrative model is more powerful than the intra-migrative model which in turn is more powerful than the non-migrative model, in the sense that, (i) a non-migrative solution can always be transformed into an intra-migrative solution and similarly, an intra-migrative solution can always be transformed into a

fully-migrative solution whereas (ii) a fully-migrative solution cannot always be transformed into an intra-migrative solution and similarly, an intra-migrative solution cannot always be transformed into a non-migrative solution. So, the relation of these models can be depicted as shown in Figure 2.1 which reflects that the non-migrative model is contained in the intra-migrative model which in turn is contained in the fully-migrative model.

Comparison of an algorithm with an adversary. Recall that, this thesis deals with designing non-migrative and intra-migrative task assignment algorithms (apart from shared resource scheduling problem, which will be discussed in detail in Chapter 5 and Chapter 8 separately). In this context, it is important to know the adversaries against which the speed competitive ratio of such algorithms can be quantified. This is illustrated in Figure 2.2. As shown in Figure 2.2a, *the speed competitive ratio of an intra-migrative algorithm can be expressed against an equally powerful intra-migrative adversary or more powerful fully-migrative adversary*. Similarly, as shown in Figure 2.2b, *the speed competitive ratio of a non-migrative algorithm can be expressed against an equally powerful non-migrative adversary or a more powerful intra-migrative adversary or even more powerful fully-migrative adversary*.

Remark about the notations. In the rest of the thesis, the speed competitive ratio of an algorithm against an adversary, for example, of a non-migrative algorithm against a non-migrative adversary, will be stated as follows: **"The speed competitive ratio of a non-migrative algorithm $\mathscr{A}$ is** $\text{SCR}_{\mathscr{A}}$ **against the non-migrative adversary"**. For any other combinations of algorithms and adversaries, the speed competitive ratio is expressed in a similar manner.

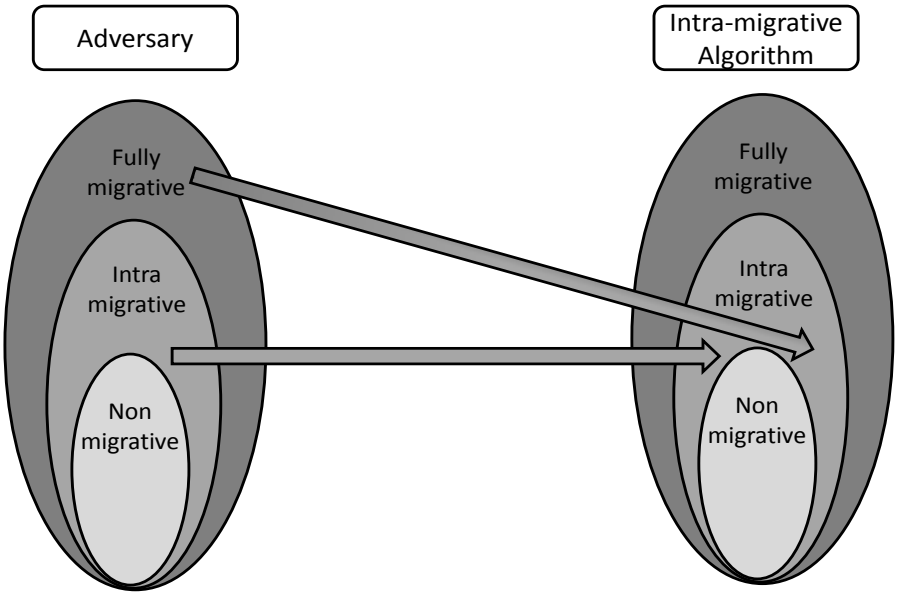
2.5.2 How do we use the speed competitive ratio metric

The speed competitive ratio can be used as a metric to compare different algorithms. We now illustrate how this metric can be used to decide which of the two algorithms, say $\mathscr{A}_1$ and $\mathscr{A}_2$, has a better performance. Obviously, both these algorithms must be of same category for a fair comparison, for example, let us say both $\mathscr{A}_1$ and $\mathscr{A}_2$ are non-migrative algorithms. There are two cases to be considered as described below.

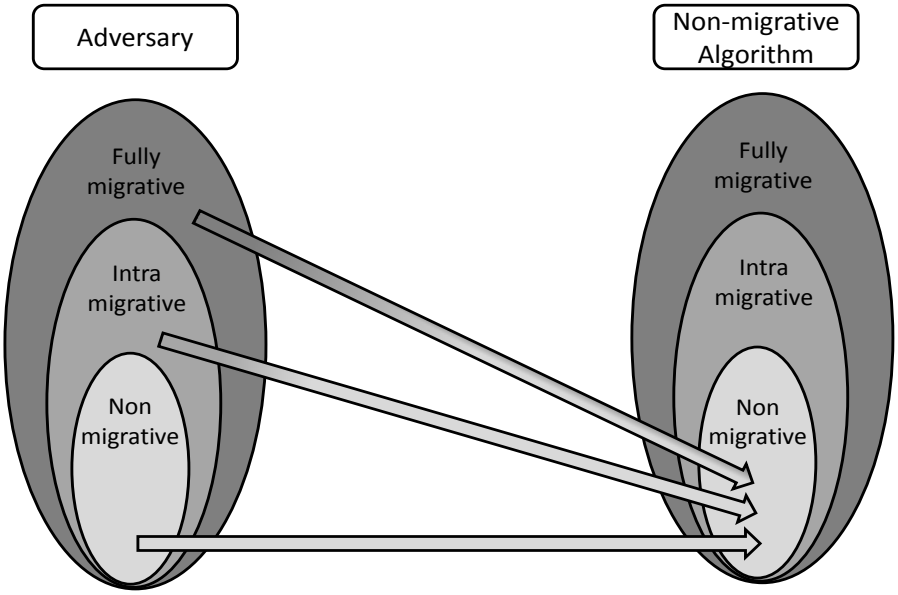
- **When the adversaries are same.** When their adversaries are the same, the algorithm with a lower speed competitive ratio is said to have a better performance guarantee. This is illustrated in Table 2.1.

- **When the adversaries are different.** In this case, if the speed competitive ratio of an algorithm, say $\mathscr{A}_1$, with a more powerful adversary is no greater than the speed competitive ratio of the other algorithm, say $\mathscr{A}_2$, (which has a weaker adversary than $\mathscr{A}_1$) then $\mathscr{A}_1$ is said to have a better performance than $\mathscr{A}_2$; otherwise, nothing can be inferred about their performance. This is illustrated in Table 2.2.

As can be seen from the last row of Table 2.1, for the case when the adversaries of both the algorithms are the same and their speed competitive ratios are same as well, using speed competitive ratio alone is not sufficient to determine which algorithm offers a better performance



(a) An intra-migrative algorithm — adversary can be intra-migrative or fully-migrative.



(b) A non-migrative algorithm — adversary can be non-migrative or intra-migrative or fully-migrative.

Figure 2.2: Comparison of (intra-migrative and non-migrative) algorithms against different adversaries — an algorithm can only be compared against either an equally powerful adversary or a more powerful adversary.

When the adversaries of both the algorithms are same	
$\text{SCR}_{\mathscr{A}1} < \text{SCR}_{\mathscr{A}2}$	Algorithm $\mathscr{A}_1$ is said to have a better performance than algorithm $\mathscr{A}_2$
$\text{SCR}_{\mathscr{A}1} > \text{SCR}_{\mathscr{A}2}$	Algorithm $\mathscr{A}_2$ is said to have a better performance than algorithm $\mathscr{A}_1$
$\text{SCR}_{\mathscr{A}1} = \text{SCR}_{\mathscr{A}2}$	Both the algorithms, $\mathscr{A}_1$ and $\mathscr{A}_2$, are said to have the same performance

Table 2.1: Comparison of algorithm $\mathscr{A}_1$ with a speed competitive ratio $\text{SCR}_{\mathscr{A}1}$ and algorithm $\mathscr{A}_2$ with a speed competitive ratio $\text{SCR}_{\mathscr{A}2}$ when their adversaries are the same. Note that both the algorithms are of same category, for example, non-migrative.

When the adversary of $\mathscr{A}_1$ is more powerful than the adversary of $\mathscr{A}_2$	
$\text{SCR}_{\mathscr{A}1} \leq \text{SCR}_{\mathscr{A}2}$	Algorithm $\mathscr{A}_1$ is said to have a better performance than algorithm $\mathscr{A}_2$
$\text{SCR}_{\mathscr{A}1} > \text{SCR}_{\mathscr{A}2}$	Nothing can be inferred about which algorithm has a better performance

Table 2.2: Comparison of algorithm $\mathscr{A}_1$ with a speed competitive ratio $\text{SCR}_{\mathscr{A}1}$ and algorithm $\mathscr{A}_2$ with a speed competitive ratio $\text{SCR}_{\mathscr{A}2}$ when their adversaries are different. Specifically, the adversary of $\mathscr{A}_1$ is more powerful than the adversary of $\mathscr{A}_2$. Note that both the algorithms are of same category, for example, non-migrative.

guarantee. To resolve the tie in such cases, we use two more metrics which are discussed in the next section.

2.5.3 Performance metrics: Time-complexity and Necessary multiplication factor

In this section, we describe two more metrics that are used in this work to quantify the performance of the algorithms. First metric is the *time-complexity* of the algorithm which indicates approximately how much time does an algorithm take to execute as a function of the input size and the second metric is the *necessary multiplication factor* which indicates, for a given problem instance (a task set and a heterogeneous platform), how much faster processors the algorithm needs in order to succeed. This factor is always upper bounded by the speed competitive ratio of the algorithm and in this work, this is observed via simulations. We now present these metrics in detail and also discuss how they are used.

2.5.3.1 Time-Complexity of the algorithm

How much time does a given task assignment algorithm takes to output the solution? It could possibly take a very long time on large inputs (that is many task sets and processors) to give a task assignment; it is not desirable to wait indefinitely or for years to obtain such a task assignment! So, it makes sense to be able to estimate the running time of the algorithm apriori. However, it is not necessary to know the exact execution time of the algorithm and it only suffices to know the *approximate* execution time. One way to do this is by quantifying the amount of time taken by the

algorithm as a function of the length of the string representing the input — generally referred to as the *time-complexity* of the algorithm. For example, the time-complexity of a task assignment algorithm can be expressed as a function of its inputs, say as a function of the number of tasks n and the number of processors m.

The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. The worst-case time-complexity of an algorithm can be and is generally expressed using big O notation, which excludes coefficients and lower order terms [CLRS01]. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$.

An algorithm is said to be of *polynomial time-complexity* if its running time is upper bounded by a polynomial expression in the size of the input of the algorithm. A task assignment algorithm with a time-complexity of $O(n^k)$ for some constant k is an example of the polynomial time-complexity algorithm.

2.5.3.2 How do we use the time-complexity metric

We make use of the time-complexity of the algorithm in this work as follows.

- We aim to design algorithms with polynomial time-complexity, preferably low-degree polynomial.
- While comparing two algorithms, for the case when their adversaries as well as their speed competitive ratios are same, the algorithm with a lower-degree polynomial time-complexity among the two is said to have a better performance than the other.

This research uses time-complexity of the algorithm as one of the performance metrics.

2.5.3.3 Necessary multiplication factor of the algorithm

Recall that speed competitive ratio of an algorithm holds for *any task set* and computing platform, in the sense that, for any task set, if it is possible for the adversary to schedule the tasks on the computing platform to meet all deadlines then an algorithm $\mathscr{A}$ with a speed competitive ratio $\text{SCR}_{\mathscr{A}}$ succeeds in finding a feasible task assignment on a platform in which every processor is $\text{SCR}_{\mathscr{A}}$ times faster. In other words, there exists no task set such that the adversary can schedule it on a computing platform to meet all deadlines but algorithm $\mathscr{A}$ will fail to do it on a platform which is $\text{SCR}_{\mathscr{A}}$ times faster. However, it may happen that, for a given task set, algorithm $\mathscr{A}$ might succeed in finding a feasible task assignment on a platform which is *less than* $\text{SCR}_{\mathscr{A}}$ times faster (of course, this platform cannot be slower than the one used by the adversary). The *necessary multiplication factor* captures this behavior.

Definition 17 (Necessary multiplication factor). *For a given task set, we define the necessary multiplication factor* $\text{NMF}_{\mathscr{A}}$ *of an algorithm* $\mathscr{A}$ *as the minimum amount of extra speed of processors that algorithm* $\mathscr{A}$ *needs, so as to succeed in finding a feasible task assignment as compared to adversary. This factor is always upper bounded by the speed competitive ratio* $\text{SCR}_{\mathscr{A}}$ *of algorithm* $\mathscr{A}$.

Speed competitive ratio vs. Necessary multiplication factor. The speed competitive ratio is a property of an algorithm whereas the necessary multiplication factor is a property of an algorithm but for a given problem instance (tasks and processors). Informally, the necessary multiplication factor can be viewed as the speed competitive ratio of an algorithm for a given problem instance. In the context of this work, the speed competitive ratio of an algorithm is a theoretically derived value but the necessary multiplication factor of the algorithm is an observed value via simulations. Also, note that for a given problem instance, the necessary multiplication factor of an algorithm is always upper bounded by its speed competitive ratio, i.e., for a given task set τ and an algorithm $\mathscr{A}$, it always holds that: $\text{NMF}_{\mathscr{A}} \leq \text{SCR}_{\mathscr{A}}$.

2.5.3.4 How do we use the necessary multiplication factor metric

We use the necessary multiplication factor metric to evaluate the average-case performance of the algorithms. We use this metric as described below.

- For the case when the adversaries of two algorithms are same and the speed competitive ratio of these algorithms are same as well, we use necessary multiplication factor to determine the algorithm with a better performance. A large number of problem instances (number of processors, number of tasks and their utilizations) are generated randomly. For each problem instance we run both algorithms and obtain their necessary multiplication factors for each problem instance. The algorithm that has low necessary multiplication factor for many task sets is said to exhibit a better average-case performance than the other.
- The necessary multiplication factor is also used for *stand-alone evaluation* of the algorithm. Specifically, it is used to evaluate the average-case performance behavior of the algorithm. The problem instances are generated randomly and the algorithm is run to obtain the necessary multiplication factor for each problem instance. Then, for the majority of the task sets, if the algorithm has low necessary multiplication factor then the algorithm is said to exhibit a good average-case behavior.

This research uses necessary multiplication factor as one of the performance metrics.

2.6 Contributions and significance of this work

This work makes the following contributions to state-of-the-art in real-time scheduling theory. A more detailed description of the contributions will be given in subsequent chapters of this report.

C1. Intra-migrative scheduling on two-type platforms. For this problem, this work proposes a task assignment algorithm with a speed competitive ratio of 1.5 against an intra-migrative adversary. For intra-migrative scheduling on two-type platforms, no previous task assignment algorithm is known to exist and hence the proposed algorithm is the first of its kind[1].

C2. Non-migrative scheduling on two-type platforms. For this problem, this work proposes the following task assignment algorithms with finite speed competitive ratios.

C2.a A task assignment algorithm (and a couple of its variants) is proposed with a speed competitive ratio of 2 against non-migrative adversary. This is the first work to show how bin-packing heuristics can be used to design task assignment algorithms for two-type heterogeneous multiprocessors with a finite speed competitive ratio. The proposed algorithm has the same speed competitive ratio as the previously known algorithms; however, it outperforms these algorithms in average-case performance evaluations.

C2.b Another task assignment algorithm is proposed with a speed competitive ratio of 2 but against a more powerful intra-migrative adversary. This algorithm outperforms all the previously known task assignment algorithms for non-migrative scheduling (including the one mentioned in **C2.a** above).

C2.c A task assignment algorithm is proposed with a speed competitive ratio of 1.5 and in addition it requires three additional processors, compared to non-migrative adversary. This algorithm outperforms all the previously known task assignment algorithms for non-migrative scheduling (whose speed competitive ratios are derived against non-migrative adversary). This is because, for systems with large number of processors, the additional three processors become negligible and hence the speed competitive ratio of this algorithm tends to 1.5. Also, this is the first work to show how cutting planes can be used in linear programming to improve the speed competitive ratio of algorithms for provably good algorithms for assigning real-time tasks to processors so as to meet all deadlines.

C2.d Finally, a polynomial time approximation scheme (PTAS) for assigning tasks to processors is proposed as well. It has a speed competitive ratio of $1+3\varepsilon$ (where ε is an input parameter) against a non-migrative adversary. This algorithm combines dynamic programming techniques and bin-packing heuristics to obtain a polynomial time approximation scheme which is efficient to be usable in practice. Further, it outperforms the previously known PTAS in average-case performance evaluations.

C3. Shared resource scheduling on two-type platforms. For this problem, this work proposes an algorithm with a proven speed competitive ratio. For this problem, no previous algorithm is known to exist and hence the proposed algorithm is the first of its kind.

[1] Some of the non-migrative algorithms from state-of-the-art can be "adapted" to intra-migrative scenario, however, these "adapted" algorithms will either end up with a significantly higher time-complexity (which severely limits the practicality of these algorithms) or a higher speed competitive ratio compared to our proposed intra-migrative algorithm.

C4. Intra-migrative scheduling on t-type platforms. For this problem, a task assignment algorithm is proposed with a speed competitive ratio of $1 + \frac{t-1}{t}$ against an intra-migrative adversary where t is the number of distinct types of processors. For intra-migrative scheduling on t-type platforms, no previous task assignment algorithm is known to exist and hence the proposed algorithm is the first one[2].

C5. Non-migrative scheduling on t-type platforms. For this problem, a task assignment algorithm is proposed with a speed competitive ratio of 2 against a more powerful intra-migrative adversary. This algorithm outperforms all the previously known task assignment algorithms for non-migrative scheduling on t-type heterogeneous multiprocessors.

C6. Shared resource scheduling on t-type platforms. For this problem, this work proposes an algorithm with a proven speed competitive ratio. For this problem, no previous algorithm is known to exist and hence the proposed algorithm is the first of its kind.

2.7 Organization of the report

The rest of the report is organized as follows. Part II (i.e., Chapters 3–5) discusses in detail the work on two-type heterogeneous multiprocessors carried out as part of this dissertation. Then Part III (i.e., Chapters 6–8) discusses the work on t-type heterogeneous multiprocessors carried out as part of this dissertation. Finally Part IV (i.e., Chapter 9) presents some concluding remarks. Specifically, the rest of the thesis is organized as follows.

- Chapter 3 discusses the intra-migrative task assignment problem on two-type heterogeneous multiprocessors and proves its hardness. It then proposes an algorithm, namely SA, for this problem. The algorithms SA relies on a simple technique of sorting the tasks in a certain way and then assigning them one by one to processor types. The speed competitive ratio of SA is proven against an equally powerful intra-migrative adversary.

- Chapter 4 describes the non-migrative task assignment problem on two-type heterogeneous multiprocessors and proves its hardness. For this problem, it presents several algorithms, namely FF-3C, SA-P, LP-CUT and PTAS, in detail and proves their respective speed competitive ratios:

  - The algorithm FF-3C is based on bin-packing heuristics and its speed competitive ratio is proven against an equally powerful non-migrative adversary.
  - The algorithm SA-P is an extension of SA and its speed competitive ratio is proven against a more powerful intra-migrative adversary.

[2]Similar to two-type heterogeneous multiprocessors, although some of the non-migrative algorithms from state-of-the-art can be "adapted" to intra-migrative scenario, our algorithm performs better than the "adapted" algorithms either in terms of the time-complexity or in terms of the speed competitive ratio.

  - The LP-CUT algorithm makes use of cutting planes in linear programming formulation for assigning tasks. The speed competitive ratio of LP-CUT is proven against an equally powerful non-migrative adversary.
  - The PTAS_{NF} algorithm makes use of dynamic programming technique and bin-packing heuristics to output the task assignment. For PTAS_{NF}, the speed competitive ratio is proven as a function of an input parameter (such class of algorithms are referred to as *polynomial time approximation schemes*) against an equally powerful non-migrative adversary.

- Chapter 5 describes the shared resource scheduling problem on two-type heterogeneous multiprocessors and proves its hardness. It then presents an algorithm, namely FF-3C-vpr, for this problem which is based on the FF-3C algorithm, and proves its speed competitive ratio against an equally powerful adversary.

- Chapter 6 discusses the intra-migrative task assignment problem on t-type heterogeneous multiprocessors and its hardness. It then presents an algorithm, namely LPG_{IM}, for this problem. The LPG_{IM} algorithm relies on solving a linear program formulation and uses graph theory techniques for obtaining an intra-migrative task assignment. The speed competitive ratio of LPG_{IM} is proven against an equally powerful intra-migrative adversary.

- Chapter 7 describes the non-migrative task assignment problem on t-type heterogeneous multiprocessors and its hardness. For this problem, it presents an algorithm, namely LPG_{NM}, which is an extension of the intra-migrative algorithm, LPG_{IM}. The speed competitive ratio of LPG_{NM} algorithm is proven against a more powerful intra-migrative adversary.

- Chapter 8 describes the shared resource scheduling problem on t-type heterogeneous multiprocessors and its hardness. It then presents an algorithm, namely LP-EE-vpr, for this problem which is based in LP-EE algorithm from the state-of-the-art and proves its speed competitive ratio against an equally powerful adversary.

- Finally, Chapter 9 presents some concluding remarks by summarizing the results obtained in this research, discussing the implications of these results and briefly mentioning a couple of directions in which this research could be extended in the future.

Part II

Two-type Heterogeneous Multiprocessors

Chapter 3

Intra-migrative Scheduling on Two-type Heterogeneous Multiprocessors

3.1 Introduction

Recall that, on heterogeneous multiprocessor systems, scheduling algorithms that assume tasks can migrate between processors of different types are hard to design for many practical systems since processors of different types typically have different instructions sets, register formats, etc. This is because it is difficult to achieve task migration between processors of different types (if not impossible as shown in [DVT12]). Hence, we focus on studying scheduling approaches which do not assume such migrations (between processors of different types). The intra-migrative scheduling and non-migrative scheduling are two such approaches.

In this chapter, we consider the problem of intra-migrative scheduling of tasks on two-type heterogeneous multiprocessors. Recall that in the intra-migrative model, every task needs to be statically assigned to a *processor type* before run-time. Then during run-time, the jobs of each task can migrate from one processor to another as long as these processors are of the same type to which the task is assigned. Once all the tasks are assigned to processor types, by treating processors of each type as an identical multiprocessor platform, tasks assigned to each processor type are scheduled using a global scheduling algorithm (designed for identical multiprocessors) on the respective processor types. The global scheduling problem on identical multiprocessors is a well-studied topic — some *optimal* scheduling algorithms exist for this problem (such as ER-Fair [AS00], DP-Fair [LFS$^+$10], U-EDF [NBN$^+$12], etc.). These algorithms are optimal, in the sense that, for every valid job arrival pattern, if a schedule exists that meets all deadlines then these algorithms construct a schedule that meets all deadlines as well. So, in the intra-migrative model, once the tasks are assigned to processor types, one of these optimal scheduling algorithms can be used to schedule the tasks on each processor type. Hence, the challenging part is to find a *task-to-processor-type* assignment for which *there exists* a schedule (for every valid job arrival pattern) that meets all the deadlines — such an assignment is said to be *feasible* task-to-processor-type

assignment hereafter. This problem is equivalent to the problem of assigning tasks to two processors, each of different type, such that each processor is used at most 100% of its capacity. Even the simpler instance of this problem, in which tasks must be assigned to *two* identical processors, is NP-Complete (Theorem 18.1 in [KV06], p. 426). So, this result continues to hold for two-type platforms as well. Hence, in this work, we propose a non-optimal algorithm of low-degree polynomial time-complexity for this problem, for which no previous algorithm is known to exist. We also prove its speed competitive ratio against an equally powerful intra-migrative adversary.

Problem Statement. In this chapter, we consider the problem of intra-migrative scheduling of implicit-deadline sporadic tasks on two-type heterogeneous multiprocessors. That is, assuming that an optimal identical multiprocessor scheduling algorithm is used on processors of each type to schedule the tasks, we design an algorithm to determine a feasible assignment of tasks to processor types.

Related Work. The scheduling problem on heterogeneous multiprocessors has been studied in the past [HS76, Bar04b, Bar04c, CSV12, LST90, WBB13, JP99]. However, all these approaches [HS76, Bar04b, Bar04c, CSV12, LST90, WBB13, JP99] consider the problem of non-migrative scheduling, i.e., the problem of assigning tasks to individual processors and none of them consider the problem of intra-migrative scheduling in which tasks need to be assigned to processor types.

Contributions and Significance of The Work Discussed in This Chapter. We present a task assignment algorithm, called SA, which has a $O(n \log n)$ time-complexity and offers the following guarantee. Consider a two-type platform π and an implicit-deadline sporadic task set τ in which, for every task in τ, it holds that: (i) utilization of the task on processors of type-1 is either no greater than α or is greater than 1 and (ii) utilization of the task on processors of type-2 is either no greater than α or is greater than 1, where $0 < \alpha \leq 1$. If there exists a feasible intra-migrative assignment of τ on π (i.e., task-to-processor-type assignment) then, using SA, it is guaranteed to find such a feasible intra-migrative assignment of τ on a platform π' in which only *one* processor is $1 + \frac{\alpha}{2} \leq 1.5$ (since $0 < \alpha \leq 1$) times faster than the corresponding processor in π. For defining the speed competitive ratio of SA, we say that SA succeeds to find such a feasible intra-migrative assignment on a platform $\pi^{(1+\frac{\alpha}{2})}$, where $\pi^{(1+\frac{\alpha}{2})}$ is a two-type platform in which *every* processor is $1 + \frac{\alpha}{2}$ times faster than the corresponding processor in π; in other words, the speed competitive ratio of intra-migrative algorithm SA is $1 + \frac{\alpha}{2} \leq 1.5$ against equally powerful intra-migrative adversary. We also evaluate the average-case performance of algorithm SA by generating task sets randomly and measuring how much faster processors the algorithm needs (i.e., its necessary multiplication factor), for a given task set, in order to output a feasible intra-migrative task assignment.

We believe that the significance of this work is two-fold. First, for the problem of intra-migrative task assignment, no previous algorithm exists and hence our algorithm, SA, is the first for this problem[1]. Second, in our evaluations with randomly generated task sets, for the vast major-

[1] Some of the non-migrative algorithms from state-of-the-art (for example, the one presented in [HS76, LST90]) can be "adapted" to intra-migrative scenario, however, these "adapted" algorithms will either end up with a significantly higher time-complexity [HS76] (which severely limits the practicality of these algorithms) or a higher speed competitive ratio [LST90] compared to our SA algorithm.

ity of task sets, the necessary multiplication factor of our algorithm is observed to be significantly smaller than its speed competitive ratio.

A global view. The context of the new algorithm, SA, can be visualized as shown in Figure 3.1.

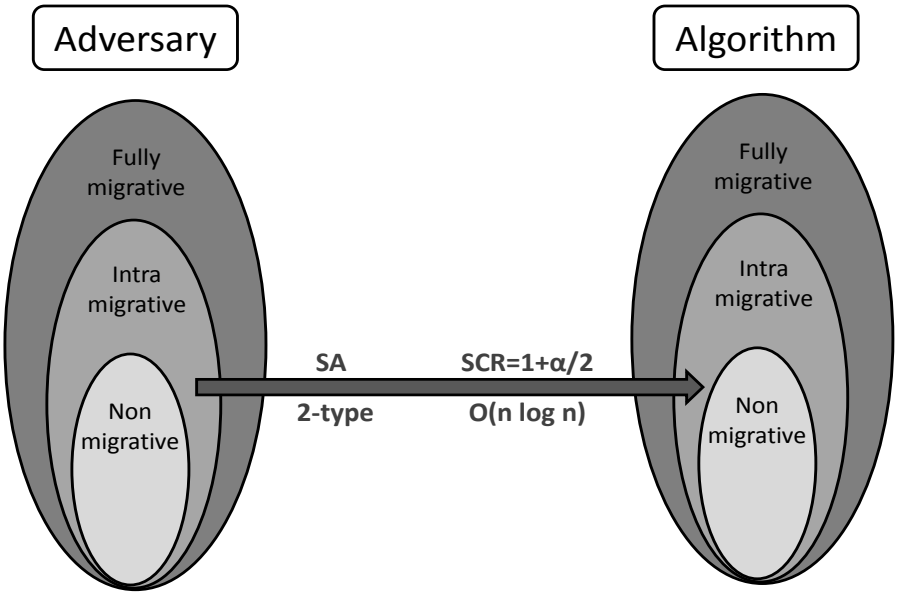


Figure 3.1: A global view of the new algorithm, SA, proposed in this chapter. Here, SCR denotes the "speed competitive ratio", α is a property of the task set — it is the maximum of all the task utilizations that are no greater than one (and hence can take a value in the range $(0,1]$) and n denotes the number of tasks.

Organization of the chapter. The rest of the chapter is organized as follows. Section 3.2 describes the system model. Section 3.3 discusses the hardness of the intra-migrative task assignment problem on two-type heterogeneous multiprocessors. Section 3.4 presents an optimal *intra-migrative* task assignment algorithm, MILP-Algo, that uses Mixed Integer Linear Programming (MILP) formulation. Since solving MILP typically takes a long time (MILP without restrictions is known to be NP-Complete; see pp. 201–202 in [Pap94]), Section 3.5 presents another algorithm, LP-Algo, by relaxing the MILP formulation to Linear Programming (LP) formulation and derives its speed competitive ratio. As solving an LP formulation is also often time consuming, Section 3.6 presents a new intra-migrative algorithm, SA, of time-complexity $O(n \log n)$ that does not rely on solving an LP formulation but has the same speed competitive ratio as LP-Algo, which is proven in Section 3.7. Section 3.8 offers average-case performance evaluations of SA and finally, Section 3.9 concludes.

Tasks	Utilizations of tasks	
	$\mathbf{u_i^1}$	$\mathbf{u_i^2}$
τ_1	0.5	∞
τ_2	1.2	0.8
τ_3	0.7	0.9

Table 3.1: An example to illustrate how to determine the value of α from a given task set.

3.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks on a two-type heterogeneous multiprocessor computing platform $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprising m processors, of which m_1 processors are of type-1 and m_2 processors are of type-2.

On a two-type platform, the worst-case execution time of a task depends on the type of the processor on which the task executes. We denote by C_i^1 and C_i^2 the worst-case execution time of a task τ_i when executed on a processor of type-1 and a processor of type-2, respectively. The minimum inter-arrival time of task τ_i is denoted by T_i. We denote by $u_i^1 \stackrel{\text{def}}{=} C_i^1/T_i$ and $u_i^2 \stackrel{\text{def}}{=} C_i^2/T_i$ the utilizations of the task τ_i on type-1 and type-2 processors, respectively. A task that cannot be executed upon a certain processor type is modeled by setting its worst-case execution time (and thus its utilization) on that processor type to ∞.

Let α be a real number defined as follows:

$$\alpha \stackrel{\text{def}}{=} \max_{\forall \tau_i \in \tau, t \in \{1,2\}} \left\{ u_i^t : u_i^t \leq 1 \right\} \tag{3.1}$$

Then it holds that the utilization of any task on any processor type is either no greater than α or is greater than 1. Formally,

$$\begin{aligned} &\forall \tau_i \in \tau: \quad (u_i^1 \leq \alpha) \ \vee \ (u_i^1 > 1) \quad \text{and} \\ &\forall \tau_i \in \tau: \quad (u_i^2 \leq \alpha) \ \vee \ (u_i^2 > 1) \end{aligned} \tag{3.2}$$

The following example illustrates how to determine the value of α from a given task set.

Example 6. *Consider a task set comprising three tasks, $\tau = \{\tau_1, \tau_2, \tau_3\}$ whose utilizations on type-1 and type-2 processors are as shown in Table 3.1. For this task set, as can be seen from the table, it holds that, $\alpha = 0.9$.*

We assume that all tasks assigned to type-1 (respectively, type-2) processors are scheduled on the set of type-1 (respectively, type-2) processors using an algorithm that is optimal for the problem of scheduling tasks on identical multiprocessors (e.g., ERFair [AS00], DP-Fair [LFS+10], U-EDF [NBN+12]).

For convenience, we sometimes denote a two-type platform π with m_1 processors of type-1 and m_2 processors of type-2 by $\pi(m_1, m_2)$. Also, we denote by $\pi^{(x)}$, a two-type platform in which every processor is $x > 0$ times faster than the corresponding processor in platform π.

3.3 The hardness of the intra-migrative task assignment problem

In this section, we show that the problem of intra-migrative task assignment on two-type heterogeneous multiprocessors is NP-Complete. We denote this problem as HET2-INTRA-ASSIGN and is stated in Figure 3.2. In order to show this, we will first consider a *restricted version* of this

HET2-INTRA-ASSIGN PROBLEM	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. The utilization of a task τ_i on a processor of type-t is given by u_i^t where $i \in \{1,2,\ldots,n\}$ and $t \in \{1,2\}$.
Problem	Find an assignment $f : \{1,2,\ldots,n\} \rightarrow \{1,2\}$ such that, $\forall t \in \{1,2\}$, it holds that: $\left(\sum_{i:f(i)=t} u_i^t \leq m_t\right) \wedge \left(\forall i \in \{1,2,\ldots,n\} \text{ such that } f(i) = t : u_i^t \leq 1\right)$.

Figure 3.2: The intra-migrative task assignment problem on a two-type heterogeneous multiprocessors

problem which is denoted as HET2-INTRA-ASSIGN-SPEC-CASE — see Figure 3.3. We will show that this problem is NP-complete. It then follows that the HET2-INTRA-ASSIGN problem is NP-complete as well.

HET2-INTRA-ASSIGN-SPEC-CASE PROBLEM	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. The utilization of a task τ_i on a processor of type-t is given by u_i^t where $i \in \{1,2,\ldots,n\}$ and $t \in \{1,2\}$. Assume that: $\forall \tau_i \in \tau : u_i^1 = u_i^2$ and $m_1 = 1$ and $m_2 = 1$.
Problem	Find an assignment $f : \{1,2,\ldots,n\} \rightarrow \{1,2\}$ such that, $\forall t \in \{1,2\}$, it holds that: $\left(\sum_{i:f(i)=t} u_i^t \leq m_t\right) \wedge \left(\forall i \in \{1,2,\ldots,n\} \text{ such that } f(i) = t : u_i^t \leq 1\right)$.

Figure 3.3: A restricted version of the intra-migrative task assignment problem on a two-type heterogeneous multiprocessors.

For showing that the HET2-INTRA-ASSIGN-SPEC-CASE problem is NP-Complete, we make use of the PARTITION problem. The PARTITION problem is shown in Figure 3.4 and it is well-known that this problem is NP-Complete (Corollary 15.28 in [KV06], p. 365).

Lemma 1. *The HET2-INTRA-ASSIGN-SPEC-CASE problem is NP-Complete.*

Proof. In order to show that a problem is NP-Complete, we need to: (1) show that the problem is in NP, (2) transform an NP-Complete problem to the problem under consideration and (3) show

PARTITION PROBLEM	
Instance	A list of n natural numbers $c_1, c_2, \ldots, c_n$.
Question	Is there a subset $S \subseteq \{1,2,\ldots,n\}$ such that $\sum_{j \in S} c_j = \sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j$.

Figure 3.4: The partitioning problem, which is known to be NP-Complete [KV06].

that the transformation (of Step (2)) can be done in polynomial time. We now show these for HET2-INTRA-ASSIGN-SPEC-CASE problem.

1. It is straightforward to see that the problem belongs to NP. To show that the problem is in NP, we should be able to verify, in polynomial time, the given *certificate* for an *yes*-instance of the problem. As a *certificate*, we take the assignment on each processor type. To check whether the given assignment in fact satisfies, for all $t \in \{1,2\}$: $\left(\sum_{i:f(i)=t} u_i^t \leq m_t\right) \wedge \left(\forall i \in \{1,2,\ldots,n\} \text{ such that } f(i) = t : u_i^t \leq 1\right)$, is obviously possible in polynomial time; specifically, the time complexity of this step is $O(n)$.

2. We now transform the PARTITION problem (which is NP-Complete) to the above decision problem. Given an instance $c_1, c_2, \ldots, c_n \in \mathbb{N}$ of the PARTITION problem, transform it into an instance of HET2-INTRA-ASSIGN-SPEC-CASE problem with n tasks and compute utilizations of tasks as follows:

$$\forall \tau_i \in \tau, \forall t \in \{1,2\} : \quad u_i^t = \frac{2c_i}{\sum_{k=1}^{n} c_k} \in (0,1] \tag{3.3}$$

   We now show that (intra-migrative) assignment of these n tasks on two processor types is possible *if and only if* there is a set $S \subseteq \{1,2,\ldots,n\}$ such that $\sum_{j \in S} c_j = \sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j$. We do so by first showing, in (a), some results we will use and then showing, in (b), the implication in one direction and finally showing, in (c), the implication in the other direction.

   (a) *Results we will use:*

   (a.1) It is trivial to see that $(a = b) \Rightarrow \left(a = b = \frac{a+b}{2}\right)$. This gives us:

$$\left(\sum_{j \in S} c_j = \sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j\right) \Rightarrow \left(\sum_{j \in S} c_j = \sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j = \frac{\sum_{j \in \{1,2,\ldots,n\}} c_j}{2}\right)$$

   (a.2) It is also trivial to see that $\left(\left(a \leq \frac{a+b}{2}\right) \wedge \left(b \leq \frac{a+b}{2}\right)\right) \Rightarrow (a = b)$. This gives us:

$$\left(\left(\sum_{j \in S} c_j \leq \frac{\sum_{k=1}^{n} c_k}{2}\right) \wedge \left(\sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j \leq \frac{\sum_{k=1}^{n} c_k}{2}\right)\right) \Rightarrow \left(\sum_{j \in S} c_j = \sum_{j \in (\{1,2,\ldots,n\} \setminus S)} c_j\right)$$

(a.3) Let us introduce g that maps an element in $\{1,2,\ldots,n\}$ to a processor type. It is defined as follows:

$$\begin{aligned} i \in S &\Leftrightarrow g(i) = 1 \\ i \in (\{1,2,\ldots,n\} \setminus S) &\Leftrightarrow g(i) = 2 \end{aligned}$$

(b) *Implication in one direction:* We now show (using g) that *if* there is a set $S \subseteq \{1,2,\ldots,n\}$ such that $\sum_{j\in S} c_j = \sum_{j\in(\{1,2,\ldots,n\}\setminus S)} c_j$ *then* intra-migrative assignment of these n tasks on two processor types is possible.

We will do so by assuming that the if-condition of (b) is true and then show that this implies that the then-condition of (b) must also be true. We know that $\sum_{j\in S} c_j = \sum_{j\in(\{1,2,\ldots,n\}\setminus S)} c_j$. Using (a.1) on this gives us:

$$\sum_{j\in S} c_j = \frac{\sum_{j\in\{1,2,\ldots,n\}} c_j}{2}$$

$$\sum_{j\in(\{1,2,\ldots,n\}\setminus S)} c_j = \frac{\sum_{j\in\{1,2,\ldots,n\}} c_j}{2}$$

Multiplying each side by $\frac{2}{\sum_{k=1}^{n} c_k}$ and applying the definition of u_i^t on the left hand side and using the definition of g gives us:

$$\sum_{j\in\{1,2,\ldots,n\} \text{ such that } g(j)=1} u_j^1 = 1$$

$$\sum_{j\in\{1,2,\ldots,n\} \text{ such that } g(j)=1} u_j^2 = 1$$

$$\sum_{j\in\{1,2,\ldots,n\} \text{ such that } g(j)=2} u_j^1 = 1$$

$$\sum_{j\in\{1,2,\ldots,n\} \text{ such that } g(j)=2} u_j^2 = 1$$

It obviously holds that, for a set of non-negative numbers, each element cannot be greater than the sum of all numbers in the set. Using this observation on the above gives us:

$$\begin{aligned} &\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j) = 1 : \; u_j^1 \leq 1 \\ &\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j) = 1 : \; u_j^2 \leq 1 \\ &\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j) = 2 : \; u_j^1 \leq 1 \\ &\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j) = 2 : \; u_j^2 \leq 1 \end{aligned}$$

Hence, we have shown that g is an assignment of tasks to processor types that satisfies the constraints stated in HET2-INTRA-ASSIGN-SPEC-CASE problem.

(c) *Implication in the other direction:* We now show (using g) that *if* intra-migrative assignment of these n tasks on two processor types is possible *then* there is a set $S \subseteq \{1,2,\ldots,n\}$ such that $\sum_{j\in S} c_j = \sum_{j\in(\{1,2,\ldots,n\}\setminus S)} c_j$.

We will do so by assuming that the if-condition of (c) is true and then show that this implies that the then-condition of (c) must also be true. We know that an intra-migrative assignment of these n tasks is possible. Using the function g to express this gives us:

$$\left(\sum_{\forall j\in\{1,2,\ldots,n\} \text{ such that } g(j)=1} u_j^1 \leq 1\right) \wedge$$
$$\left(\sum_{\forall j\in\{1,2,\ldots,n\} \text{ such that } g(j)=2} u_j^2 \leq 1\right) \wedge$$
$$\left(\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j)=1 : u_j^1 \leq 1\right) \wedge$$
$$\left(\forall j \in \{1,2,\ldots,n\} \text{ such that } g(j)=2 : u_j^2 \leq 1\right)$$

Using the definition of u_i^t and the mapping g and multiplying each side by $\frac{\sum_{k=1}^n c_k}{2}$ gives us:

$$\left(\sum_{\forall j\in S} c_j \leq \frac{\sum_{k=1}^n c_k}{2}\right) \wedge$$
$$\left(\sum_{\forall j\in(\{1,2,\ldots,n\}\setminus S)} c_j \leq \frac{\sum_{k=1}^n c_k}{2}\right) \wedge$$
$$\left(\forall j \in S : c_j \leq \frac{\sum_{k=1}^n c_k}{2}\right) \wedge$$
$$\left(\forall j \in (\{1,2,\ldots,n\}\setminus S) : c_j \leq \frac{\sum_{k=1}^n c_k}{2}\right)$$

Observing the first two expressions and using (a.2) gives us:

$$\sum_{j\in S} c_j = \sum_{j\in(\{1,2,\ldots,n\}\setminus S)} c_j$$

This satisfies the constraints of the PARTITION problem.

3. Finally, it can be easily seen that the transformation from PARTITION to HET2-INTRA-ASSIGN-SPEC-CASE using Expression (3.3) is possible in polynomial time; specifically, the time complexity is $O(n)$.

Hence the proof. □

Theorem 1. *The HET2-INTRA-ASSIGN problem is NP-Complete.*

Proof. Follows from Lemma 1 and the fact that HET2-INTRA-ASSIGN-SPEC-CASE problem is a restricted form of HET2-INTRA-ASSIGN problem. □

3.4 MILP-Algo: An optimal intra-migrative task assignment algorithm

In this section, we provide an *optimal intra-migrative task assignment algorithm* for assigning tasks from a task set τ to *processor types* on a two-type platform π. Recall that a task assignment algorithm is said to be *optimal* if, for each task set, it succeeds in finding a feasible assignment, provided such an assignment exists. The proposed algorithm is based on solving Mixed Integer Linear Programming (MILP) formulation. As described earlier, once the tasks have been assigned to processor types, we assume that, an optimal scheduling algorithm (e.g., ERfair [AS00], DP-Fair [LFS+10] or U-EDF [NBN+12]) that is designed for identical multiprocessors, will be used to schedule the tasks on processors of each type. From the feasibility tests of identical multiprocessor scheduling [Hor74], the following necessary and sufficient set of conditions must hold $\forall t \in \{1,2\}$, for intra-migrative task assignment to be feasible:

$$\forall t \in \{1,2\}: \quad \forall \tau_i \in \tau^t : u_i^t \leq \quad 1 \tag{3.4}$$

$$\forall t \in \{1,2\}: \quad \sum_{\tau_i \in \tau^t} u_i^t \leq \quad m_t \tag{3.5}$$

where τ^t denotes the set of tasks that are assigned to processors of type-t. The first condition (Expression (3.4)) is essential since the system model does not allow a task to execute simultaneously on more than one processor at any time (as mentioned earlier in Section 3.2). The second condition (Expression (3.5)) is essential as it is a feasibility condition for implicit-deadline sporadic tasks on identical multiprocessors [Hor74] which ensures that the computing *load* does not exceed the processing *capacity*.

Given these necessary and sufficient feasibility conditions, we now describe, how to obtain an optimal intra-migrative task assignment algorithm. We partition the task set τ into four subsets H12, H1, H2 and L as defined below.

H12 is the set of tasks whose utilization exceeds one on both processor types, i.e., these tasks violate the feasibility condition shown in Expression (3.4), irrespective of the processor type they are assigned to. Formally,

$$\text{H12} \stackrel{\text{def}}{=} \left\{ \tau_i \in \tau : u_i^1 > 1 \ \wedge \ u_i^2 > 1 \right\} \tag{3.6}$$

A task in H12 cannot be scheduled to meet its deadline unless it executes in parallel, which is forbidden in our system model. Hence, for task sets with $\text{H12} \neq \emptyset$, no feasible task assignment exists and thus we assume this set to be empty hereafter.

H1 is the set of tasks that must be assigned to type-1 processors as their utilization on type-2 exceeds one (and hence assigning them to type-2 processors violates the feasibility condition shown in Expression (3.4)), i.e.,

$$\text{H1} \stackrel{\text{def}}{=} \left\{ \tau_i \in \tau : u_i^1 \leq \alpha \ \wedge \ u_i^2 > 1 \right\} \tag{3.7}$$

Minimize Z subject to the following constraints:

I1.	$\forall \tau_i \in \mathrm{L}$: $x_i^1 + x_i^2 = 1$
I2.	$U^1 + \sum_{\tau_i \in \mathrm{L}} x_i^1 \times u_i^1 \leq Z \times m_1$
I3.	$U^2 + \sum_{\tau_i \in \mathrm{L}} x_i^2 \times u_i^2 \leq Z \times m_2$
I4.	$\forall \tau_i \in \mathrm{L}$: $x_i^1 \in \{0,1\}$ and $x_i^2 \in \{0,1\}$; Z is a non-negative real number

Figure 3.5: MILP formulation – MILP-Feas(L, π, U^1, U^2) for assigning tasks in L to processor types in π.

Analogously, H2 is the set of tasks that must be assigned to type-2 processors as their utilization on type-1 exceeds one (and hence assigning them to type-2 processors violates the feasibility condition shown in Expression (3.4)), i.e.,

$$\mathrm{H2} \stackrel{\text{def}}{=} \left\{ \tau_i \in \tau : u_i^1 > 1 \ \wedge \ u_i^2 \leq \alpha \right\} \tag{3.8}$$

Finally, L is the set of tasks that can be assigned to either processor type as their utilizations on both processor types do not exceed one, i.e.,

$$\mathrm{L} \stackrel{\text{def}}{=} \left\{ \tau_i \in \tau : u_i^1 \leq \alpha \ \wedge \ u_i^2 \leq \alpha \right\} \tag{3.9}$$

In these definitions, we can intuitively understand the meaning of “H" as “heavy" and “L" as “light" tasks.

The optimal intra-migrative task assignment algorithm that we propose, namely MILP-Algo, works as follows.

First, assign the tasks in H1 to type-1 (respectively, tasks in H2 to type-2) processors. Let U^1 denote the capacity consumed on type-1 processors after assigning H1 tasks, formally,

$$U^1 = \sum_{\tau_i \in \mathrm{H1}} u_i^1 \tag{3.10}$$

Analogously, let U^2 denote the capacity consumed on type-2 processors after assigning H2 tasks, formally,

$$U^2 = \sum_{\tau_i \in \mathrm{H2}} u_i^2 \tag{3.11}$$

If $U^1 > m_1$ or $U^2 > m_2$ then declare failure as this violates the feasibility condition shown in Expression (3.5).

Second, solve the MILP formulation shown in Figure 3.5 for assigning tasks in L. The formulation in Figure 3.5 is an MILP formulation on x_i^j variables and Z variable.” In this formulation, variable Z denotes the average used capacity of either type-1 or type-2 processors, whichever is greater, and is set as the objective function to be minimized. Each variable x_i^t (where $t \in \{1,2\}$) indicates the assignment of task τ_i to type-t processors. The first set of constraints specifies that

every task must be assigned to a processor type. The second (respectively, third) set of constraints asserts that at most $Z \times m_1$ capacity of type-1 (respectively, $Z \times m_2$ capacity of type-2) processors can be used. The fourth set of constraints asserts that each task must be assigned entirely to either processors of type-1 or type-2. Using the solution of this MILP formulation, assign the tasks in L to processor types as follows: for each $\tau_i \in \mathrm{L}$, τ_i is assigned to type-t processors if and only if $x_i^t = 1$. If $Z > 1$ then declare failure as this indicates that the feasibility condition in Expression (3.5) is violated.

Theorem 2. *The MILP formulation MILP-Feas*$(\mathrm{L}, \pi, \sum_{\tau_i \in \mathrm{H1}} u_i^1, \sum_{\tau_i \in \mathrm{H2}} u_i^2)$ *shown in Figure 3.5 has a solution with* $Z \leq 1$ *if and only if the task set* τ *is intra-migrative feasible on the two-type platform* π.

Proof. Suppose that the task set τ is intra-migrative feasible on platform π and let $\mathscr{X}$ denote a feasible assignment. It then holds that, in this assignment, all the tasks in H1 are assigned to processors of type-1 (otherwise, the condition shown in Expression (3.4) is violated) and analogously, all the tasks in H2 are assigned to processors of type-2. It can be seen that, by setting $U^1 \leftarrow \sum_{\tau_i \in \mathrm{H1}} u_i^1$ and by setting $U^2 \leftarrow \sum_{\tau_i \in \mathrm{H2}} u_i^2$ and $\forall \tau_i \in \mathrm{L}$, by assigning values to x_i^t variables of MILP formulation of Figure 3.5 as:

$$\text{if } \mathscr{X}(i) = 1 \text{ then } x_i^1 \leftarrow 1, x_i^2 \leftarrow 0$$
$$\text{if } \mathscr{X}(i) = 2 \text{ then } x_i^1 \leftarrow 0, x_i^2 \leftarrow 1$$

gives a (feasible) solution to the MILP formulation in which it holds that: $Z \leq 1$.

Now, suppose that there is a feasible solution with $Z \leq 1$ to the MILP formulation, MILP-Feas(L, π, $\sum_{\tau_i \in \mathrm{H1}} u_i^1$, $\sum_{\tau_i \in \mathrm{H2}} u_i^2$), of Figure 3.5. Using this solution, define the assignment of tasks to processor types as follows:

$$\begin{aligned}
\forall i \in \mathrm{H1}: &\quad \mathscr{X}(i) \leftarrow 1 \\
\forall i \in \mathrm{H2}: &\quad \mathscr{X}(i) \leftarrow 2 \\
\forall i \in \mathrm{L}: &\quad \mathscr{X}(i) \leftarrow 1, \quad \text{if } x_i^1 = 1 \wedge x_i^2 = 0 \\
&\quad \mathscr{X}(i) \leftarrow 2, \quad \text{if } x_i^1 = 0 \wedge x_i^2 = 1
\end{aligned}$$

By constraint I1 of the MILP formulation, each task is assigned to exactly one processor type in the assignment $\mathscr{X}$ obtained as shown above. By constraint I2 (respectively, I3) of the MILP formulation, the capacity of type-2 (respectively, type-3) processors is not exceeded in the assignment $\mathscr{X}$ (since $Z \leq 1$ in the feasible solution to MILP formulation). Hence, $\mathscr{X}$ is a feasible intra-migrative task assignment. □

Corollary 1 (MILP-Algo is an optimal intra-migrative algorithm). *If there exists a feasible intra-migrative task assignment of* τ *on* π *then MILP-Algo is guaranteed to return such a feasible intra-migrative task assignment. In other words, MILP-Algo is an optimal intra-migrative task assignment algorithm.*

Minimize Z subject to the following constraints:

C1.	$\forall \tau_i \in \mathrm{L}$: $x_i^1 + x_i^2 = 1$
C2.	$U^1 + \sum_{\tau_i \in \mathrm{L}} x_i^1 \times u_i^1 \leq Z \times m_1$
C3.	$U^2 + \sum_{\tau_i \in \mathrm{L}} x_i^2 \times u_i^2 \leq Z \times m_2$
C4.	$\forall \tau_i \in \mathrm{L}$: x_i^1, x_i^2 are non-negative **real** numbers $\in [0,1]$; Z is a non-negative real number

Figure 3.6: Relaxed LP formulation – LP-Feas($\mathrm{L}, \pi, U^1, U^2$) for assigning tasks in L to processor types in π.

Proof. Follows from Theorem 2. □

Since MILP-Algo relies on solving MILP formulation for which no polynomial time-complexity algorithm is known to exist (when there are no restrictions [Pap94]), we now present a *sub-optimal* polynomial-time algorithm by relaxing the MILP formulation to an LP formulation.

3.5 LP-Algo: An intra-migrative task assignment algorithm

We relax our MILP formulation to LP as shown in Figure 3.6. In this LP formulation, variables Z and x_i^t have the same meaning as the corresponding variables in the MILP formulation and the first three constraints are the same as well. Only the fourth constraint is different (i.e., *relaxed*) and it now asserts that a task can either be *integrally* or *fractionally* assigned to processor types. Since the LP formulation is less constrained than the MILP, the following lemma holds.

Lemma 2. *For any task set* L*, two-type platform* π *and non-negative real numbers* U^1 *and* U^2*, let* Z_{MILP} *be the value of the objective function that any MILP solver would return by solving MILP-Feas*$(\mathrm{L}, \pi, U^1, U^2)$ *shown in Figure 3.5. Similarly, let* Z_{LP} *be the value of the objective function that any LP solver would return by solving LP-Feas(*L*,* π*,* U^1*,* U^2*) shown in Figure 3.6. It then holds that* $Z_{\mathrm{LP}} \leq Z_{\mathrm{MILP}}$.

Our intra-migrative task assignment algorithm, LP-Algo, works as follows.

1. Assign the tasks in H1 to type-1 (respectively, tasks in H2 to type-2) processors. Let U^1 and U^2 denote the same entities as before. If $U^1 > m_1$ or $U^2 > m_2$ then declare failure as it violates the feasibility condition shown in Expression (3.5).

2. Assign the tasks in L by solving the LP formulation shown in Figure 3.6. In the returned solution, if $x_i^t = 1$ (where $t \in \{1,2\}$) then *entirely* (also referred to as *integrally*) assign the corresponding task τ_i to processors of type-t. If $0 < x_i^t < 1$ then assign a *fraction* x_i^t of task τ_i to processors of type-t; we say that such tasks are *fractionally* assigned and are referred to as *fractional* tasks in the rest of the chapter. If $Z > 1$ then declare failure as this indicates that the feasibility condition shown in Expression (3.5) is violated.

Among all the optimal solutions to an LP problem, at least one solution lies at a *vertex* of the *feasible region*[2](see pp. 117 in [LY08]). We are interested in such a solution, as we show below that it leads to a task assignment with at most one fractional task. For ease of discussion, we use index $1,2,\ldots,\ell$ to refer to tasks in subset L hereafter.

Lemma 3. *For any optimal solution* $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ *to the LP formulation shown in Figure 3.6, if S lies at a vertex of the feasible region then there exists at most one task from* L *which is fractionally assigned to both processor types (and the rest are integrally assigned to processors of type-1 and type-2) in the task assignment that S reflects, i.e., there exists at most one index* $f \in \{1,2,\ldots,\ell\}$ *such that* $0 < x_f^1 < 1$ *and* $0 < x_f^2 < 1$.

Proof. The proof is based on Fact 2 in [Bar04c]: *"consider a linear program on n variables* $x_1, x_2, \ldots, x_n$*, in which each variable* x_i *is subject to the non-negativity constraint, i.e.,* $x_i \geq 0$*. Suppose that there are further m linear constraints. If* $m < n$*, then at each vertex of the feasible region (including the basic solution), at most m of the variables have non-zero values"*. Clearly, the LP formulation of Figure 3.6 is a linear program on $n' = 2\ell + 1$ variables (i.e., 2ℓ variables x_i^t, plus variable Z), all subject to non-negativity constraint, and $m' = \ell + 2$ further linear constraints (ℓ constraints due to C1 plus one constraint each due to C2 and C3). As $m' < n'$ (we assume $\ell > 1$; otherwise the problem becomes trivial), we know from the above fact that in every optimal solution at the vertex of the feasible region, it holds that at most $m' = \ell + 2$ variables take non-zero values. Since Z is certain to be non-zero, at most $\ell + 1$ variables x_i^t can be non-zero.

Since there are only ℓ constraints $x_i^1 + x_i^2 = 1$ and at most $\ell + 1$ non-zero variables x_i^t, it can be seen that at most one constraint can have its two variables set to non-zero values. Indeed, for any $f \in \{1,2,\ldots,\ell\}$, if we set the two variables x_f^1 and x_f^2 of the constraint $x_f^1 + x_f^2 = 1$ to fractional values, then there remain $\ell - 1$ non-zero values to distribute to the $\ell - 1$ remaining constraints $x_k^1 + x_k^2 = 1$ ($\forall k \in \{1,2,\ldots,\ell\}$, $k \neq f$). Since none of those constraints can have its two variables set to 0, at least one variable (either x_k^1 or x_k^2) has to take a non-zero value in each of these $(\ell - 1)$ remaining constraints. Again, because $x_k^1 + x_k^2 = 1$ ($\forall k \in \{1,2,\ldots,\ell\}$, $k \neq f$), all these non-zero values have to be equal to 1 and thus, at most one task (in this case, τ_f) can be fractionally assigned. □

Lemma 4. *Any solution,* S_f^{LP}*, to the LP formulation (shown in Figure 3.6) with at most one fractional task and* $Z_f^{\mathrm{LP}} \leq 1$*, can be converted to a solution,* $S_{\mathrm{nf}}^{\mathrm{LP}}$*, with no fractional task and*

$$Z_{\mathrm{nf}}^{\mathrm{LP}} \;\leq\; Z_f^{\mathrm{LP}} + \frac{\alpha}{2} \;\leq\; 1 + \frac{\alpha}{2} \tag{3.12}$$

Proof. Let $S_f^{\mathrm{LP}} = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z_f^{\mathrm{LP}}\}$ be a solution with only one index $f \in \{1,2,\ldots,\ell\}$ such that $0 < x_f^1 < 1$ and $0 < x_f^2 < 1$ (i.e., τ_f is the fractional task). Now, let us convert this solution,

[2]The feasible region of a linear program in n-dimensional space is the region over which all the constraints hold.

S_f^{LP}, into $S_{\text{nf}}^{\text{LP}} = \{x_1^{1'}, x_1^{2'}, x_2^{1'}, x_2^{2'}, \ldots, x_\ell^{1'}, x_\ell^{2'}, Z_{\text{nf}}^{\text{LP}}\}$ such that $\forall i \in \{1,2,\ldots,\ell\}$: $x_i^{1'} = 1 \vee x_i^{2'} = 1$, as follows:

$$\forall i \in \{1,2,\ldots,\ell\}, i \neq f : x_i^{1'} \leftarrow x_i^1 \ \wedge \ x_i^{2'} \leftarrow x_i^2 \tag{3.13}$$

Now, for index f, two options remain:
either perform $x_f^{1'} \leftarrow x_f^1 + x_f^2 \wedge x_f^{2'} \leftarrow 0$ which results in

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \frac{x_f^2 \times u_f^1}{m_1}$$

or perform $x_f^{1'} \leftarrow 0 \wedge x_f^{2'} \leftarrow x_f^1 + x_f^2$ which results in

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \frac{x_f^1 \times u_f^2}{m_2}$$

None of the above two operations violate constraints C1-C4 of the LP formulation. So, let us choose the one that results in the lowest upper bound on $Z_{\text{nf}}^{\text{LP}}$, i.e.,

$$Z_{\text{nf}}^{\text{LP}} \leq \min\left(Z_f^{\text{LP}} + \frac{x_f^2 \times u_f^1}{m_1},\ Z_f^{\text{LP}} + \frac{x_f^1 \times u_f^2}{m_2}\right)$$

Rewriting the above expression, we get:

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \min\left(\frac{x_f^2 \times u_f^1}{m_1},\ \frac{x_f^1 \times u_f^2}{m_2}\right)$$

The min term in the above expression increases as (i) m_1 and m_2 decrease and (ii) u_f^1 and u_f^2 increase. Hence, by setting m_1 and m_2 to their minimum values, i.e., $m_1 = m_2 = 1$, and by setting u_f^1 and u_f^2 to their maximum values, i.e., $u_f^1 = u_f^2 = \alpha$, we get:

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \min\left(\alpha \times x_f^2,\ \alpha \times x_f^1\right)$$

Using the fact $x_f^2 = 1 - x_f^1$ and rewriting yields:

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \alpha \times \min\left(1 - x_f^1,\ x_f^1\right)$$

The maximum values that Z_f^{LP} and the "min" term can take are 1.0 and 0.5, respectively. Hence, the above expression becomes:

$$Z_{\text{nf}}^{\text{LP}} \leq Z_f^{\text{LP}} + \frac{\alpha}{2} \leq 1 + \frac{\alpha}{2}$$

Thus, we showed that this transformed solution $S_{\text{nf}}^{\text{LP}} = \{x_1^{1'}, x_1^{2'}, x_2^{1'}, x_2^{2'}, \ldots, x_\ell^{1'}, x_\ell^{2'}, Z_{\text{nf}}^{\text{LP}}\}$ has no fractional tasks (i.e., indicator variables with fractional values) and satisfies Expression (3.12) and all the constraints of LP formulation. Hence the proof. □

Tasks	Utilizations of tasks	
	$\mathbf{u_i^1}$	$\mathbf{u_i^2}$
τ_1	0.5	0.5
τ_2	1.0	1.0
τ_3	0.5	0.5

Table 3.2: An example to illustrate that the proven speed competitive ratio of LP-Algo algorithm is a tight bound.

Recall that $\pi^{(x)}$ denotes a two-type platform in which each processor is $x > 0$ times faster than the corresponding processor in platform π. We now prove the speed competitive ratio of LP-Algo.

Corollary 2 (Speed competitive ratio of LP-Algo). *If there exists a feasible intra-migrative assignment of τ on π then using LP-Algo, it is guaranteed to find such a feasible intra-migrative assignment of τ on $\pi^{(1+\frac{\alpha}{2})}$.*

Proof. We know that LP-Algo assigns tasks in H1 and H2 in the same way as an optimal intra-migrative task assignment algorithm does (as there is no other way to assign those tasks to meet deadlines). It then uses LP formulation to assign tasks in L. Combining Lemma 1, 2 and 3 gives us: if there exists a feasible intra-migrative task assignment of τ on π then LP-Algo returns an assignment of τ on π in which at most one task from L is fractionally assigned and the rest are integrally assigned to either type-1 or type-2 processors. Then, it follows from Lemma 4 that this fractional task can be assigned integrally to one of the processor types if given a platform in which processors are $1 + \frac{\alpha}{2} \leq 1.5$ (since $0 < \alpha \leq 1$) times faster. Hence the proof. □

We now show that the proven speed competitive ratio $1 + \frac{\alpha}{2} \leq 1.5$ of LP-Algo is a tight bound.

Theorem 3 (Speed competitive ratio of LP-Algo is tight). *The proven speed competitive ratio* 1.5 *of algorithm LP-Algo is a tight bound.*

Proof. In order to show that the proven speed competitive ratio of LP-Algo algorithm is a tight bound, it is sufficient to show that there exists a (feasible intra-migrative) problem instance for which LP-Algo needs 1.5 times faster processors to output a feasible intra-migrative assignment. We now show that such a problem instance exists.

Consider a problem instance with a task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ comprising three tasks and a two-type platform $\pi = \{\pi_1, \pi_2\}$ comprising two processors. Let π_1 be a processor of type-1 and π_2 be a processor of type-2. The utilizations of tasks are shown in Table 3.2.

Observe that the given task set τ is intra-migrative feasible on the given platform π. A feasible intra-migrative assignment is obtained by assigning (i) τ_1 and τ_3 to type-1 processors (which has a single processor, π_1) and (ii) τ_2 to type-2 processors (which has a single processor, π_2). This assignment is shown in Table 3.3.

Now consider algorithm LP-Algo. Initially, the task set is partitioned using Expressions (3.6)–(3.9) as follows: $H12 = \emptyset$, $H1 = \emptyset$, $H2 = \emptyset$ and $L = \{\tau_1, \tau_2, \tau_3\}$. Since there are no heavy tasks,

Processor types	Tasks assigned
type-1 (π_1)	τ_1 and τ_3
type-2 (π_2)	τ_2

Table 3.3: A feasible intra-migrative assignment for tasks shown in Table 3.2 on a two-type platform π having one processor of type-1 and one processor of type-2.

LP-Algo solves LP formulation shown in Figure 3.6 for assigning light tasks. Upon solving the LP formulation, we obtain a solution shown in Table 3.4. Upon assigning tasks to processor types using the solution output by the solver (which is shown in Table 3.4), it holds that:

- type-1 processors are fully utilized
- type-2 processors are fully utilized and
- task τ_2 is equally split between type-1 and type-2 processors

It can be seen that, in order to assign τ_2 integrally to type-1 processors, the speed of type-1 processors *must be* increased to 1.5. Analogously, for assigning τ_2 integrally to type-2 processors, the speed of type-2 processors *must be* increased to 1.5 as well. Therefore, a speedup of 1.5 is required to assign τ_2 integrally to one of the processor types.

Hence, the proven speed competitive ratio 1.5 of LP-Algo algorithm is a tight bound. □

Remark 1 Although Corollary 2 states that, for an intra-migrative feasible task set, LP-Algo needs a platform in which *every processor* is $1+\frac{\alpha}{2}$ times faster, in order to output an intra-migrative feasible task assignment, it is trivial to see from the proof of Corollary 2 that a platform in which only *one processor* is $1+\frac{\alpha}{2}$ times faster is sufficient (to which the *at most one* fractional task can be integrally assigned).

Recall that $\pi(m_1,m_2)$ denotes a two-type platform in which $m_1 > 0$ processors are of type-1 and $m_2 > 0$ processors are of type-2. We now state the performance of LP-Algo in terms of additional number of processors.

Corollary 3. *If there exists a feasible intra-migrative assignment of τ on $\pi(m_1,m_2)$ then, using LP-Algo, it is guaranteed to obtain such a feasible intra-migrative assignment of τ on $\pi'(m_1+1,m_2)$, which has one additional processor of type-1 compared to π.*

Proof. Combining Lemma 1, 2 and 3 gives us: if there exists a feasible intra-migrative task assignment of τ on π then LP-Algo returns an assignment of τ on π in which at most one task from L, say τ_f, is fractionally assigned to both processor types and the rest are integrally assigned to either type-1 or type-2 processors. From the definition of L, we know that $u_f^1 \leq \alpha$ and $u_f^2 \leq \alpha$ where $0 < \alpha \leq 1$. Hence, if such a task τ_f exists then it could be integrally assigned to the set of type-1 processors, which has an additional processor in π'. Hence the proof. □

Variables	Values
Z	1.0
x_1^1	1.0
x_1^2	0.0
x_2^1	0.5
x_2^2	0.5
x_3^1	0.0
x_3^2	1.0

Table 3.4: A solution output by the LP solver to the LP formulation shown in Figure 3.6 for the problem instance under consideration.

Remark 2 It is trivial to see that Corollary 3 holds true if LP-Algo is given a platform $\pi'(m_1, m_2 + 1)$ comprising $m+1$ processors of which m_1 processors are of type-1 and m_2+1 processors are of type-2.

It is trivial to see that the assignment techniques that rely on solving LP formulations take considerable amount of time to output a solution compared to techniques that do not solve LP formulations and rely on simpler techniques. So, we now propose an algorithm, namely SA, that has the same speed competitive ratio as LP-Algo but does not solve LP formulation and instead uses a simple and elegant assignment technique.

3.6 SA: An intra-migrative task assignment algorithm

In this section, we describe the working of algorithm, SA, and show that it has a time-complexity of $O(n \log n)$.

3.6.1 The description of SA algorithm

SA is an intra-migrative task assignment algorithm and works as follows.

1. Partition the task set τ into subsets H12, H1, H2 and L as shown in Expression (3.6) to Expression (3.9). If H12 $\neq \emptyset$ then declare failure.

2. Assign tasks in H1 to type-1 (respectively, H2 to type-2) processors on platform π. If $U^1 = \sum_{\tau_i \in \text{H1}} u_i^1 > m_1$ or $U^2 = \sum_{\tau_i \in \text{H2}} u_i^2 > m_2$ then declare failure.

3. Sort the tasks in L in non-increasing order of $\frac{u_i^2}{u_i^1}$ — intuitively, in non-increasing order of their preference to be assigned to type-1 processors.

4. Traverse this sorted list from “left to right" and assign the tasks one after the other to type-1 processors until there is no capacity left on type-1 processors to assign a task integrally (or all the tasks in L are assigned to type-1 processors leading to a successful assignment).

5. Traverse the sorted list from "right to left" and assign the remaining tasks one after the other to type-2 processors until there is no capacity left on type-2 processors to assign a task integrally (or the task that could not be assigned in the previous step is assigned to type-2 processors thereby resulting in a successful assignment).

6. Finally, assign the remaining task, if any, fractionally to both processor types (we show in Theorem 4 that there can be at most *one* such task, if there exists a feasible intra-migrative assignment of τ on π). While assigning this remaining task, assign as big a fraction of the task as possible to type-1 processors (i.e., the entire remaining capacity of type-1 processors is used), and assign the remaining fraction to type-2 processors. If there is not enough capacity left to assign this remaining task fractionally then declare failure.

SA is named so because we "**S**ort and **A**ssign" the tasks in L.

3.6.2 Time-complexity of SA algorithm

We now show that the time-complexity of SA is a low-degree polynomial function of the number of tasks (n). By inspecting the six steps of algorithm, SA, described above, we know that:

- H1 tasks are assigned to type-1 processors (i.e., at most n tasks). The time-complexity of this operation is $O(n)$.
- H2 tasks are assigned to type-2 processors (i.e., at most n tasks). The time-complexity of this operation is $O(n)$.
- Sorting is performed over a subset of τ (i.e., at most n tasks). The time-complexity of this operation is $O(n \cdot \log n)$ e.g., using Heapsort.
- Traverse the sorted list L (i.e., at most n tasks) and assign the tasks to processor types. The time-complexity of this operation is $O(n)$.

Thus, the time-complexity of the algorithm is at most

$$\underbrace{O(n)}_{\text{assign H1 tasks}} + \underbrace{O(n)}_{\text{assign H2 tasks}} + \underbrace{O(n \cdot \log n)}_{\text{sort L tasks}} + \underbrace{O(n)}_{\text{assign L tasks}} = O(n \cdot \log n)$$

3.7 Speed competitive ratio of SA algorithm

In this section, we derive the speed competitive ratio of SA. For this, we mainly focus on the assignment of tasks in L as SA assigns tasks in H1 and H2 in the same way as an optimal intra-migrative assignment algorithm does.

First, we introduce a term, *swap solution*, that is extensively used in the rest of this section.

Definition 18 (*Swap solution*)**.** *A solution $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ to the LP formulation of Figure 3.6 is said to be a swap solution if and only if $\forall \tau_i, \tau_j \in \mathrm{L}$ such that $\tau_i \neq \tau_j$ and $\frac{u_i^2}{u_i^1} \geq \frac{u_j^2}{u_j^1}$, it holds that $x_i^1 = 1 \vee x_j^2 = 1$.*

Property 1 (A single fractional task)**.** *From Definition 18, it can be easily shown that, in any swap solution* $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$*, there exists at most one task which is fractionally assigned to both processor types, i.e., there exists at most one index* $f \in \{1, 2, \ldots, \ell\}$ *such that* $0 < x_f^1 < 1$ *and* $0 < x_f^2 < 1$.

The remainder of this section is organized as follows. In subsection 3.7.1, we describe a method to transform any feasible solution of the LP formulation (shown in Figure 3.6) into a feasible swap solution (Lemma 5). Then, in subsection 3.7.2, we show that the solution returned by SA for assigning tasks in L is similar to the *swap solution*, in the sense that, at most one task is fractionally assigned to both processor types and the rest are integrally assigned to type-1 and type-2 processors (Theorem 4). Finally, we show that this fractional task can be integrally assigned to a processor type if given a platform in which processors are $1 + \frac{\alpha}{2} \leq 1.5$ times faster (Theorem 5). Using all this information and considering that SA assigns tasks in H1 and H2 in a same way as an optimal intra-migrative task assignment algorithm does, we establish that its speed competitive ratio is $1 + \frac{\alpha}{2} \leq 1.5$.

3.7.1 The swapping method

We now show that any feasible solution to our LP formulation can be transformed into a feasible swap solution.

Lemma 5. *Any feasible solution* $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ *to the LP formulation of Figure 3.6 can be transformed into a feasible swap solution* $S' = \{x_1^{1'}, x_1^{2'}, x_2^{1'}, x_2^{2'}, \ldots, x_\ell^{1'}, x_\ell^{2'}, Z'\}$ *for which* $Z' = Z$.

Proof. If S is not a swap solution, then we know by definition that there exists $\tau_p, \tau_q \in \mathrm{L}$ such that:

$$\tau_p \neq \tau_q \text{ and } \frac{u_p^2}{u_p^1} \geq \frac{u_q^2}{u_q^1} \text{ and } x_p^1 < 1 \ \wedge \ x_q^2 < 1 \tag{3.14}$$

We prove the claim by (iteratively) transforming this solution S into another solution S' in which the following properties hold:

P1. $\forall \tau_i \in \mathrm{L},\ \tau_i \neq \tau_p, \tau_i \neq \tau_q$: $x_i^{1'} = x_i^1$ and $x_i^{2'} = x_i^2$

P2. $x_p^{1'} = 1 \vee x_q^{2'} = 1$

P3. Constraints C1-C4 of LP formulation hold and $Z' = Z$

The steps involved in transforming solution S into S' are described below. Performing those steps iteratively as long as such a pair $\tau_p, \tau_q \in \mathrm{L}$ fulfilling Expression (3.14) exists, will ultimately lead to a feasible swap solution S' with Z' equal to Z. Property P1 and P2 ensure that, with each iteration, the solution is moving closer towards the swap solution and P3 ensures that this (intermediate) solution is feasible. At each iteration, we denote by $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ the feasible solution computed in the previous iteration (in the first iteration, this solution is

the given one) and by $S' = \{x_1^{1'}, x_1^{2'}, x_2^{1'}, x_2^{2'}, \ldots, x_\ell^{1'}, x_\ell^{2'}, Z'\}$ the modified feasible solution after the current iteration (note that S' of iteration k acts as S in iteration $k+1$). The solution obtained after the final iteration is the feasible swap solution. Each iteration is performed as follows:

$\forall \tau_i \in \mathrm{L}, \tau_i \neq \tau_p, \tau_i \neq \tau_q$:

$$x_i^{1'} \leftarrow x_i^1 \tag{3.15}$$
$$x_i^{2'} \leftarrow x_i^2 \tag{3.16}$$

and

$$x_p^{1'} \leftarrow x_p^1 + \delta_1 \tag{3.17}$$
$$x_p^{2'} \leftarrow x_p^2 - \delta_1 \tag{3.18}$$
$$x_q^{1'} \leftarrow x_q^1 - \delta_2 \tag{3.19}$$
$$x_q^{2'} \leftarrow x_q^2 + \delta_2 \tag{3.20}$$

where $\delta_1 \stackrel{\text{def}}{=} \min(x_p^2, x_q^1 \times \frac{u_q^1}{u_p^1})$ and $\delta_2 \stackrel{\text{def}}{=} \min(x_p^2 \times \frac{u_p^1}{u_q^1}, x_q^1)$.

Proof of P1. From Expressions (3.15) and (3.16), it is trivial to see that Property **P1** holds.

Proof of P2. We have to consider two cases:
Case (i): $x_p^2 \leq x_q^1 \times \frac{u_q^1}{u_p^1}$. In this case, $\delta_1 = x_p^2$ and $\delta_2 = x_p^2 \times \frac{u_p^1}{u_q^1}$. Substituting the value of δ_1 in Expression (3.17) gives: $x_p^{1'} \leftarrow x_p^1 + x_p^2$. Since we know that $x_p^1 + x_p^2 = 1$ (it is true in the initial solution S and it holds true in all the subsequent iterations as well, as shown later in **Proof of P3**), we get $x_p^{1'} \leftarrow 1$ and hence Property **P2** is satisfied.
Case (ii): $x_p^2 > x_q^1 \times \frac{u_q^1}{u_p^1}$. This case is analogous to the previous case. In this case, $\delta_1 = x_q^1 \times \frac{u_q^1}{u_p^1}$ and $\delta_2 = x_q^1$. Substituting the value of δ_2 in Expression (3.20) gives: $x_q^{2'} \leftarrow x_q^1 + x_q^2$. Since we know that $x_q^1 + x_q^2 = 1$ (it is true in the initial solution S and it holds true in all the subsequent iterations as well, as shown later in **Proof of P3**), we get $x_q^{2'} \leftarrow 1$ and hence Property **P2** is satisfied.

Proof of P3. Since the initial solution S is feasible, constraint C1 holds by definition, i.e., $\forall \tau_i \in \mathrm{L} : x_i^1 + x_i^2 = 1$. Let us see whether this holds in solution S' which is obtained from S with the help of Expressions (3.15)-(3.20). Let us consider the following two cases:
Case (i): $\forall \tau_i \in \mathrm{L}, \tau_i \neq \tau_p, \tau_i \neq \tau_q$. Adding Expressions (3.15) and (3.16), we get: $x_i^{1'} + x_i^{2'} = x_i^1 + x_i^2$. Since we know that $\forall \tau_i \in \mathrm{L} : x_i^1 + x_i^2 = 1$, we obtain: $x_i^{1'} + x_i^{2'} = 1$. Recall that, in the next iteration, this solution S' acts as S while computing another S'. Hence, this holds in that iteration and all subsequent iterations. Hence constraint C1 holds true.
Case (ii): $\tau_i = \tau_p \vee \tau_i = \tau_q$. Analogous to the previous case, adding Expressions (3.17) and (3.18), gives: $x_p^{1'} + x_p^{2'} = 1$ and adding Expressions (3.19) and (3.20), gives: $x_q^{1'} + x_q^{2'} = 1$. This holds true in all the iterations for the reasons stated in the previous case. Hence, $\forall \tau_i \in L$, constraint C1 holds true.

Now, we show that constraint C2 holds. From Equations (3.15)–(3.20), we have:

$$\begin{aligned}\sum_{i=1}^{\ell}(x_i^{1'} \times u_i^1) &= \sum_{\substack{i=1 \\ i\neq p, i\neq q}}^{\ell}(x_i^1 \times u_i^1) \\ &+ \left(x_p^1 + \min\left(x_p^2, x_q^1 \times \frac{u_q^1}{u_p^1}\right)\right) \times u_p^1 \\ &+ \left(x_q^1 - \min\left(x_p^2 \times \frac{u_p^1}{u_q^1}, x_q^1\right)\right) \times u_q^1 \end{aligned} \tag{3.21}$$

In both cases (i.e., $x_p^2 \leq x_q^1 \times \frac{u_q^1}{u_p^1}$ and $x_p^2 > x_q^1 \times \frac{u_q^1}{u_p^1}$), the 'min' terms in Expression (3.21) cancel out and hence the expression simplifies to:

$$\sum_{i=1}^{\ell}(x_i^{1'} \times u_i^1) = \sum_{i=1}^{\ell}(x_i^1 \times u_i^1) \leq Z \times m_1 \tag{3.22}$$

Hence, Constraint C2 is not violated. With analogous reasoning, it can be shown that

$$\sum_{i=1}^{\ell}(x_i^{2'} \times u_i^2) = \sum_{i=1}^{\ell}(x_i^2 \times u_i^2) \leq Z \times m_2 \tag{3.23}$$

Hence, Constraint C3 is also not violated.

Now let us consider constraint C4. We know by definition that in solution S, $\forall \tau_i \in \mathrm{L}$, it holds that $x_i^1 \geq 0$ and $x_i^2 \geq 0$. Hence, from Expressions (3.15) and (3.16), in solution S', $\forall \tau_i \in \mathrm{L}, \tau_i \neq \tau_p, \tau_i \neq \tau_q$, it holds that $x_i^{1'} \geq 0$ and $x_i^{2'} \geq 0$. Now, for $\tau_i = \tau_p \vee \tau_i = \tau_q$, we have two cases:
Case (i): $x_p^2 \leq x_q^1 \times \frac{u_q^1}{u_p^1}$. In this case, we have $\delta_1 = x_p^2$ and $\delta_2 = x_p^2 \times \frac{u_p^1}{u_q^1}$. Since we have shown that constraint C1 holds, substituting the value of δ_1 in Expression (3.17) and (3.18), we get $x_p^{1'} = 1$ and $x_p^{2'} = 0$, respectively. From the case, we have: $x_q^1 \geq x_p^2 \times \frac{u_p^1}{u_q^1} > 0$. So, substituting the value of δ_2 in Expression (3.19) and (3.20) gives us $x_q^{1'} \geq 0$ and $x_q^{2'} > 0$, respectively. Hence, constraint C4 holds in this case.
Case (ii): $x_p^2 > x_q^1 \times \frac{u_q^1}{u_p^1}$. This case is analogous to the previous case. In this case, we have $\delta_1 = x_q^1 \times \frac{u_q^1}{u_p^1}$ and $\delta_2 = x_q^1$. Since we have shown that constraint C1 holds, substituting the value of δ_2 in Expression (3.19) and (3.20), we get $x_q^{1'} = 0$ and $x_q^{2'} = 1$, respectively. From the case, we have: $x_p^2 \geq x_q^1 \times \frac{u_q^1}{u_p^1} > 0$. So, substituting the value of δ_1 in Expression (3.17) and (3.18) gives us $x_p^{1'} > 0$ and $x_p^{2'} \geq 0$, respectively. Hence, constraint C4 holds in this case. Thus, $\forall \tau_i \in L$, constraint C4 holds true.

Since none of the constraints, C1-C4, of LP formulation are violated, the transformed solution remains feasible, and from Expression (3.22) and Expression (3.23), we can conclude that $Z' = Z$. Thus, at the end of the iteration, for a pair of tasks τ_p, τ_q that are considered in the iteration, it holds that: $x_p^1 = 1 \vee x_q^2 = 1$. Hence, applying the transformation shown in Expressions (3.15)–(3.20) repeatedly, we obtain a feasible swap solution. □

Lemma 6. *For any feasible swap solution* $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ *to the LP formulation, we can re-index tasks in* L *such that* $\frac{u_1^2}{u_1^1} \geq \frac{u_2^2}{u_2^1} \geq \cdots \geq \frac{u_\ell^2}{u_\ell^1}$ *(with ties broken favoring the task with lower index before re-indexing) and with this order, there is an index* $f \in \{0, 1, 2, \ldots, \ell, \ell+1\}$ *such that:*

$$\forall i < f: \quad x_i^1 = 1 \quad \textit{and}$$
$$\forall i > f: \quad x_i^2 = 1$$

Proof. Let $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ be any feasible swap solution. We re-index the tasks (together with x_i^1 and x_i^2 values in S, $\forall \tau_i \in$ L) such that:

$$\frac{u_1^2}{u_1^1} \geq \frac{u_2^2}{u_2^1} \geq \cdots \geq \frac{u_\ell^2}{u_\ell^1} \tag{3.24}$$

with ties broken as described in the claim. We now prove that there exists $f \in \{0, 1, 2, \ldots, \ell, \ell+1\}$ such that $\forall \tau_i \in$ L, if $i < f$ then $x_i^1 = 1$ and if $i > f$ then $x_i^2 = 1$. The following three cases may arise (recall from Property 1 that, in a swap solution, there is at most one fractional task): (1) all the tasks in L are assigned to the same processor type or (2) tasks in L are assigned to both processor types and there is *one* fractional task or (3) tasks in L are assigned to both processor types and there is *no* fractional task. We now consider each of these cases separately below.

Case (1): All the tasks in L are assigned to processors of type-1 (respectively, type-2); The claim trivially holds for $f = \ell + 1$ (respectively, $f = 0$).

Case (2): The tasks in L are assigned to both processor types and there is *one* fractional task; let f be the index of this fractional task, i.e., there exists $\tau_f \in$ L for which $0 < x_f^1 < 1$ and $0 < x_f^2 < 1$. We need to consider two sub-cases:

Case 2.1 ($\forall \tau_i \in$ L **such that** $i < f$)**:** Since $\frac{u_i^2}{u_i^1} \geq \frac{u_f^2}{u_f^1}$, we know from Definition 18 that $x_i^1 = 1 \vee x_f^2 = 1$. However, by definition of f we know that τ_f is fractionally assigned and thus, $0 < x_f^2 < 1$; so, it must hold that $x_i^1 = 1$. Consequently, all the tasks $\tau_i \in$ L with $i < f$ are integrally assigned to type-1 processors.

Case 2.2 ($\forall \tau_i \in$ L **such that** $i > f$)**:** Since $\frac{u_f^2}{u_f^1} \geq \frac{u_i^2}{u_i^1}$, we know from Definition 18 that $x_f^1 = 1 \vee x_i^2 = 1$. Following the same reasoning as above, we have $0 < x_f^1 < 1$ and thus, it must hold that $x_i^2 = 1$. Hence, all tasks $\tau_i \in$ L with $i > f$ are integrally assigned to type-2 processors.

Case (3): The tasks in L are assigned to both processor types and there is *no* fractional task. In this case, let f be the index of the first task in the sorted order (of tasks in L as shown in Expression (3.24)) that is integrally assigned to type-2 processors. By definition of τ_f, we know that all the tasks $\tau_i \in$ L with $i < f$ must be integrally assigned to type-1 processors. Now consider any task $\tau_i \in$ L with $i > f$. Since $\frac{u_f^2}{u_f^1} \geq \frac{u_i^2}{u_i^1}$, we know from Definition 18 that $x_f^1 = 1 \vee x_i^2 = 1$. But, we know that $x_f^1 = 0$, so it must hold that $x_i^2 = 1$. Hence, all tasks $\tau_i \in$ L with $i > f$ are integrally assigned to type-2 processors.

We showed that the claim holds for all the cases, i.e., there exists an index $f \in \{0,1,2,\ldots,\ell,\ell+1\}$ such that all the tasks in L (sorted as shown in Expression (3.24)) to its *left* are assigned to type-1 processors and all the tasks in L to its *right* are assigned to type-2 processors. Hence the proof. □

3.7.2 The speed competitive ratio of SA algorithm

In this section, we show that the speed competitive ratio of SA algorithm is $1+\frac{\alpha}{2} \leq 1.5$. Before that, we prove a property of SA which in turn helps us to prove its speed competitive ratio.

Theorem 4. *If there exists an intra-migrative feasible assignment of task set τ on two-type platform π then* SA *succeeds in finding a feasible assignment of τ on π in which at most one task from* L *is fractionally assigned to both processor types and the rest are integrally assigned to type-1 and type-2 processors.*

Proof. We know from Lemma 1 that if task set τ is intra-migrative feasible on two-type platform π then MILP-Algo succeeds in finding such an intra-migrative feasible assignment of τ on π as well. This implies that there exists a feasible solution to the MILP formulation of Figure 3.5 with $Z_{\text{MILP}} \leq 1$. Then, we know from Lemma 2 that, since there exists a solution to the MILP formulation with $Z_{\text{MILP}} \leq 1$, there also exists a feasible solution to the LP formulation of Figure 3.6 with $Z_{\text{LP}} \leq 1$. We also know from Lemma 5 that such a solution can be converted into a feasible swap solution in which at most one task from L is fractionally assigned. Finally, we know from Lemma 6 that in this feasible swap solution, tasks in L can be re-indexed such that $\frac{u_1^2}{u_1^1} \geq \frac{u_2^2}{u_2^1} \geq \cdots \geq \frac{u_\ell^2}{u_\ell^1}$ (with ties broken, during re-indexing favoring the task with lower index before re-indexing) and with this order, there is an index $f \in \{0,1,\ldots,\ell,\ell+1\}$ such that:

$$\begin{aligned} &\forall i < f: \quad x_i^1 = 1 \quad \text{and} \\ &\forall i > f: \quad x_i^2 = 1 \end{aligned}$$

For the sake of readability, henceforth we simply denote by $S = \{x_1^1, x_1^2, x_2^1, x_2^2, \ldots, x_\ell^1, x_\ell^2, Z\}$ this sorted feasible swap solution (in which tasks are sorted as mentioned above). With this background, we now prove the theorem. The intuition behind the proof is that SA always succeeds in returning a solution similar to the sorted feasible swap solution S (from the reasoning above, we already know that such a swap solution always exists if τ is intra-migrative feasible on π).

We prove the theorem by contradiction. Let us assume that the task set τ is intra-migrative feasible on π but SA fails to find an assignment of τ on π in which at most one task from L is fractionally assigned. We consider all the scenarios and show that it is impossible for this to happen.

Let us study the behavior of SA. It assigns tasks in H1 and H2 in the same manner as an optimal intra-migrative task assignment algorithm does (see the algorithm, MILP-Algo, in Section 3.4). Hence, we only need to look at the assignment of tasks in L. It considers these tasks in the order:

$$\frac{u_1^2}{u_1^1} \geq \frac{u_2^2}{u_2^1} \geq \cdots \geq \frac{u_\ell^2}{u_\ell^1} \tag{3.25}$$

with ties broken favoring the task with lower index before re-indexing. It considers tasks one by one from the left-hand side in the sorted order (as shown in Expression (3.25)) and starts assigning them to type-1 processors. It stops assigning tasks to type-1 processors upon failing to assign a task say, τ_x, integrally on type-1 processors or all the tasks are successfully assigned thereby resulting in a successful assignment — whichever happens first. If it stops at τ_x then it considers tasks one by one from the right-hand side in the sorted order and starts assigning them to type-2 processors. It stops assigning tasks to processors of type-2 as soon as it fails to assign a task integrally (if τ is intra-migrative feasible on π then this task can be none other than τ_x as shown later in the theorem) or it successfully assigns τ_x integrally to a type-2 processor thereby resulting in a successful assignment — whichever happens first. If it stopped because it could not assign τ_x integrally to type-2 processor then it fractionally assigns τ_x to type-1 and type-2 processors.

We now compare the output of SA with that of the sorted feasible swap solution S and show that it is impossible for SA to fail (i.e., not to return an assignment with at most one fractional task) when τ is intra-migrative feasible on π. Note that the tasks are indexed in the same manner in both SA and S, i.e., $\frac{u_1^2}{u_1^1} \geq \frac{u_2^2}{u_2^1} \geq \cdots \geq \frac{u_\ell^2}{u_\ell^1}$, with ties broken in the same way.

We need to consider two cases with respect to the existence of a fractional task in S, i.e., a task τ_f for which $0 < x_f^1 < 1$ and $0 < x_f^2 < 1$. The remainder of the proof consists in exploring all the possible scenarios (and showing that each case leads to contradiction): it is first split into two parts, corresponding to the two cases 'such a fractional task exists or not', and each part is further divided into three cases.

Part 1: There exists a task $\tau_f \in \mathrm{L}$ in the swap solution S which is fractionally assigned to both processor types, i.e., $0 < x_f^1 < 1$ and $0 < x_f^2 < 1$. In this part, we need to consider three cases with respect to the position of x and f.

Case 1.1 ($x < f$)**:** We know that tasks $\{\tau_1, \tau_2, \ldots, \tau_{f-1}\} \in \mathrm{L}$ have been integrally assigned to type-1 processors in solution S, i.e., $\forall i \in \{1, 2, \ldots, f-1\}$: $x_i^1 = 1 \ \wedge \ x_i^2 = 0$. This means that $U^1 + \sum_{i=1}^{f-1} u_i^1 \leq m_1$ where $U^1 = \sum_{\tau_i \in \mathrm{H1}} u_i^1$ and since $x < f$, it must hold that:

$$U^1 + \sum_{i=1}^{x} u_i^1 \leq m_1 \tag{3.26}$$

i.e., tasks $\{\tau_1, \tau_2, \ldots, \tau_x\} \in \mathrm{L}$ have been integrally assigned to processors of type-1 in S. However, we know that SA failed to integrally assign those tasks $\{\tau_1, \tau_2, \ldots, \tau_x\}$ to type-1 processors, which means that $U^1 + \sum_{i=1}^{x} u_i^1 > m_1$, in contradiction with Expression (3.26).

Case 1.2 ($x > f$)**:** This case is symmetrical to Case 1.1 and also leads to contradiction. We know that tasks $\{\tau_{f+1}, \tau_{f+2}, \ldots, \tau_\ell\} \in \mathrm{L}$ have been integrally assigned to type-2 processors in solution S, i.e., $\forall i \in \{f+1, f+2, \ldots, \ell\}$: $x_i^1 = 0 \ \wedge \ x_i^2 = 1$. This means that $U^2 + \sum_{i=f+1}^{\ell} u_i^2 \leq m_2$ where $U^2 = \sum_{\tau_i \in \mathrm{H2}} u_i^2$ and since $x > f$, it must hold that:

$$U^2 + \sum_{i=x}^{\ell} u_i^2 \leq m_2 \tag{3.27}$$

i.e., tasks $\{\tau_x, \tau_{x+1}, \ldots, \tau_\ell\} \in$ L have been integrally assigned to processors of type-2 in S. However, we know that SA failed to integrally assign those tasks $\{\tau_x, \tau_{x+1}, \ldots, \tau_\ell\}$ to type-2 processors, which means that $U^2 + \sum_{i=x}^{\ell} u_i^2 > m_2$, in contradiction with Expression (3.27).

Case 1.3 ($x = f$)**:** This indicates that the two sets of tasks, i.e., $\{\tau_1, \tau_2, \ldots, \tau_{x-1}\} \in$ L and $\{\tau_{x+1}, \tau_{x+2}, \ldots, \tau_\ell\} \in$ L, are integrally assigned to type-1 and type-2 processors, respectively, in both S and the solution returned by SA. Let $x_f^{1,S}$ denote the fraction of $\tau_f \in$ L assigned to type-1 processors in S, and similarly let $x_x^{1,\text{SA}}$ denote the fraction of $\tau_x \in$ L assigned to type-1 processors in the solution returned by SA. Since S is feasible we know that $U^1 + \sum_{i=1}^{f-1} u_i^1 + x_f^{1,S} \times u_f^1 \leq m_1$, and since $f = x$ we have:

$$U^1 + \sum_{i=1}^{x-1} u_i^1 + x_f^{1,S} \times u_x^1 \leq m_1 \tag{3.28}$$

But, by design (see step 6 of SA algorithm in Section 3.6), we also know that τ_x is split under SA such that:

$$U^1 + \sum_{i=1}^{x-1} u_i^1 + x_x^{1,\text{SA}} \times u_x^1 = m_1 \tag{3.29}$$

From Expression (3.28) and (3.29), we then observe that $x_f^{1,S} \leq x_x^{1,\text{SA}}$. As a first conclusion, SA is thus able to integrally assign to type-1 processors all the tasks in τ that are integrally assigned to type-1 processors in solution S, plus (at least) the same fraction of task τ_x as that of task τ_f assigned to type-1 processor in S. Also, $x_f^{1,S} \leq x_x^{1,\text{SA}}$ implies that $x_f^{2,S} \geq x_x^{2,\text{SA}}$, which in turn yields:

$$U^2 + \sum_{i=f+1}^{n} u_i^2 + x_f^{2,S} \times u_f^2 \;\geq\; U^2 + \sum_{i=x+1}^{n} u_i^2 + x_x^{2,\text{SA}} \times u_x^2$$

The left-hand (respectively, right-hand) side of the above expression denotes the utilization of the tasks, including the fractional assignment of τ_f (which is same task as τ_x), assigned to type-2 processors in the solution S (respectively, SA). As a second conclusion, SA is thus able to integrally assign to type-2 processors all the tasks in τ that are integrally assigned to type-2 processors in solution S, and assign no greater fraction of the task τ_x (which is same task as τ_f) to type-2 processor than in solution S. So, SA succeeds in assigning all the tasks and hence this leads to a contradiction.

Thus, for the case when there is a fractional task in the swap solution, we have shown that all the three sub-cases lead to contradiction.

Part 2: There is no fractional task in solution S. Let τ_f be the first task that is integrally assigned to type-2 processor in S. Again, we need to consider three cases with respect to the position of x and f.

Case 2.1 ($x < f$)**:** This case is analogous to Case 1.1 and leads to contradiction.

Case 2.2 ($x > f$)**:** This case is analogous to Case 1.2 and leads to contradiction.

Case 2.3 ($f = x$)**:** This indicates that SA algorithm was able to assign tasks $\{\tau_1, \ldots, \tau_{x-1}\} \in L$ integrally to type-1 processors as in S. However, it failed to integrally assign tasks $\{\tau_x, \ldots, \tau_\ell\} \in L$ to type-2 processors that are integrally assigned in S. This means $U^2 + \sum_{i=x}^{\ell} u_i^2 > m_2$ whereas $U^2 + \sum_{i=f}^{\ell} u_i^2 \leq m_2$. From the case (i.e., $f = x$), this is a contradiction and hence SA would also succeed in assigning those tasks to type-2 processors.

Thus, for the case when there is no fractional task in the swap solution, we have shown that all the three sub-cases lead to contradiction.

From Parts 1 and 2 of the proof, we have shown that all the cases lead to contradiction, hence proving the theorem. □

Theorem 5 (**Speed competitive ratio of SA**)**.** *If there exists a feasible intra-migrative assignment of τ on π then, using* SA*, it is guaranteed to obtain such a feasible intra-migrative assignment of τ on $\pi^{(1+\frac{\alpha}{2})}$.*

Proof. We know from Theorem 4 that if τ is intra-migrative feasible on π then SA succeeds in returning a feasible assignment of τ on π in which at most one task from L is fractionally assigned and the rest are integrally assigned to type-1 and type-2 processors. It follows from Lemma 4 that this fractional task can also be assigned integrally to one of the processor types if given a platform in which processors are $1 + \frac{\alpha}{2} \leq 1.5$ times faster. Hence the proof. □

We now show that the proven speed competitive ratio $1 + \frac{\alpha}{2} \leq 1.5$ of SA algorithm is a tight bound. This is shown using the same technique that was used earlier (Theorem 3 in Section 3.5) to show that the proven speed competitive ratio of LP-Algo is a tight bound and also the same problem instance is used here (and for the sake of convenience, the problem instance is repeated).

Theorem 6 (Speed competitive ratio of SA algorithm is tight)**.** *The speed competitive ratio* $1 + \frac{\alpha}{2} \leq 1.5$ *of* SA *algorithm is a tight bound.*

Proof. In order to show that the speed competitive ratio is tight for SA algorithm, it is sufficient to show that, there exists a (feasible intra-migrative) problem instance for which SA needs 1.5 times faster processors to output a feasible intra-migrative assignment. We now show that such a problem instance exists.

Consider a problem instance with a task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ comprising three tasks and a two-type platform $\pi = \{\pi_1, \pi_2\}$ comprising two processors. Let π_1 be a processor of type-1 and π_2 be a processor of type-2. The utilizations of tasks are shown in Table 3.5.

Observe that the given task set τ is intra-migrative feasible on the given platform π. A feasible intra-migrative assignment is obtained by assigning (i) τ_1 and τ_3 to type-1 processors (which has a single processor, π_1) and (ii) τ_2 to type-2 processors (which has a single processor, π_2). This assignment is shown in Table 3.6.

Tasks	Utilizations of tasks	
	$\mathbf{u_i^1}$	$\mathbf{u_i^2}$
τ_1	0.5	0.5
τ_2	1.0	1.0
τ_3	0.5	0.5

Table 3.5: An example to illustrate that the proven speed competitive ratio of SA algorithm is a tight bound.

Now consider SA algorithm. Initially, the task set is partitioned using Expressions (3.6)–(3.9) as follows: $H12 = \emptyset$, $H1 = \emptyset$, $H2 = \emptyset$ and $L = \{\tau_1, \tau_2, \tau_3\}$. Since all the tasks in the task set are light, SA sorts the tasks in non-increasing order of $\frac{u_i^2}{u_i^1}$. Since this ratio is same for all the three tasks, a sorted order is as follows: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. With this sorted order, SA assigns the tasks as shown in Table 3.7. In the assignment output by SA (which is shown in Table 3.7), it holds that:

- type-1 processors are fully utilized
- type-2 processors are fully utilized and
- task τ_2 is equally split between type-1 and type-2 processors

In order to assign τ_2 integrally to type-1 processors, the speed of type-1 processors *must be* increased to 1.5. Analogously, for assigning τ_2 integrally to type-2 processors, the speed of type-2 processors *must be* increased to 1.5 as well. Therefore, a speedup of 1.5 is required to assign τ_2 integrally to one of the processor types.

Hence, the proven speed competitive ratio $1 + \frac{\alpha}{2} \leq 1.5$ of SA algorithm is a tight bound. □

Remark 3 Although Theorem 5 states that, for an intra-migrative feasible task set, SA needs a platform in which *every processor* is $1 + \frac{\alpha}{2}$ times faster, in order to output a schedulable intra-type task assignment, it is trivial to see that a platform in which only *one processor* is $1 + \frac{\alpha}{2}$ times faster is sufficient (to which the *at most one* fractional task can be integrally assigned).

Corollary 4. *If there exists a feasible intra-migrative assignment of τ on $\pi(m_1, m_2)$ then, using SA, it is guaranteed to obtain such a feasible intra-migrative assignment of τ on $\pi'(m_1 + 1, m_2)$, which has one additional processor of type-1 compared to π.*

Proof. It follows from Theorem 4 that if there exists an intra-migrative feasible assignment of τ on π then SA succeeds in returning a feasible assignment of τ on π in which at most one task from L, say τ_f, is fractionally assigned and the rest are integrally assigned to type-1 and type-2 processors. From Corollary 3, we know that if such a task τ_f exists then it can be integrally assigned to the set of type-1 processors, which has an additional processor in π'. Hence the proof. □

Remark 4 It is trivial to see that Corollary 4 holds true if SA is given a platform $\pi'(m_1, m_2 + 1)$, which has one additional processor of type-2 compared to π.

Processor types	Tasks assigned
type-1 (π_1)	τ_1 and τ_3
type-2 (π_2)	τ_2

Table 3.6: A feasible intra-migrative assignment for tasks shown in Table 3.5 on platform π.

3.8 Average-case performance evaluations

After studying the theoretical bound of SA algorithm (i.e., its speed competitive ratio), we evaluate its average-case performance by generating random task sets and by computing its *necessary multiplication factor* for each task set. For a given task set, we define the necessary multiplication factor of SA algorithm as the *minimum* amount of extra processor speed that SA needs, so as to succeed in finding a feasible task-to-processor-type assignment as compared to an optimal intra-migrative task assignment algorithm. For each task set, we evaluate the performance of SA algorithm by comparing the *necessary multiplication factor* (which is computed via simulations) with the *speed competitive ratio* (which is derived theoretically). In our simulations, we observed that for vast majority of task sets, SA performed significantly better by succeeding in finding a feasible intra-migrative task assignment with necessary multiplication factor much smaller than the speed competitive ratio. We now discuss the evaluations in detail.

The problem instances (number of tasks, their utilizations and the number of processors of each type) were generated randomly. Each problem instance had at most 25 tasks and at most 3 processors of each type. We generated 100000 task sets, denoted as $\{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(100000)}\}$, which we transformed into "critically feasible intra-migrative task sets". We define a *critically feasible intra-migrative task set* as a task set which is intra-migrative feasible on a given two-type platform but rendered intra-migrative infeasible if all the task utilizations (i.e., both u_i^1 and u_i^2 of each task) are increased by an arbitrarily small factor. The intuition behind using critically feasible task sets in our simulations is that it is "hard" to find a feasible assignment for these task sets since only a few task assignments are feasible among all possible assignments. Therefore, we believe that using such task sets in the evaluations tests the limits of the algorithm and the average-case performance exhibited by the algorithm for these task sets is be a good indicator of its true potential.

To obtain an intra-migrative critically feasible task set $\tau_{\text{crit}}^{(k)}$ from a randomly generated task set $\tau^{(k)}$, $k \in [1, 100000]$, we perform the task-to-processor-type assignment of $\tau^{(k)}$ by formulating the problem as MILP (as shown in Figure 3.5) and feeding it to IBM ILOG CPLEX tool which outputs Z, the utilization of the most utilized processor type. Then, we multiply all the task utilizations by $\frac{1}{Z}$ and repeatedly feed it back to the CPLEX solver until $0.99 < Z \leq 1$ (which gives us $\tau_{\text{crit}}^{(k)}$).

For each critically feasible intra-migrative task set $\tau_{\text{crit}}^{(k)}$, we measure the *necessary multiplication factor* of algorithm SA, denoted by $\text{NMF}_{\text{SA}}^{(k)}$. We then compare $\text{NMF}_{\text{SA}}^{(k)}$ with the speed competitive ratio denoted by $\text{SCR}_{\text{SA}}^{(k)}$[3]. Algorithm 1 shows how we compute $\text{NMF}_{\text{SA}}^{(k)}$ for every

[3]Note that, as opposed to the generic definition of the speed competitive ratio provided in Section 2.5.1 of Chapter 2

Processor types	Tasks assigned by SA
type-1 (π_1)	100% of τ_1 and 50% of τ_2
type-2 (π_2)	100% of τ_3 and 50% of τ_2

Table 3.7: The task assignment output by SA algorithm for tasks shown in Table 3.5 on platform π comprising one processor of type-1 and another processor of type-2.

critically feasible intra-migrative task set, $\tau^{(k)}_{\text{crit}}$. On line 3, we initially set $\text{NMF}^{(k)}_{\text{SA}}$ to 1.0 as it denotes the speed of processors on which an optimal intra-migrative task assignment algorithm succeeds in finding a feasible assignment for $\tau^{(k)}_{\text{crit}}$. Then, we input the task set to algorithm SA (on line 5) and if SA cannot find a feasible assignment, the necessary multiplication factor $\text{NMF}^{(k)}_{\text{SA}}$ is incremented by a small value, here 0.01 (on line 7), and the original u^1_i and u^2_i of each task of $\tau^{(k)}_{\text{crit}}$ are divided by the new necessary multiplication factor (on line 8, this step can be seen as increasing the speed of every processor by 0.01) and this resulting task set is fed back to algorithm SA (on line 5). These steps (necessary multiplication factor adjustment and feeding back the derived task set) are repeated until the algorithm SA succeeds in finding a feasible intra-migrative task assignment, which gives us the *necessary multiplication factor* of SA for the task set under consideration.

Recall that we want to evaluate the average-case performance of our algorithm by measuring how well it performs compared to its theoretical bound. In this regard, for each critically feasible intra-migrative task set, $\tau^{(k)}_{\text{crit}}$, we compute the *performance ratio* $\text{PR}^{(k)}_{\text{SA}}$ (in %) of algorithm SA as follows:

$$\text{PR}^{(k)}_{\text{SA}} \stackrel{\text{def}}{=} \frac{\text{NMF}^{(k)}_{\text{SA}} - 1}{\text{SCR}^{(k)}_{\text{SA}} - 1} \times 100 \tag{3.30}$$

Note that both $\text{NMF}^{(k)}_{\text{SA}}$ and $\text{SCR}^{(k)}_{\text{SA}}$ are numbers that take a value of $1.x$ where the integral part 1 can be seen as the speed of the processors on which an optimal algorithm succeeds to find a feasible intra-migrative task assignment and the fractional part x can be seen as the increase in the speed of processors that algorithm SA requires (compared to the optimal algorithm) in order to succeed. Hence, 1 is subtracted from both $\text{NMF}^{(k)}_{\text{SA}}$ and $\text{SCR}^{(k)}_{\text{SA}}$ in the above expression. The multiplication factor 100 converts the ratio in percentage. *This expression enables us to compare the average-case performance of* SA *algorithm for task sets with different values of* α *on a same scale.* For example, for a given task set $\tau^{(k)}_{\text{crit}}$ with $\alpha = 0.1$, if SA succeeds in finding a feasible intra-migrative task assignment with $\text{NMF}^{(k)}_{\text{SA}} = 1.01$ then the value of the above ratio is 20% (since $\text{SCR}^{(k)}_{\text{SA}}$ of SA for this task set is $1 + \frac{\alpha}{2} = 1.05$) indicating that SA required only 20% faster processors than indicated by the theoretical upper bound. As another example, for a given task set in which $\alpha = 0.2$, if SA succeeds in finding a feasible intra-migrative task assignment with $\text{NMF}^{(k)}_{\text{SA}} = 1.02$ then the value of the above ratio is again 20% (since $\text{SCR}^{(k)}_{\text{SA}}$ of SA for this task

on page 16 which says that the speed competitive ratio is a property of the algorithm alone, the speed competitive ratio of SA algorithm which is shown to be $1 + \frac{\alpha}{2} \leq 1.5$, is not only a property of the algorithm but also a property of the task set as it depends on the parameter $0 < \alpha \leq 1$ whose value in turn depends on the (utilization values of the tasks in the) task set.

Algorithm 1: Pseudo-code for determining all the necessary multiplication factor, $\text{NMF}_{\text{SA}}^{(k)}$, of SA algorithm, for 100000 critically feasible intra-migrative task sets.

Input : Algorithm SA
The critically feasible intra-migrative task sets $\{\tau_{\text{crit}}^{(1)}, \tau_{\text{crit}}^{(2)}, \ldots, \tau_{\text{crit}}^{(100000)}\}$
Output: The necessary multiplication factors $\{\text{NMF}_{\text{SA}}^{(1)}, \text{NMF}_{\text{SA}}^{(2)}, \ldots, \text{NMF}_{\text{SA}}^{(100000)}\}$

```
step ← 0.01
for k = 1 to 100000 do
    τ ← τ_crit^(k);  NMF_SA^(k) ← 1.0
    while true do
        result ← call SA(τ_crit^(k), assignment) // assignment is an output variable
            which contains the task assignment information
        if result ≠ SUCCESS then
            NMF_SA^(k) ← NMF_SA^(k) + step
            τ_crit^(k) ← τ_crit^(k) × (1/NMF_SA^(k)) // both the utilizations of each task
                are divided by NMF_SA^(k)
        else
            break
        end
    end
end
return {NMF_SA^(1), NMF_SA^(2), ..., NMF_SA^(100000)} ;
```

set is $1+\frac{\alpha}{2} = 1.10$) indicating that SA required only 20% faster processors than indicated by the theoretical upper bound.

In general, for a given task set and a given algorithm, the smaller the *performance ratio* (shown in Expression (3.30)), the better the average-case performance of the algorithm. For example, if this ratio takes a value of 100% then it implies that the algorithm is not performing any better than what is indicated by its theoretical bound and if this ratio takes a smaller value, say 10%, then it implies that the algorithm is performing much better (to be precise, 90% better) than its theoretical bound. Hence, an algorithm is said to exhibit a good average-case performance if this ratio is small for many task sets.

We plot the histogram of the performance ratios for algorithm SA in Figure 3.7. As we can see from Figure 3.7, for approximately 70% of the task sets, SA succeeds in finding a feasible intra-migrative assignment within $(0-10]$% of its theoretical bound, for approximately 15% of the task sets, SA succeeds in finding a feasible intra-migrative assignment within $(10-20]$% of its theoretical bound, and so on.

To summarize, in our simulations, for the vast majority of task sets, the algorithm SA performed significantly better than indicated by its theoretical bound.

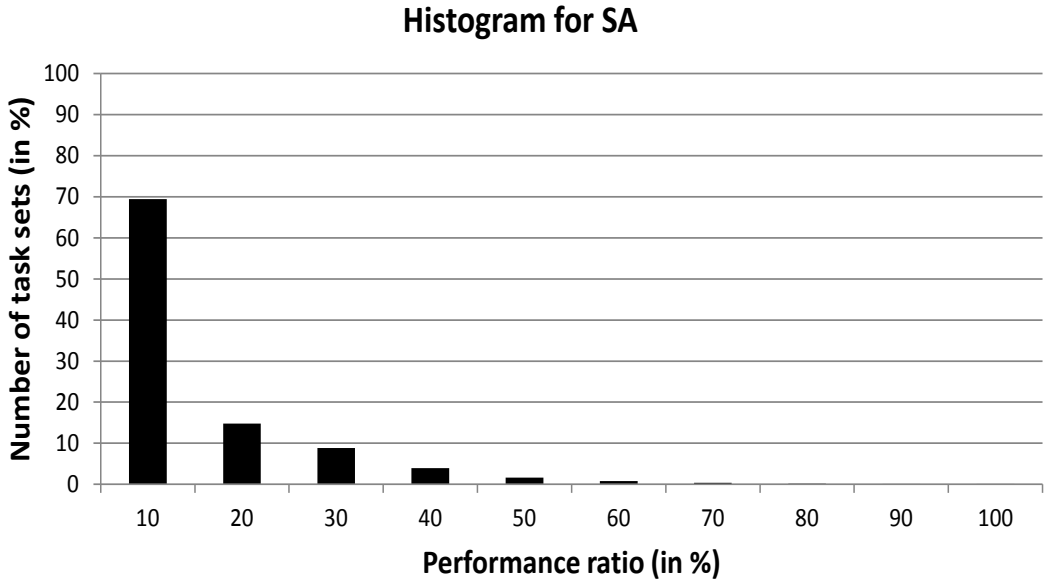


Figure 3.7: Average-case performance of SA algorithm in terms of the *performance ratio* for task sets with different values of α (if an algorithm has low performance ratio for many task sets then the algorithm is said to perform well).

3.9 Conclusions

In this chapter, we considered the problem of intra-migrative scheduling of implicit-deadline sporadic tasks on two-type heterogeneous multiprocessors. This problem can be solved in two steps: first, assign tasks to processor types and then globally schedule the tasks assigned to each processor type (since all the processors of each type can be seen an identical multiprocessor platform) using a global scheduling algorithm designed for identical multiprocessors. The global scheduling problem on identical multiprocessors is well-studied. There are couple of optimal global scheduling algorithms in literature (for example, ERFair [AS00], DP-Fair [LFS$^+$10], U-EDF [NBN$^+$12]). So, assuming that such an optimal scheduling algorithm is used to schedule the tasks on each processor type (by treating all the processors of each type as identical multiprocessors), the challenge is to assign tasks to processor types such that, for all valid job arrival patterns, *there exists* a schedule that meets all deadlines.

We showed that the problem of assigning tasks to processor types on two-type platforms is NP-Complete. We then proposed an optimal intra-migrative task assignment algorithm that relies on solving a Mixed Integer Linear Programming (MILP) formulation. Since solving MILP formulation is NP-Complete, we then relaxed this MILP formulation to LP formulation and proposed another algorithm that relies on solving this relaxed LP formulation and showed that it has a finite speed competitive ratio. Since solving a linear programming formulation is generally time consuming, we then proposed a low-degree polynomial time-complexity algorithm with a finite speed competitive ratio. Specifically, the proposed algorithm, SA, has $O(n \log n)$ time-complexity

and offers the following guarantee. If there exists a feasible intra-migrative assignment of a task set on a two-type platform then using SA, it is guaranteed to find such a feasible intra-migrative assignment for the task set but given a platform in which *one* processor is $1+\frac{\alpha}{2}$ times faster where the parameter $0<\alpha\leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than one. From the perspective of speed competitive ratio, we say that SA needs a platform in which *every* processor is $1+\frac{\alpha}{2}\leq 1.5$ times faster which defines the speed competitive ratio of SA algorithm as $1+\frac{\alpha}{2}\leq 1.5$. To the best of our knowledge, no previous algorithm exists for the problem of intra-migrative scheduling on two-type heterogeneous multiprocessors and hence SA is the first algorithm with proven performance guarantee. Although some of the non-migrative algorithms from state-of-the-art (for example, the one presented in [HS76, LST90]) can be "adapted" to intra-migrative scenario, however, these "adapted" algorithms will either end up with a significantly higher time-complexity [HS76] (which severely limits the practicality of these algorithms) or a higher speed competitive ratio [LST90] compared to our SA algorithm. We also evaluated the average-case performance of SA algorithm by generating task sets randomly and measuring how much faster processors the algorithm needs (i.e., its necessary multiplication factor), for a given task set, in order to output a feasible intra-migrative task assignment. In our simulations, we observed that, SA exhibits a good average-case behavior since, for the vast majority of the task sets, SA requires significantly smaller necessary multiplication factor than what is indicated by its theoretical bound (i.e., its speed competitive ratio).

Chapter 4

Non-migrative Scheduling on Two-type Heterogeneous Multiprocessors

4.1 Introduction

In this chapter, we consider the problem of non-migrative scheduling of tasks on two-type heterogeneous multiprocessors. Recall that in the *non-migrative* model (also referred to as *partitioned* model in the literature), every task must be statically assigned to a processor before run time and all its jobs must execute on that processor at run time (i.e., jobs *cannot* migrate between different processors). The challenge is to find, before run time, a task-to-processor assignment such that, at run time, a uniprocessor scheduling algorithm running on each processor meets all deadlines of the tasks on the respective processor. Scheduling the tasks to meet deadlines on a uniprocessor is a well-understood problem. One may use Earliest-Deadline First (EDF) [LL73], for example. EDF is an *optimal* scheduling algorithm on a uniprocessor system [LL73, Der74], with the interpretation that, for every valid job arrival pattern, if a schedule exists that meets all deadlines then EDF succeeds to construct such a schedule that meets all the deadlines as well. Therefore, assuming that an optimal uniprocessor scheduling algorithm is used on every processor, the challenging part is to find a task-to-processor assignment for which *there exists* a schedule that meets all deadlines — such an assignment is said to be a *feasible* task-to-processor assignment hereafter. Even in the simpler case of identical multiprocessors, finding a feasible task-to-processor assignment is NP-Complete in the strong sense [Joh73]. So, this result continues to hold for two-type platforms as well. In this chapter, for the problem under consideration, we propose four polynomial time-complexity algorithms (of which two are low-degree polynomial) with different speed competitive ratios which outperform state-of-the-art.

Problem Definition. In this chapter, we consider the problem of non-migrative scheduling of implicit-deadline sporadic tasks on two-type heterogeneous multiprocessors. That is, assuming that an optimal uniprocessor scheduling algorithm (such as EDF) is used on each processor to schedule the tasks, we design algorithms to determine a feasible assignment of tasks to individual processors.

Computing Platform	Adversary	Task Assignment Algorithms			
	Task migration	Algorithm	Task migration	Speed competitive ratio	Complexity
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	non-migrative	FF-3C (Section 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA-P (Section 4.4)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	non-migrative	LPC (Section 4.5)	non-migrative	1.5 (and 3 extra processors)	$O(P)$
2-type	non-migrative	PTAS_{NF} (Section 4.6)	non-migrative	PTAS	exponential in $1/\varepsilon$

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that although an LP formulation can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 4.1: Summary of the state-of-the-art task assignment algorithms for heterogeneous multiprocessors along with the algorithms proposed in this chapter.

Related work. The partitioning problem on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, RABN12, RAB13, RN12b, WBB13]. It is a well-known fact that the problem under consideration is equivalent to the problem of scheduling a set of non-real-time jobs, arriving at time zero, on unrelated parallel machine, so that they all finish before a specified time and this equivalent problem is studied in [HS76, LST90, JP99, CSV12]. In [Bar04c, Bar04b, LST90], the authors propose algorithms for the problem of non-migrative task assignment on heterogeneous multiprocessors with a speed competitive ratio of 2 against an equally powerful non-migrative adversary. All these approaches [Bar04c, Bar04b, LST90] focused on generic heterogeneous multiprocessor platforms with two or more processor types and the task assignment was modeled as Zero-One ILP. Such a formulation can be solved directly but has high computational complexity. In particular, the decision problem ILP is NP-complete and even with knowledge of the structure of the constraints in the modeling of heterogeneous multiprocessor scheduling, no polynomial-time algorithm is known (see [GJ79], p. 245). However, via relaxation of ILP formulation to LP and certain tricks [Pot85], these approaches [Bar04c, Bar04b, LST90] attain polynomial time-complexity. None of these algorithms, however, attains low-degree (linear or quadratic) polynomial time-complexity.

Moving to algorithms whose speed competitive ratios have been proven against a more powerful adversary, recently, in [CSV12], authors propose a non-migrative algorithm with a speed competitive ratio of 4 against the fully-migrative adversary. Further, it is also shown that, this bound is *exact*, i.e., it is *impossible* to design a non-migrative algorithm with a speed competitive ratio smaller than 4 against the fully-migrative adversary [CSV12].

In [HS76, JP99, WBB13], authors propose *polynomial-time approximation scheme*s (PTAS) for this problem. A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal. PTAS is theoretically a significant result since such algorithms partition the task set in polynomial time, to any desired degree of accuracy. However, (most often) their practical significance is severely limited since they incur a very high run-time complexity.

The state-of-the-art (along with the contributions of this chapter) is summarized in Table 4.1. Each row in the table corresponds to a different algorithm. For example, the first row in the table is read as follows: for a generic t-type heterogeneous multiprocessor platform in which there can be two or more types of processors, a non-migrative algorithm is proposed in [Bar04b] and it has been shown that this algorithm has a speed competitive ratio of 2 against equally powerful non-migrative adversary and the algorithm has a time-complexity of $O(P)$ (explained in Table 4.1). For the benefit of the reader, this table is repeated at several places in this chapter, especially in the sections that introduce and describe a new algorithm.

Contributions and Significance of the work discussed in this chapter. This chapter proposes four polynomial time-complexity algorithms for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors. The first algorithm, FF-3C, relies on bin-packing heuristics to output the task assignment. Its speed competitive ratio is proven against an equally powerful non-migrative adversary. The second algorithm, SA-P, is an extension of algorithm SA (discussed in Chapter 3), and for SA-P, the speed competitive ratio is proven against a more powerful intra-migrative adversary. The third algorithm, LPC, relies on solving linear programming formulation to find the task assignment. Its speed competitive ratio is proven against an equally powerful non-migrative adversary. The fourth and the last non-migrative algorithm, PTAS_{NF}, is a Polynomial Time Approximation Scheme (PTAS) which makes use of dynamic programming techniques for determining the task assignment and its speed competitive ratio is proven against an equally powerful non-migrative adversary.

The significance of each algorithm is listed in the section in which the algorithm is described. The overall significance of this chapter can be summarized as follows. First, this is the first work (FF-3C algorithm) to show how bin packing heuristics can be applied to the problem of non-migrative task assignment on two-type heterogeneous multiprocessors to obtain an algorithm with a finite speed competitive ratio. Second, this is the first work (LPC algorithm) to show how cutting planes can be used to improve the speed competitive ratio of algorithms for assigning real-time tasks to processors. Third, this work (PTAS_{NF}) shows how to design a polynomial time approximation scheme for non-migrative task assignment on two-type heterogeneous multiprocessors

HET2-NON-ASSIGN PROBLEM	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. The utilization of a task τ_i on a processor of type-t is given by u_i^t where $i \in \{1,2,\ldots,n\}$ and $t \in \{1,2\}$.
Problem	Find an assignment $f : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ such that $\forall j \in$ type-t of π, it holds that: $\sum_{i:f(i)=j} u_i^t \leq 1$, where $t \in \{1,2\}$.

Figure 4.1: The non-migrative task assignment problem on two-type heterogeneous multiprocessors.

which is efficient to be usable in practice.

Organization of the chapter. The rest of the chapter is organized as follows. Section 4.2 discusses the hardness of the non-migrative task assignment problem on two-type heterogeneous multiprocessors. Section 4.3 discusses FF-3C algorithm, proves its speed competitive ratio. It also discusses a couple of variants of FF-3C (which exhibit a better average-case performance) and proves their speed competitive ratios as well and lastly presents average-case performance evaluations. Section 4.4 discusses another algorithm, namely SA-P, proves its speed competitive ratio and also presents average-case performance evaluations. Section 4.5 presents LPC algorithm and proves its speed competitive ratio and Section 4.6 describes the polynomial time approximation scheme, PTAS_{NF}, proves its speed competitive ratio and presents average-case performance evaluations. Finally, Section 4.7 concludes.

4.2 The hardness of the non-migrative task assignment problem

In this section, we show that the problem of non-migrative task assignment on a two-type heterogeneous multiprocessor platform is NP-Complete in the strong sense. We denote this problem as HET2-NON-ASSIGN and is stated in Figure 4.1. In order to show this, we will first consider a *restricted version* of this problem which is denoted as HET2-NON-ASSIGN-SPEC-CASE — see Figure 4.2. We will show that this problem is NP-complete in the strong sense. It then follows that the HET2-NON-ASSIGN problem is NP-complete in the strong sense as well.

For showing that the HET2-NON-ASSIGN-SPEC-CASE problem is NP-Complete in the strong sense, we make use of the 3-PARTITION problem. The 3-PARTITION problem is shown in Figure 4.3 and it is well-known that this problem is NP-Complete in the strong sense [GJ78].

Lemma 7. *The HET2-NON-ASSIGN-SPEC-CASE problem is NP-Complete in the strong sense.*

Proof. In order to show that a problem is NP-Complete in the strong sense, we need to: (1) show that the problem is in NP, (2) transform a problem which is NP-Complete in the strong sense to the problem under consideration and (3) show that the transformation (of Step (2)) can be done in polynomial time. We now show these for HET2-NON-ASSIGN-SPEC-CASE problem.

1. It is straightforward to see that the problem belongs to NP. As a *certificate*, we take the assignment on each processor. To check whether the given assignment in fact satisfies

HET2-NON-ASSIGN-SPEC-CASE PROBLEM	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. The utilization of a task τ_i on a processor of type-t is given by u_i^t where $i \in \{1,2,\ldots,n\}$ and $t \in \{1,2\}$. Assume that: $\forall \tau_i \in \tau : u_i^1 = u_i^2$ and $\forall \tau_i \in \tau, \forall t \in \{1,2\} : \frac{1}{4} < u_i^t < \frac{1}{2}$ and $\frac{1}{m} \times \left(\sum_{i \in \{1,2,\ldots,n\}} u_i^1\right) = \frac{1}{m} \times \left(\sum_{i \in \{1,2,\ldots,n\}} u_i^2\right) = 1$.
Problem	Find an assignment $f : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ such that $\forall j \in$ type-t of π, it holds that $\sum_{i:f(i)=j} u_i^t \leq 1$, where $t \in \{1,2\}$.

Figure 4.2: A restricted version of the non-migrative task assignment problem on two-type heterogeneous multiprocessors.

$\sum_{i:f(i)=j} u_i^t \leq 1$ for every processor $j \in$ type-t of π (where $t \in \{1,2\}$) is obviously possible in polynomial time; specifically the time complexity is $O(n)$.

2. We now transform the 3-PARTITION problem (which is NP-Complete in the strong sense [GJ78]) to the above decision problem. Given an instance $c_1, c_2, \ldots, c_{n=3m}$ and B of the 3-PARTITION problem, transform it into an instance of HET2-NON-ASSIGN-SPEC-CASE problem with $n = 3m$ tasks by computing utilizations of tasks as follows:

$$\forall \tau_i \in \tau, \forall t \in \{1,2\} : \quad u_i^t = \frac{c_i}{B} \tag{4.1}$$

We now show that (non-migrative) assignment of these $3m$ tasks on m processors is possible *if and only if* $c_1, c_2, \ldots, c_{n=3m}$ can be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j \in \{1,2,\ldots,m\} : \sum_{i \in I_j} c_i = B$. We do so by first showing, in (a), some results we will use and then showing, in (b), the implication in one direction and finally showing, in (c), the implication in the other direction.

(a) *Results we will use:*

(a.1) Let us introduce g that maps an element in $\{1,2,\ldots,3m\}$ to a processor. It is defined as follows:

$$i \in I_j \Leftrightarrow g(i) = j$$

(b) *Implication in one direction:* We now show (using g) that *if* $c_1, c_2, \ldots, c_{3m}$ can be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j \in \{1,2,\ldots,m\} : \sum_{i \in I_j} c_i = B$ *then* there is an assignment of these $3m$ tasks on m processors.

We will do so by assuming that the if-condition of (b) is true and then show that this implies that the then-condition of (b) must also be true. We know that $c_1, c_2, \ldots, c_{3m}$ can be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j \in \{1,2,\ldots,m\} : \sum_{i \in I_j} c_i = B$. Multiplying each side by $\frac{1}{B}$ and applying the definition of u_i^t on the left hand side and

3-PARTITION PROBLEM	
Instance	A list of $3m$ integers $I = \{c_1, c_2, \ldots, c_{3m}\}$ where $\forall i : c_i \geq 2$ and a bound B such that $\sum_{i=1}^{3m} c_i = mB$ and $\forall i : B/4 < c_i < B/2$.
Question	Can I be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j : \sum_{i \in I_j} c_i = B$.

Figure 4.3: The 3-partitioning problem, which is known to be NP-Complete in the strong sense [GJ78].

using the definition of g gives us:

$$\forall j \in \{1,2,\ldots,m\} : \sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^1 = 1$$
$$\forall j \in \{1,2,\ldots,m\} : \sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^2 = 1$$

Hence, we have shown that g is an assignment of tasks to processors that satisfies the constraints stated in HET2-NON-ASSIGN-SPEC-CASE problem.

(c) *Implication in the other direction:* We now show (using g) that *if* non-migrative assignment of these n tasks on m processors is possible *then* c_1, c_2, …, c_{3m} can be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j \in \{1,2,\ldots,m\} : \sum_{i \in I_j} c_i = B$.

We will do so by assuming that the if-condition of (c) is true and then show that this implies that the then-condition of (c) must also be true. We know that a non-migrative assignment of these n tasks is possible. Using the function g to express this gives us:

$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^1 \leq 1 \right) \wedge$$
$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^2 \leq 1 \right)$$

Since it is a non-migrative assignment, it also holds that (from one of the assumptions of HET2-NON-ASSIGN-SPEC-CASE problem):

$$\frac{1}{m} \times \left(\sum_{i \in \{1,2,\ldots,n\}} u_i^1 \right) = \frac{1}{m} \times \left(\sum_{i \in \{1,2,\ldots,n\}} u_i^2 \right) = 1$$

Applying this on the earlier expression gives:

$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^1 = 1 \right) \wedge$$
$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} u_i^2 = 1 \right)$$

Multiply both sides by B and using the definition of u_i^t gives us:

$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} c_i = B \right) \wedge$$
$$\forall j \in \{1,2,\ldots,m\} : \left(\sum_{\forall i \in \{1,2,\ldots,n\} \text{ such that } g(i)=j} c_i = B \right)$$

Note that these two expressions state the same thing so only one is needed. Also, we form the partitioning as follows. Let I_j be the set of all integers such that $i \in \{1,2,\ldots,n\}$ and $g(i) = j$. This gives us:

$$\forall j \in \{1,2,\ldots,m\} : \sum_{\forall i \in I_j} c_i = B$$

This satisfies the constraints of the 3-PARTITION problem.

3. Finally, it can be easily seen that the transformation from 3-PARTITION to HET2-NON-ASSIGN-SPEC-CASE using Expression (4.1) is possible in polynomial time; specifically, the time complexity is $O(n)$.

Hence the proof. □

Theorem 7. *The HET2-NON-ASSIGN problem is NP-Complete in the strong sense.*

Proof. Follows from Lemma 7 and the fact that HET2-NON-ASSIGN-SPEC-CASE problem is a restricted form of HET2-NON-ASSIGN problem. □

In the subsequent sections, we describe our four low-degree polynomial time-complexity algorithms for the problem under consideration. We also prove the speed competitive ratios of these algorithms.

4.3 FF-3C algorithm and its variants

4.3.1 Introduction

Among known task assignment schemes for multiprocessors in general (i.e., not necessarily heterogeneous), (i) bin-packing heuristics (e.g., first-fit), (ii) Integer Linear Programming (ILP) modeling and the Linear Programming (LP) relaxation approaches and (iii) dynamic programming techniques perform provably well. Bin-packing heuristics [CGJ97] are popular for task assignment but unfortunately, the proof techniques used on identical multiprocessors do not easily translate to heterogeneous multiprocessors. Traditionally, the literature offered no bin-packing heuristic for assigning real-time tasks on heterogeneous multiprocessors. Our work discussed in this section is the first one to make use of bin-packing heuristics for designing an algorithm with a finite speed competitive ratio for assigning tasks to processors on two-type heterogeneous multiprocessors.

Related Work. As discussed earlier in Section 4.1, the problem of assigning tasks to processors on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, LST90, HS76, JP99, WBB13, CSV12]. However, most of these approaches rely on Linear Programming and/or dynamic programming techniques to provide a solution. Hence, they have a high time-complexity as shown in Table 4.2. Therefore, we provide a low-degree polynomial time-complexity algorithm using bin-packing heuristics.

Computing Platform	**Adversary**	**Task Assignment Algorithms**			
	Task migration	**Algorithm**	**Task migration**	**Speed competitive ratio**	**Complexity**
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	non-migrative	FF-3C and its variants	non-migrative	$1+\alpha \leq 2$	low-degree polynomial

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 4.2: Summary of state-of-the-art task assignment algorithms along with the FF-3C algorithm proposed in this section.

Contributions and Significance of the work discussed in this section. We present a new algorithm, FF-3C, for the problem of non-migrative task assignment — this algorithm uses a

bin-packing heuristic for assigning tasks and also makes use of the fact that in bin-packing, packing small items (i.e., tasks with small utilizations) allows better performance bounds. FF-3C offers low time-complexity and provably good performance. Specifically, FF-3C (i) offers a time-complexity of $O(n \cdot \max(m, \log n))$, where n denotes the number of tasks and m denotes the number of processors and (ii) has a speed competitive ratio of $1 + \alpha \leq 2$ against the non-migrative adversary, where the parameter $0 < \alpha \leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than one. We also present several extensions to FF-3C; these offer the same time-complexity and speed competitive ratio but in addition, they offer improved average-case performance. As can be seen from Table 4.2, FF-3C has a superior performance to state-of-the-art[1]. This is because (i) FF-3C has the same speed competitive ratio as algorithms in [Bar04b, Bar04c, LST90] (whose speed competitive ratios have been proven against a non-migrative adversary) but with a better time-complexity and also has a better speed competitive ratio compared to the algorithm in [CSV12] (whose speed competitive ratio has been proven against a more powerful fully-migrative adversary) and (ii) compared to PTAS algorithms [HS76, JP99, WBB13] that offer better speed competitive ratios (for lower values of ε) but whose practical significance is severely limited as they incur a very high time-complexity (i.e., exponential in number processors or exponential in $1/\varepsilon$), our algorithm offers a significantly lower (i.e., low-degree polynomial) time-complexity.

Via experiments with randomly generated task sets, we compare the performance of FF-3C and its variants with two established state-of-the-art algorithms and their variations [Bar04b, Bar04c]. We evaluate algorithms based on (i) the average running time and (ii) the necessary multiplication factor. Overall our new algorithms compare favorably to the state-of-the-art. In particular, in our evaluations with randomly generated task sets, one of the variants of FF-3C, namely FF-4C-COMB, runs 12000 to 160000 times faster and further, for vast majority of the task sets, it has a significantly smaller necessary multiplication factor than the state-of-the-art algorithms [Bar04b, Bar04c].

A global view. The context of the new algorithm FF-3C can be visualized as shown in Figure 4.4.

Organization of Section 4.3. The rest of the section is organized as follows. Section 4.3.2 describes the system model and offers necessary preliminaries. Section 4.3.3 presents some previously known and some new results that we use while proving the speed competitive ratio of FF-3C. Section 4.3.4 formulates the algorithm FF-3C and an example illustrating the working of FF-3C algorithm is given in Section 4.3.5. Section 4.3.6 proves its speed competitive ratio and its time-complexity is discussed in Section 4.3.7. Section 4.3.8 describes the variants of FF-3C that offer better average-case performance and proves their speed competitive ratios. Section 4.3.9 offers average-case performance evaluations and finally Section 4.3.10 concludes.

[1] SA algorithm is listed in the table for the sake of completeness and since it is an intra-migrative algorithm, FF-3C cannot be compared with it.

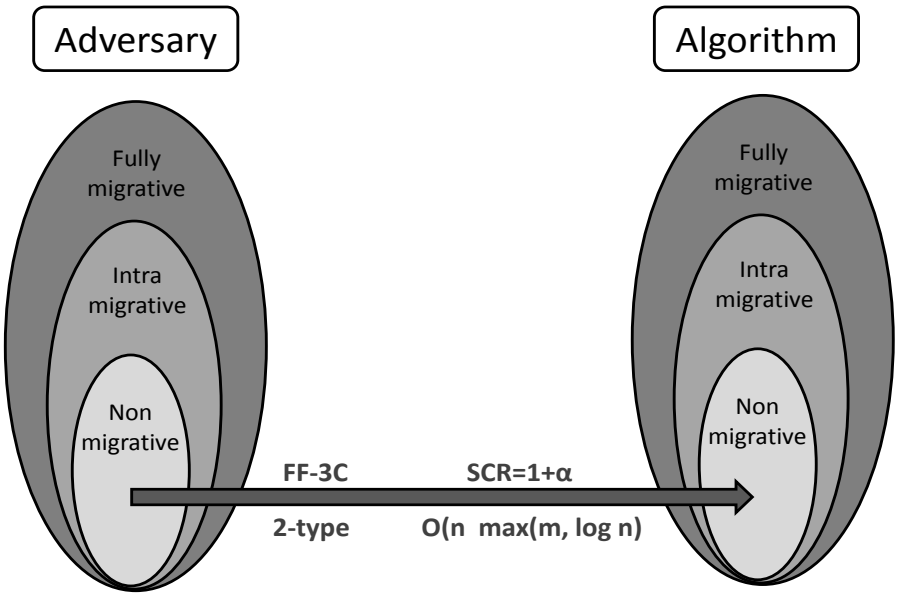


Figure 4.4: A global view of the new algorithm, FF-3C, proposed in this section. Here, SCR denotes the "speed competitive ratio", n denotes the number of tasks and m denotes the number of processors.

4.3.2 System model and some preliminaries

In a computer platform with two distinct types of processors, let P^1 be the set of type-1 processors and P^2 be the set of type-2 processors. The workload consists of τ, a set of implicit-deadline sporadic tasks.

A task is assigned to a processor and all jobs released by this task must execute there. The utilization of a task τ_i depends on the type of processor to which it is assigned. The utilization of task τ_i is u_i^1 if τ_i is assigned to a type-1 processor. Analogously, the utilization of task τ_i is u_i^2 if τ_i is assigned to a type-2 processor. Note that we allow $u_i^1 = \infty$ (respectively, $u_i^2 = \infty$) if task τ_i cannot be assigned at all to a type-1 (respectively, type-2) processor.

We assume that tasks are assigned unique identifiers. This allows two tasks with the same parameters to be in a set. For example, with $u_i^1 = 0.2$, $u_i^2 = 0.4$ and $u_j^1 = 0.2$, $u_j^2 = 0.4$, we can form the set $\{\tau_i, \tau_j\}$. We also assume that processors are assigned unique identifiers. This assumption is instrumental because if we sort processors in ascending order of their identifiers then we can be sure that, when applying normal bin-packing schemes (e.g., first-fit) repeatedly on the same task set, with tasks ordered in the same way, the bin-packing scheme outputs the same task assignment for each run.

Let $\tau[p]$ denote the set of tasks assigned to a processor p. Earliest-Deadline-First (EDF) is a very popular algorithm in uniprocessor scheduling [LL73]. A slight adaptation of a previously known result [LL73] about EDF gives the following:

Lemma 8. *If all tasks in $\tau[p]$ are scheduled under EDF on a processor p (which is of type-t, where $t \in \{1,2\}$) and $\sum_{\tau_i \in \tau[p]} u_i^t \leq 1$, then all deadlines are met.*

Then the necessary and sufficient set of conditions for schedulability on a partitioned heterogeneous multiprocessor with two types of processors is the following:

$$\forall p \in P^1 : \sum_{\tau_i \in \tau[p]} u_i^1 \leq 1 \tag{4.2}$$

$$\forall p \in P^2 : \sum_{\tau_i \in \tau[p]} u_i^2 \leq 1 \tag{4.3}$$

Thus our problem of scheduling tasks on two-type heterogeneous multiprocessors is reduced to assigning tasks to processors such that the above constraints are satisfied.

We now introduce few notations that will be used later (from Section 4.3.6 onwards) while proving the speed competitive ratio of our algorithms.

Let $\Pi(|P^1|,|P^2|)$ denote a two-type heterogeneous platform comprising $|P^1|$ processors of type-1 and $|P^2|$ processors of type-2. Let $\Pi(|P^1|,|P^2|) \times \langle s_1, s_2 \rangle$ denote a two-type heterogeneous platform in which the speed of every processor of type-1 is s_1 times the speed of a type-1 processor in $\Pi(|P^1|,|P^2|)$ and the speed of every processor of type-2 is s_2 times the speed of a type-2 processor in $\Pi(|P^1|,|P^2|)$ where s_1 and s_2 are positive real-numbers.

Let $sched(A, \tau, \Pi(|P^1|,|P^2|) \times \langle s_1, s_2 \rangle)$ denote a predicate to signify that a task set τ *meets all its deadlines* when scheduled by an algorithm A on a two-type heterogeneous multiprocessor platform — $\Pi(|P^1|,|P^2|) \times \langle s_1, s_2 \rangle$. The term *meets all its deadlines* in this and other predicates means 'meets deadlines for every possible arrival of tasks that is valid as per the given parameters of τ'.

Let $sched(\text{nmo-feasible}, \tau, \Pi(|P^1|,|P^2|) \times \langle s_1, s_2 \rangle)$ denote a predicate to signify that there exists a *non-migrative-offline-feasible* preemptive schedule which meets all deadlines for the specified system. Here, *non-migrative* schedule refers to a schedule in which all jobs of a task execute on the same processor to which the task is assigned. In this predicate and other predicates, the term *offline* encompasses the schedules generated by algorithms which (i) may use inserted idle times and/or (ii) are "clairvoyant" (i.e., use knowledge of future task arrival times).

4.3.3 Useful results

Bin-packing heuristics are popular for assigning tasks on identical [LDG04, LGDG03] and uniform [AT07b, AT07a] multiprocessors (where a processor x times faster executes all tasks x times faster) because they run fast and offer finite speed competitive ratio. Yet, straightforward application of bin-packing heuristics to two-type heterogeneous multiprocessors performs poorly, as illustrated by Example 7.

Example 7. *Consider a task set τ of $2k$ tasks and a two-type heterogeneous multiprocessor Π of 2 processors (for an integer $k \geq 3$). Processor $\pi_1 \in \Pi$ is of type-1 and processor $\pi_2 \in \Pi$ is of*

type-2. Tasks indexed $1,\ldots,k$ *are characterized by* $u_i^1 = 1$, $u_i^2 = \frac{1}{k}$ *and tasks indexed* $k+1,\ldots,2k$ *are characterized by* $u_i^1 = \frac{1}{k}$, $u_i^2 = 1$.

Tasks can be assigned such that the condition of Lemma 8 is met for both processors, e.g., assigning tasks $1,\ldots,k$ *to* π_2 *and the rest to* π_1 *as shown in Figure 4.5a. Yet, the application of a normal bin-packing algorithm (designed for identical multiprocessors such as First-Fit) causes failure. These algorithms consider tasks in a sequence and each time use the condition of Lemma 8 to decide if the task in consideration can be assigned to a processor. Under First-Fit,* τ_1 *ends up on* π_1 *(as processors are considered by order of ascending index). Yet, at most one task of those indexed* $1,\ldots,k$ *can be assigned there. Thus, the* $k-1 \geq 2$ *tasks indexed* $2,\ldots,k$ *will have to be assigned to* π_2*. Next, the bin-packing scheme tries to assign tasks* $k+1,\ldots,2k$ *to* π_2*; none fits and the algorithm fails.*

Let us now provide this bin-packing algorithm with processors $k-1$ *times faster. Then, tasks indexed* $1,\ldots,k-1$ *will be assigned to* π_1 *and the* k^{th} *task to* π_2 *before considering tasks indexed* $k+1,\ldots,2k$*. Of the latter, many can be assigned to* π_2 *but not all and, since none can be assigned to* π_1*, the bin-packing algorithm would again fail as shown in Figure 4.5b.*

This holds for any $k \geq 3$*. For* $k \to \infty$*, we see that the speed competitive ratio of such bin-packing schemes is infinite.*

It can be seen that the cause of low performance of such a bin-packing scheme is that, by considering tasks one by one, it lacks a "global view" of the problem, hence may assign a task to a processor where it executes slowly. It seems a good idea to try to assign each task to the processor where it executes faster. We will use this idea; let us thus introduce the following definitions:

P^1 is the set of type-1 processors and P^2 is the set of type-2 processors. The task set τ is viewed as two disjoint subsets, τ^1 and τ^2. The set τ^1 consists of those tasks which run at least as fast on a type-1 processor as on a type-2 processor; τ^2 consists of all other tasks. In notation:

$$\tau = \tau^1 \cup \tau^2 \tag{4.4}$$

$$\forall \tau_i \in \tau^1 : u_i^1 \leq u_i^2 \tag{4.5}$$

$$\forall \tau_i \in \tau^2 : u_i^1 > u_i^2 \tag{4.6}$$

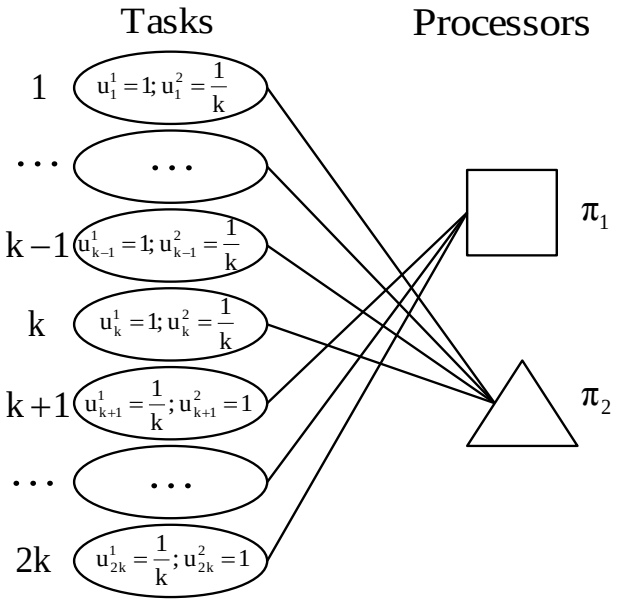
We now list two useful observations along with their proofs.

Lemma 9. *If there is a task* τ_i *in* τ^1 *such that* $u_i^1 > 1$*, it is then impossible to meet all deadlines with partitioning. Likewise for a task* τ_i *in* τ^2 *with* $u_i^2 > 1$.

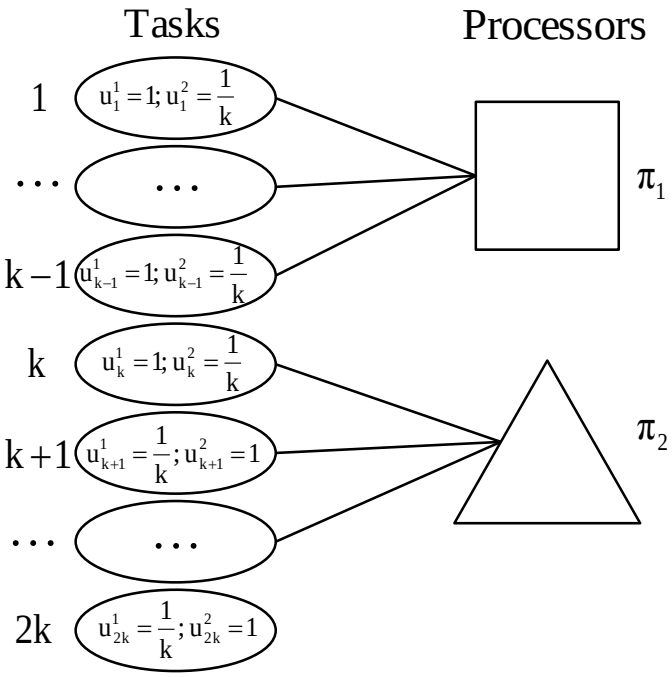
Proof. Intuitively, if the execution time of τ_i exceeds its deadline on processor type where it runs fastest, it cannot be assigned anywhere to meet deadlines. □

Lemma 10. *It is impossible to meet all deadlines of a task set* τ *on a two-type platform* Π *if*

$$\sum_{\tau_i \in \tau^1} u_i^1 + \sum_{\tau_i \in \tau^2} u_i^2 > |P^1| + |P^2| \tag{4.7}$$



(a) A successful assignment of tasks on processors.



(b) The standard first-fit bin packing fails to assign tasks on $k-1$ times faster processors.

Figure 4.5: The standard first-fit (or any other) bin-packing heuristic does not perform well for assigning tasks on two-type heterogeneous multiprocessor platform.

Proof. The proof is by contradiction. Let τ be a task set for which Inequality 4.7 holds and for which a feasible partitioning exists. Given that τ is feasible, the set of constraints expressed by

Inequalities 4.2 and 4.3 must hold. Then, respectively from those inequalities, we have:

$$\forall p \in P^1 : \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^1 + \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^1 \leq 1 \tag{4.8}$$

$$\forall p \in P^2 : \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^2 + \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^2 \leq 1 \tag{4.9}$$

However, from Inequalities 4.6 and 4.5, we know that:

$$(4.6) \Rightarrow \quad \forall \tau_i \in \tau^2 : u_i^1 > u_i^2 \tag{4.10}$$

$$\text{and } (4.5) \Rightarrow \quad \forall \tau_i \in \tau^1 : u_i^1 \leq u_i^2 \tag{4.11}$$

Then, respectively:

$$(4.8) \overset{(4.10)}{\Rightarrow} \quad \forall p \in P^1 : \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^1 + \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^2 < 1 \tag{4.12}$$

$$(4.9) \overset{(4.11)}{\Rightarrow} \quad \forall p \in P^2 : \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^1 + \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^2 \leq 1 \tag{4.13}$$

We can combine Inequalities 4.12 and 4.13 into:

$$\forall p \in \Pi : \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^1 + \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^2 \leq 1 \tag{4.14}$$

Via summation of Inequality 4.14 over all p we obtain

$$\begin{aligned} &\sum_p \sum_{\tau_i \in \tau[p] \cap \tau^1} u_i^1 + \sum_p \sum_{\tau_i \in \tau[p] \cap \tau^2} u_i^2 \leq \sum_p 1 \\ &\Rightarrow \sum_{\tau_i \in \tau^1} u_i^1 + \sum_{\tau_i \in \tau^2} u_i^2 \leq |P^1| + |P^2| \end{aligned} \tag{4.15}$$

This contradicts Inequality 4.7. □

We next highlight how the problem in consideration is related to fractional knapsack problem, to help with proofs later. If you read this chapter for the first time, you may want to skip this section now and revisit it later.

Fractional Knapsack Problem: A vector x has n elements. The problem instance is represented by vectors v and w of real numbers, arranged such that $\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}}$ $\forall i \in \{1, 2, \ldots, n-1\}$. Intuitively, v_i and w_i may be thought of as, respectively, the "value" and "weight" of an element. Consider the problem of assigning values to the elements in vector x so as to maximize $\sum_{i=1}^{n} x_i \cdot v_i$ subject to $\sum_{i=1}^{n} x_i \cdot w_i \leq \text{CAP}$ where x_i is a real number such that $0 \leq x_i \leq 1$ and CAP is a given upper bound. Intuitively, determine how much of each item to use such that cumulative value is maximized, subject to cumulative weight not exceeding some bound.

Lemma 11. *An optimal solution to the Fractional Knapsack Problem is obtained by Algorithm 15.*

Algorithm 2: An optimal algorithm for fractional knapsack problem.

```
1  re-index tuples {v_i, w_i} by order of descending v_i/w_i
2  for i=1 to n do x_i := 0;
3  end
4  i := 1; SUMWEIGHT:=0; SUMVALUE:=0
5  while (SUMWEIGHT+w_i ≤ CAP) ∧ (i ≤ n) do
6  |   x_i:=1
7  |   SUMWEIGHT:=SUMWEIGHT+w_i
8  |   SUMVALUE:=SUMVALUE+v_i
9  |   i:=i+1
10 end
11 if i ≤ n then
12 |   x_i:=(CAP – SUMWEIGHT)/w_i
13 |   SUMWEIGHT:=SUMWEIGHT+w_i · x_i
14 |   SUMVALUE:=SUMVALUE+v_i · x_i
15 end
```

Proof. This is found in textbooks (Chap. 16.2 [CLRS01]). □

For a given problem instance in our scheduling problem, we can create an instance of a fractional knapsack problem as follows: (i) for each task in our scheduling problem, create a corresponding item in the fractional knapsack problem, (ii) the weight of an item in the fractional knapsack problem is the utilization of the corresponding task where the utilization here is taken for the processor on which the task executes fast and (iii) the value of an item in the fractional knapsack problem is how much lower the utilization of its corresponding task is when the task is assigned to the processor on which it executes fast as compared to its utilization if assigned to the processor on which it executes slowly. Informally speaking, we can see that if tasks could be split, then solving the fractional knapsack problem is equivalent to assigning tasks to processors so that the cumulative utilization of tasks is minimized. Again, informally speaking, we can then show that a task assignment minimizes the cumulative utilization of tasks assuming that (i) the cumulative utilization of tasks that are assigned to the processors on which they execute fast is sufficiently high and (ii) the tasks that are assigned to the processors where they execute fast has a higher ratio (u_i^2/u_i^1) than the ones that are not. Lemma 12 and Lemma 13 expresses this formally and proves it.

Lemma 12. *Consider n tasks and a heterogeneous multiprocessor conforming to the system model of Section 4.3.2. Let x denote a number such that* $0 \leq x \leq |P^1| \cdot (1-y)$ *where* $0 < y \leq \frac{1}{2}$. *Let* $A1$ *denote a subset of* τ^1 *such that*

$$\sum_{\tau_i \in A1} u_i^1 > |P^1| \cdot (1-y) - x \tag{4.16}$$

and for every pair of tasks $\tau_i \in A1$ *and* $\tau_j \in \tau^1 \backslash A1$ *it holds that* $\frac{u_i^2}{u_i^1} - 1 \geq \frac{u_j^2}{u_j^1} - 1$. *Let* $A2$ *denote* $\tau^1 \backslash A1$.

Let B1 denote a subset of τ^1 *such that*

$$\sum_{\tau_i \in B1} u_i^1 \leq |P^1| \cdot (1-y) - x \tag{4.17}$$

Let B2 denote $\tau \backslash B1$. *It then holds that:*

$$\sum_{\tau_i \in A1} u_i^1 + \sum_{\tau_i \in A2} u_i^2 + \sum_{\tau_i \in \tau^2} u_i^2 \leq \sum_{\tau_i \in B1} u_i^1 + \sum_{\tau_i \in B2} u_i^2 \tag{4.18}$$

Proof. Let us arbitrarily choose $A1$, $B1$ as defined. We will prove that this implies Inequality 4.18. Using Inequalities 4.16 and 4.17 we clearly get:

$$\sum_{\tau_i \in A1} u_i^1 > \sum_{\tau_i \in B1} u_i^1 \tag{4.19}$$

With this choice of $A1$ and $B1$, let us consider different instances of the fractional knapsack problem:

Instance1:

CAP = left-hand side of Inequality 4.19.

For each $\tau_i \in \tau$, create an item i with $v_i = u_i^2 - u_i^1$ and $w_i = u_i^1$

SUMVALUE_1=value of variable SUMVALUE when Algorithm 15 terminates with Instance1 as input.

Instance2:

CAP = left-hand side of Inequality 4.19.

For each $\tau_i \in A1$, create an item i with $v_i = u_i^2 - u_i^1$ and $w_i = u_i^1$

SUMVALUE_2=value of variable SUMVALUE when Algorithm 15 terminates with Instance2 as input.

Instance3:

CAP = right-hand side of Inequality 4.19.

For each $\tau_i \in B1$, create an item i with $v_i = u_i^2 - u_i^1$ and $w_i = u_i^1$

SUMVALUE_3=value of variable SUMVALUE when Algorithm 15 terminates with Instance3 as input.

Instance4:

CAP = right-hand side of Inequality 4.19.

For each $\tau_i \in \tau$, create an item i with $v_i = u_i^2 - u_i^1$ and $w_i = u_i^1$

SUMVALUE_4=value of variable SUMVALUE when Algorithm 15 terminates with Instance4 as input.

Observe that:

O1: In all four instances, it holds for each element that $\frac{v_i}{w_i} = \frac{u_i^2}{u_i^1} - 1$.

O2: Instance1 and Instance2 have the same capacity.

O3: Although Instance2 has a subset of the elements of Instance1, this subset is the subset of those

elements with the largest v_i/w_i — follows from definition of $A1$.
O4: CAP in Instance2 is exactly the sum of the weights of the elements in $A1$.
O5: From O1,O2,O3 and O4: SUMVALUE_2=SUMVALUE_1.
O6: Instance3 and Instance4 have the same capacity.
O7: Instance3 has a subset of the elements of Instance4.
O8: From O6 and O7: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_4$.
O9: Instance4 has smaller capacity than Instance1.
O10: Instance4 has the same elements as Instance1.
O11: From O9 and O10: $\text{SUMVALUE}_4 \leq \text{SUMVALUE}_1$.
O12: From O8 and O11: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_1$.
O13: From O12 and O5: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_2$.

Using O13 and the definitions of the instances of $A1$ and $B1$ and observing that the capacity of Instance2 and Instance3 are set such that all elements in either instance will fit into the respective "knapsack", we obtain:

$$\sum_{\tau_i \in B1} (u_i^2 - u_i^1) \leq \sum_{\tau_i \in A1} (u_i^2 - u_i^1) \tag{4.20}$$

Now, observing that $\tau = \tau^1 \cup \tau^2 = B1 \cup B2$ gives us:

$$\sum_{\tau_i \in \tau^1} u_i^2 + \sum_{\tau_i \in \tau^2} u_i^2 = \sum_{\tau_i \in B1} u_i^2 + \sum_{\tau_i \in B2} u_i^2 \tag{4.21}$$

Combining Expression 4.20 and 4.21 gives us:

$$\sum_{\tau_i \in \tau^1} u_i^2 + \sum_{\tau_i \in \tau^2} u_i^2 - \Big(\sum_{\tau_i \in A1} u_i^2 - \sum_{\tau_i \in A1} u_i^1 \Big) \leq \sum_{\tau_i \in B1} u_i^2 + \sum_{\tau_i \in B2} u_i^2 - \Big(\sum_{\tau_i \in B1} u_i^2 - \sum_{\tau_i \in B1} u_i^1 \Big) \tag{4.22}$$

Rearranging terms and exploiting $A2 = \tau^1 \setminus A1$ yields:

$$\sum_{\tau_i \in A1} u_i^1 + \sum_{\tau_i \in A2} u_i^2 + \sum_{\tau_i \in \tau^2} u_i^2 \leq \sum_{\tau_i \in B1} u_i^1 + \sum_{\tau_i \in B2} u_i^2$$

This is the statement of the lemma. □

Lemma 12 considers a task set τ. We can however apply this on only a subset of τ. Let us assume that $H1$ and $H2$ are two disjoint subsets of τ. By applying Lemma 12 on $\tau \setminus (H1 \cup H2)$ and then adding the same sum to both sides of Inequality 4.18, we get:

Lemma 13. *Consider n tasks and a heterogeneous multiprocessor conforming to the system model (and notation) of Section 4.3.2. Let x denote a number such that* $0 \leq x \leq |P^1| \cdot (1-y)$ *where* $0 < y \leq \frac{1}{2}$. *Let* $A1$ *denote a subset of* $(\tau^1 \setminus (H1 \cup H2))$ *such that*

$$\sum_{\tau_i \in A1} u_i^1 > |P^1| \cdot (1-y) - x \tag{4.23}$$

and for every pair of tasks $\tau_i \in A1$ *and* $\tau_j \in (\tau^1 \setminus (H1 \cup H2)) \setminus A1$ *it holds that* $\frac{u_i^2}{u_i^1} - 1 \geq \frac{u_j^2}{u_j^1} - 1$. *Let* $A2$ *denote* $(\tau^1 \setminus (H1 \cup H2)) \setminus A1$.

Let $B1$ *denote a subset of* $\tau^1 \setminus (H1 \cup H2)$ *such that*

$$\sum_{\tau_i \in B1} u_i^1 \leq |P^1| \cdot (1-y) - x \tag{4.24}$$

Let $B2$ *denote* $(\tau \setminus (H1 \cup H2)) \setminus B1$. *It then holds that:*

$$\sum_{\tau_i \in H1} u_i^1 + \sum_{\tau_i \in H2} u_i^2 + \sum_{\tau_i \in A1} u_i^1 + \sum_{\tau_i \in A2} u_i^2 + \sum_{\tau_i \in \tau^2 \setminus (H1 \cup H2)} u_i^2 \leq \sum_{\tau_i \in H1} u_i^1 + \sum_{\tau_i \in H2} u_i^2 + \sum_{\tau_i \in B1} u_i^1 + \sum_{\tau_i \in B2} u_i^2$$

Lemma 13 is used while proving the performance of our new algorithm, FF-3C, which is described in the next section.

4.3.4 The FF-3C algorithm

The new algorithm, FF-3C, is based on two ideas.

Idea1: A task should ideally be assigned to the processor type where it runs faster (termed "favorite" type).

Idea2: A task with utilization above $\frac{1}{2}$ on its non-favorite type of processor must be assigned to its favorite type of processor. This special case of Idea1 is stated separately because this facilitates creating an algorithm with the desired speed competitive ratio (which is upper bounded by 2): Since we will compare the performance of our new algorithm versus every other algorithm that uses processors of at most $\frac{1}{2}$ the speed, following Idea2 ensures that each of those tasks is assigned to the same corresponding processor type as under every other successful assignment algorithm.

Based on these ideas and the concepts of τ^1 and τ^2 (defined in Section 4.3.2), we also define the following disjoint sets:

$$H1 = \{\tau_i \in \tau^1 : u_i^2 > \tfrac{1}{2}\} \tag{4.25}$$

$$H2 = \{\tau_i \in \tau^2 : u_i^1 > \tfrac{1}{2}\} \tag{4.26}$$

$$F1 = \tau^1 \setminus H1 \tag{4.27}$$

$$F2 = \tau^2 \setminus H2 \tag{4.28}$$

A task is termed to be *heavy on type-1 processors* (respectively, type-2 processors) if its utilization on that processor type strictly exceeds $\frac{1}{2}$. Intuitively, $H1$ and $H2$ identify those tasks which should be assigned based on Idea2. $H1$ stands for "Set of tasks with type-1 processors as favorite and are heavy if they are assigned to their non-favorite processor type (type-2)". Analogous for $H2$. (Obviously, a task in $H1$ or $H2$ might also be heavy on its favorite processor type.) Also, intuitively, $F1$ and $F2$ identify those tasks which should be assigned based on Idea1. $F1$ stands for "Set of tasks that have type-1 processors as their favorite and are not heavy on either processor

Algorithm 3: FF-3C: An algorithm for assigning implicit-deadline sporadic tasks on two-type heterogeneous multiprocessors.

Input : τ denotes set of tasks; Π denotes set of processors
Output: τ[p] specifies the tasks assigned to processor p

```
1  Form sets H1,H2,F1,F2 as defined by Expressions 4.25-4.28
2  ∀p: U[p] := 0
3  ∀p: τ[p] := ∅
4  if (first-fit(H1, P^1) ≠ H1) then declare FAILURE;
5  if (first-fit(H2, P^2) ≠ H2) then declare FAILURE;
6  F11 := first-fit(F1, P^1)
7  F22 := first-fit(F2, P^2)
8  if (F11 = F1) ∧ (F22 = F2) then declare SUCCESS;
9  if (F11 ≠ F1) ∧ (F22 ≠ F2) then declare FAILURE;
10 if (F11 ≠ F1) ∧ (F22 = F2) then
11 |  F12 := F1 \ F11
12 |  if (first-fit(F12, P^2) = F12) then
13 |  |  declare SUCCESS
14 |  else
15 |  |  declare FAILURE
16 |  end
17 end
18 if (F11 = F1) ∧ (F22 ≠ F2) then
19 |  F21 := F2 \ F22
20 |  if (first-fit(F21, P^1) = F21) then
21 |  |  declare SUCCESS
22 |  else
23 |  |  declare FAILURE
24 |  end
25 end
```

type". Analogous for $F2$. From the definitions of $H1$, $H2$, $F1$, $F2$ (and Inequalities 4.5 and 4.6), we have:

$$\tau_i \in H1 \Rightarrow \quad u_i^2 > \frac{1}{2} \tag{4.29}$$

$$\tau_i \in H2 \Rightarrow \quad u_i^1 > \frac{1}{2} \tag{4.30}$$

$$\tau_i \in F1 \Rightarrow \quad u_i^1 \leq \frac{1}{2} \quad \wedge u_i^2 \leq \frac{1}{2} \tag{4.31}$$

$$\tau_i \in F2 \Rightarrow \quad u_i^1 \leq \frac{1}{2} \quad \wedge u_i^2 \leq \frac{1}{2} \tag{4.32}$$

Algorithm 3 shows the pseudo-code of the new algorithm, FF-3C. The intuition behind the design of FF-3C is that first we assign tasks to their favorite processors which would be heavy on other processor type (Lines 4-5). Then we assign the non-heavy tasks to their favorite processors (Lines 6-7). Then, if there are remaining non-heavy tasks, these have to be assigned to processors that are not their favorite (Lines 12 and 20).

Algorithm 4: first-fit(ts, ps): First-fit bin-packing algorithm for assigning tasks to processors.

Input : ts denotes set of tasks; ps denotes set of processors
Output: assigned_tasks denotes set of assigned tasks

```
1  assigned_tasks := ∅
2  If ps consists of type-1 (respectively, type-2) processors, then order ts by decreasing u_i^2/u_i^1 (respectively,
   decreasing u_i^1/u_i^2) with ties broken favoring the task with lower identifier. Sort processors in ascending order of
   their unique identifiers.
3  τ_i := first task in ts
4  p := first processor in ps
5  Let k denote the type of processor p (either 1 or 2)
6  if (U[p]+ u_i^k ≤ 1) then
7      U[p] := U[p] + u_i^k
8      τ[p] := τ[p] ∪ {τ_i}
9      assigned_tasks := assigned_tasks ∪ {τ_i}
10     if remaining tasks exist in ts then
11         τ_i := next task in ts
12         go to Line 4.
13     else
14         return assigned_tasks
15     end
16 else
17     if remaining processors exist in ps then
18         p := next processor in ps
19         go to Line 6.
20     else
21         return assigned_tasks
22     end
23 end
```

FF-3C is named after the fact that each task has three chances to be assigned using first-fit: (i) according to Idea2 (to avoid making a task heavy), (ii) assignment to its favorite and (iii) assignment to its non-favorite processor type.

As already mentioned, the FF-3C algorithm performs several passes with first-fit bin-packing. It uses the subroutine `first-fit` (see Algorithm 4 for pseudo-code) which takes two parameters, a set of tasks to be assigned using first-fit bin-packing heuristic and a set of processors to assign these tasks, and it returns the set of successfully assigned tasks. FF-3C keeps track of processor utilizations in a global vector U, initialized to zero (Line 2 in Algorithm 3).

4.3.5 An example to illustrate the working of FF-3C algorithm

In this section, we illustrate the working of FF-3C with an example.

Example 8. *Consider a two-type heterogeneous multiprocessor platform* Π *with one processor of type-1 (namely,* π_1*) and two processors of type-2 (namely,* π_2 *and* π_3*) and a task set as shown in Table 4.3. Let us see how FF-3C assigns tasks to processors. The task set* τ *is partitioned as follows:* $\tau^1 = \{\tau_1, \tau_3, \tau_6, \tau_7\}$ *and* $\tau^2 = \{\tau_2, \tau_4, \tau_5, \tau_8, \tau_9\}$ *— see Inequalities 4.5 and 4.6. On Line 1, FF-3C (pseudo-code shown in Algorithm 3) forms sets H1, H2, F1 and F2 (as defined*

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9
$\mathbf{u_i^1}$	0.60	0.70	0.14	0.35	0.98	0.10	0.25	0.60	0.15
$\mathbf{u_i^2}$	0.80	0.06	0.48	0.25	0.75	0.15	0.85	0.20	0.10

Table 4.3: An example task set to illustrate the working of FF-3C algorithm.

by Inequalities 4.25-4.28) as follows: $H1 = \{\tau_1, \tau_7\}$, $H2 = \{\tau_2, \tau_5, \tau_8\}$, $F1 = \{\tau_3, \tau_6\}$ *and* $F2 = \{\tau_4, \tau_9\}$.

On Line 4, FF-3C calls first-fit sub-routine (shown in Algorithm 4) to assign tasks in $H1 = \{\tau_1, \tau_7\}$ *on processor* π_1 *(of type-1). The first-fit sub-routine (on Line 2 in Algorithm 4) sorts the tasks in* $H1$ *in descending order of* u_i^2/u_i^1*, i.e.,* $\langle \tau_7, \tau_1 \rangle$*. The sub-routine successfully assigns both the tasks in* $H1$ *to processor* π_1*. After assigning* $H1$ *tasks, the remaining utilization of processor* π_1 *is* 0.15.

On Line 5, FF-3C calls first-fit sub-routine to assign tasks in $H2 = \{\tau_2, \tau_5, \tau_8\}$ *on processor* π_2 *and* π_3 *(of type-2). The first-fit sub-routine sorts the tasks in* $H2$ *in ascending order of* u_i^2/u_i^1*, i.e.,* $\langle \tau_2, \tau_8, \tau_5 \rangle$*. The sub-routine successfully assigns* τ_2 *and* τ_8 *to processor* π_2 *(but fails to assign* τ_5 *to* π_2*) and* τ_5 *to processor* π_3*. After assigning* $H2$ *tasks, the remaining utilization of processor* π_2 *is* 0.74 *and the remaining utilization of processor* π_3 *is* 0.25.

On Line 6, FF-3C calls first-fit sub-routine to assign tasks in $F1 = \{\tau_3, \tau_6\}$ *on processor* π_1 *(of type-1). The first-fit sub-routine sorts the tasks in* $F1$ *in descending order of* u_i^2/u_i^1*, i.e.,* $\langle \tau_3, \tau_6 \rangle$*. The sub-routine successfully assigns the task* τ_3 *to* π_1 *but fails to assign* τ_6 *to* π_1 *as there is not enough capacity left in* π_1*. After assigning* τ_3*, the remaining utilization of processor* π_1 *is* 0.01*. Hence, when first-fit returns on Line 6, we have:* $F11 = \{\tau_3\}$.

On Line 7, FF-3C calls first-fit sub-routine to assign tasks in $F2 = \{\tau_4, \tau_9\}$ *on processors* π_2 *and* π_3 *(of type-2). The first-fit sub-routine (on Line 2 in Algorithm 4) sorts the tasks in* $F2$ *in ascending order of* u_i^2/u_i^1*, i.e.,* $\langle \tau_9, \tau_4 \rangle$*. The sub-routine successfully assigns both the tasks in* $F2$ *to processor* π_2*. After assigning* τ_9 *and* τ_4*, the remaining utilization of processor* π_2 *is* 0.39*. Hence, when first-fit returns on Line 7, we have:* $F22 = \{\tau_4, \tau_9\}$.

The condition on Line 10, i.e., $(F11 \neq F1) \wedge (F22 = F2)$ *is* TRUE *and hence, new task set* $F12$ *is formed on Line 11, i.e.,* $F12 = \{\tau_6\}$*. FF-3C on Line 12 calls first-fit sub-routine to assign tasks in* $F12$ *to processors* π_2 *and* π_3 *(of type-2). The first-fit sub-routine successfully assigns the single task* τ_6 *of* $F12$ *to processor* π_2*. The remaining utilization of processor* π_2 *is* 0.24*. Since the sub-routine managed to assign all tasks in* $F12$ *to type-2 processors, FF-3C declares SUCCESS on Line 13.*

So, the final assignment of tasks to processors looks as follows: τ_1, τ_3 *and* τ_7 *are assigned to processor* π_1 *(of type-1),* $\tau_2, \tau_4, \tau_6, \tau_8$ *and* τ_9 *are assigned to processor* π_2 *(of type-2) and* τ_5 *is assigned to processor* π_3 *(of type-2).*

4.3.6 The speed competitive ratio of FF-3C algorithm

In this section, we will prove the speed competitive ratio of FF-3C. We will derive its speed competitive ratio in terms of a task set parameter, namely β. The parameter β is a property of the task set on which FF-3C is applied and it reflects the values that the task utilizations on either processor types range over. Specifically, $0 < \beta \leq 0.5$ is the smallest number such that, for each task (in the task set on which FF-3C is applied), it holds that its utilization is no greater than β or greater than $1-\beta$ on a processor of type-1 and its utilization is no greater than β or greater than $1-\beta$ on a processor of type-2.

Lemma 14. *Let β denote a real number:*

$$0 < \beta \leq \frac{1}{2} \tag{4.33}$$

Let us derive a new task set τ' from the task set τ as follows:

$$\forall \tau_i \in \tau: \;\; u_i^{1'} = \frac{u_i^1}{1-\beta} \;\; \wedge \;\; u_i^{2'} = \frac{u_i^2}{1-\beta} \tag{4.34}$$

***If** for τ, it holds that:*

$$\begin{aligned} \forall \tau_i \in \tau: \quad & (u_i^1 \leq \beta) \;\vee\; (1-\beta < u_i^1) \\ \textit{and} \quad \forall \tau_i \in \tau: \quad & (u_i^2 \leq \beta) \;\vee\; (1-\beta < u_i^2) \end{aligned} \tag{4.35}$$

then
sched(nmo-feasible, τ', $\Pi(|P^1|,|P^2|)$) $\Rightarrow$ sched(FF-3C, τ, $\Pi(|P^1|,|P^2|)$)

Proof. An equivalent claim is that if a task set τ is not schedulable under FF-3C over a computing platform Π then the task set τ' would likewise be unschedulable, using any algorithm, over platform Π. We will prove this by contradiction.

Combining the definitions of H1–F2 (Inequalities 4.29–4.32), the definition of β (Inequality 4.33 in Lemma 14) and the assumptions of task set τ (Inequality 4.35 in Lemma 14), we obtain:

$$\tau_i \in H1 \overset{(4.29)}{\Rightarrow} \quad u_i^2 > \tfrac{1}{2} \overset{(4.33)}{\Rightarrow} u_i^2 \not\leq \beta \overset{(4.35)}{\Rightarrow} \quad u_i^2 > 1-\beta \tag{4.36}$$

$$\tau_i \in H2 \overset{(4.30)}{\Rightarrow} \quad u_i^1 > \tfrac{1}{2} \overset{(4.33)}{\Rightarrow} u_i^1 \not\leq \beta \overset{(4.35)}{\Rightarrow} \quad u_i^1 > 1-\beta \tag{4.37}$$

$$\tau_i \in F1 \overset{(4.31)}{\Rightarrow} \quad u_i^1 \leq \tfrac{1}{2} \wedge u_i^2 \leq \tfrac{1}{2} \overset{(4.33)}{\Rightarrow} u_i^1 \not> 1-\beta \wedge u_i^2 \not> 1-\beta \overset{(4.35)}{\Rightarrow} u_i^1 \leq \beta \wedge u_i^2 \leq \beta \tag{4.38}$$

$$\tau_i \in F2 \overset{(4.32)}{\Rightarrow} \quad u_i^1 \leq \tfrac{1}{2} \wedge u_i^2 \leq \tfrac{1}{2} \overset{(4.33)}{\Rightarrow} u_i^1 \not> 1-\beta \wedge u_i^2 \not> 1-\beta \overset{(4.35)}{\Rightarrow} u_i^1 \leq \beta \wedge u_i^2 \leq \beta \tag{4.39}$$

Assume that FF-3C failed to assign τ on Π but it is possible (using an algorithm OPT) to assign τ' on Π. Since FF-3C failed to assign τ on Π, it must have declared FAILURE. We explore all possibilities for the failure of FF-3C to occur:

Failure on Line 4 in FF-3C.
It has been shown [LDG04] that if first-fit (or any other *reasonable allocation algorithm*[2]) is used and

$$\sum_{\tau_i \in \tau} u_i \leq m - (m-1)u_{max}$$

then the task set is successfully assigned on an identical multiprocessor platform; where m is the number of processors and u_{max} is the maximum utilization of any task in the given task set.

Clearly, from trivial arithmetic, we have $m(1-u_{max}) <= m-(m-1)u_{max}$ and this gives us the following: if first-fit (or any other reasonable allocation algorithm) is used and

$$\sum_{\tau_i \in \tau} u_i \leq m(1-u_{max})$$

then the task set is successfully assigned on an identical multiprocessor platform.

Applying the above expression to the tasks in $H1$ for which it holds that $\forall \tau_i \in H1 : u_i^1 \leq \beta$ (shown later in the proof, immediately after Expression 4.55) and to the type-1 processors, we obtain:

$$\text{If } \sum_{\tau_i \in H1} u_i \leq |P^1|(1-\beta) \text{ then first-fit succeeds.}$$

Since FF-3C failed (because first-fit failed), it must hold that

$$\sum_{\tau_i \in H1} u_i^1 > |P^1| \cdot (1-\beta) \overset{(4.34)}{\Rightarrow} \sum_{\tau_i \in H1} u_i^{1'} > |P^1|$$

Therefore, OPT cannot assign all tasks in $H1$ to P^1. Hence, it assigns at least one task $\tau_i \in H1$ to P^2. From Expression 4.34 and 4.36 we get $u_i^{2'} > 1$, hence (from Lemma 9) OPT produces an infeasible assignment – a contradiction.

Failure on Line 5 in FF-3C.
This results in contradiction (symmetric to the case above).

Failure on Line 9 in FF-3C.
From the case, we obtain that $F11 \subset F1$ and $F22 \subset F2$. Therefore, when executing Line 6 in FF-3C, there was a task $\tau_{failed1} \in F1$ which could not be assigned on any processor in P^1 and when executing Line 7 in FF-3C there was a task $\tau_{failed2} \in F2$ which could not be assigned on any processor in P^2. Hence:

$$\forall p \in P^1 : U[p] + u^1_{failed1} > 1 \tag{4.40}$$

$$\text{and } \forall p \in P^2 : U[p] + u^2_{failed2} > 1 \tag{4.41}$$

where $U[p]$ is the current utilization of a processor p.

[2] A *reasonable allocation algorithm* is an algorithm that fails to assign a task *only* when there is no processor in the system that can hold the task [LDG04]. Allocation algorithms such as first-fit and best-fit are two examples of reasonable allocation algorithms.

We know from Expression 4.38 that $u^1_{failed1} \leq \beta$ and from Expression 4.39 that $u^2_{failed2} \leq \beta$. Using these on Inequalities 4.40 and 4.41 gives:

$$\forall p \in P^1 : U[p] > 1 - \beta \tag{4.42}$$

$$\text{and } \forall p \in P^2 : U[p] > 1 - \beta \tag{4.43}$$

Observing that tasks assigned on processors in P^1 are a subset of τ^1 and using Inequality 4.42 gives us:

$$\sum_{\tau_i \in \tau^1} u^1_i > |P^1| \cdot (1 - \beta) \tag{4.44}$$

With analogous reasoning, Inequality 4.43 gives us:

$$\sum_{\tau_i \in \tau^2} u^2_i > |P^2| \cdot (1 - \beta) \tag{4.45}$$

Applying Expression 4.34 on Inequalities 4.44 and 4.45, we obtain:

$$\sum_{\tau_i \in \tau^1} u^{1'}_i > |P^1| \tag{4.46}$$

$$\text{and } \sum_{\tau_i \in \tau^2} u^{2'}_i > |P^2| \tag{4.47}$$

Observing these two inequalities and Lemma 10 gives us that OPT fails to assign τ' on Π. This is a contradiction.

Failure on Line 15 in FF-3C.

From the case, we obtain that $F11 \subset F1$ and $F22 = F2$. Therefore, when executing Line 12 there was a task $\tau_{failed} \in (F1 \setminus F11)$ for which an assignment attempt was made on each of the processors in P^2. But all of these attempts failed. Therefore:

$$\forall p \in P^2 : U[p] + u^2_{failed} > 1 \tag{4.48}$$

We can add these inequalities together and get:

$$\sum_{p \in P^2} U[p] > |P^2| \cdot (1 - u^2_{failed}) \tag{4.49}$$

We know that the tasks assigned to processors in P^2 are $H2 \cup F22 \cup \tau^{F12assigned}$ where $\tau^{F12assigned}$ is the set of tasks that were assigned when executing Line 12 of FF-3C. We also know that $\tau^{F12assigned} \subset F12$. Hence, Inequality 4.49 becomes:

$$\sum_{\tau_i \in (H2 \cup F22 \cup F12)} u^2_i > |P^2| \cdot (1 - u^2_{failed})$$

From Expression 4.38, we obtain $u^2_{failed} \leq \beta$. Thus, the above inequality becomes:

$$\sum_{\tau_i \in (H2 \cup F22 \cup F12)} u^2_i > |P^2| \cdot (1-\beta) \tag{4.50}$$

We also know that FF-3C has executed Line 6 and when it performed first-fit bin-packing, there must have been a task $\tau_{failed1} \in (F1 \setminus F11)$ which was attempted to each of the processors in P^1. But all of them failed. Note that this task $\tau_{failed1}$ may be the same as τ_{failed} or it may be different. Because it was not possible to assign $\tau_{failed1}$ on any of the processors in P^1, we have:

$$\forall p \in P^1 : U[p] + u^1_{failed1} > 1 \tag{4.51}$$

Adding these inequalities together gives us:

$$\sum_{p \in P^1} U[p] > |P^1| \cdot (1 - u^1_{failed1}) \tag{4.52}$$

We know that the tasks assigned to processors in P^1 just after executing Line 6 in FF-3C are $H1 \cup F11$. Also, we know from Expression 4.38 that $u^1_{failed1} \leq \beta$. Therefore, we have:

$$\sum_{\tau_i \in (H1 \cup F11)} u^1_i > |P^1| \cdot (1-\beta) \tag{4.53}$$

Let us now discuss OPT, the algorithm which succeeds in assigning the task set τ' on platform Π. Let us discuss tasks in $H1$. From Expression 4.36, we know that:

$$\forall \tau_i \in H1 : u^2_i > 1 - \beta \tag{4.54}$$

Using Expression 4.34 gives us:

$$\forall \tau_i \in H1 : u^{2'}_i > 1 \tag{4.55}$$

If $\exists \tau_i \in H1 : u^1_i > 1-\beta$, then $\exists \tau_i \in H1 : u^{1'}_i > 1$ and using $\tau_i \in H1$ and Inequality 4.5 gives us $\exists \tau_i \in H1 \subseteq \tau^1 : u^{2'}_i > 1$. Hence such a task cannot be assigned by OPT on any processor of Π (of any type) and this is a contradiction. Hence we can assume that $\forall \tau_i \in H1 : u^1_i \leq 1-\beta$, to be precise, $\forall \tau_i \in H1 : u^1_i \leq \beta$ — see Expression 4.35. Combining this and Expression 4.34, we get:

$$\forall \tau_i \in H1 : u^{1'}_i \leq 1 \tag{4.56}$$

Using Inequalities 4.55 and 4.56 yields that every task in H1 is assigned to processors in P^1 by OPT. With analogous reasoning, we have that every task in $H2$ is assigned to a processor in P^2. Let τ^{OPT1} denote the tasks (except those from $H1$) assigned to processors in P^1 by OPT. Analogously, let τ^{OPT2} denote the tasks (except those from $H2$) assigned to processors in P^2 by OPT. Therefore

(using Inequalities 1 and 2), we know that:

$$\sum_{\tau_i \in (H1 \cup \tau^{OPT1})} u_i^{1'} \leq |P^1| \tag{4.57}$$

$$\text{and} \quad \sum_{\tau_i \in (H2 \cup \tau^{OPT2})} u_i^{2'} \leq |P^2| \tag{4.58}$$

Using Expression 4.34 gives us:

$$\sum_{\tau_i \in (H1 \cup \tau^{OPT1})} u_i^1 \leq |P^1| \cdot (1-\beta) \tag{4.59}$$

$$\text{and} \quad \sum_{\tau_i \in (H2 \cup \tau^{OPT2})} u_i^2 \leq |P^2| \cdot (1-\beta) \tag{4.60}$$

We can now reason about the inequalities we obtained about the assignments of FF-3C and OPT. Rewriting Inequalities 4.53 and 4.59 respectively yields:

$$\sum_{\tau_i \in F11} u_i^1 > |P^1| \cdot (1-\beta) - \sum_{\tau_i \in H1} u_i^1 \tag{4.61}$$

$$\sum_{\tau_i \in \tau^{OPT1}} u_i^1 \leq |P^1| \cdot (1-\beta) - \sum_{\tau_i \in H1} u_i^1 \tag{4.62}$$

We can see that Inequalities 4.61 and 4.62 with $x = \sum_{\tau_i \in H1} u_i^1$ and $y = \beta$ ensure that the assumptions of Lemma 13 are true, given also the ordering of $F1$ during assignment over P^1 (Line 2 in Algorithm 4), which ensures that $\forall \tau_i \in F11, \forall \tau_j \in F12 : \frac{u_i^2}{u_i^1} \geq \frac{u_j^2}{u_j^1}$. Using Lemma 13 gives us:

$$\sum_{\tau_i \in H1} u_i^1 + \sum_{\tau_i \in H2} u_i^2 + \sum_{\tau_i \in F11} u_i^1 + \sum_{\tau_i \in F12} u_i^2 + \sum_{\tau_i \in F22} u_i^2 \leq \sum_{\tau_i \in H1} u_i^1 + \sum_{\tau_i \in H2} u_i^2 + \sum_{\tau_i \in \tau^{OPT1}} u_i^1 + \sum_{\tau_i \in \tau^{OPT2}} u_i^2$$

Applying Inequalities 4.59 and 4.60 to the inequality above gives us:

$$\sum_{\tau_i \in H1} u_i^1 + \sum_{\tau_i \in H2} u_i^2 + \sum_{\tau_i \in F11} u_i^1 + \sum_{\tau_i \in F12} u_i^2 + \sum_{\tau_i \in F22} u_i^2 \leq |P^1| \cdot (1-\beta) + |P^2| \cdot (1-\beta) \tag{4.63}$$

Applying Inequalities 4.50 and 4.53 to left-hand side of Inequality 4.63 gives us:

$$|P^1| \cdot (1-\beta) + |P^2| \cdot (1-\beta) < |P^1| \cdot (1-\beta) + |P^2| \cdot (1-\beta) \tag{4.64}$$

This is a contradiction.

Failure on Line 23 in FF-3C.

A contradiction results – proof analogous to previous case.

We see that all cases where FF-3C declares FAILURE lead to contradiction. Hence, the lemma holds. □

Note: The value of β must depend on the utilization of tasks in the task set on which FF-3C is applied. To apply the above result for a task assignment problem, β must be assigned the

smallest value so that Expression 4.35 holds for the task set. As the value of β increases, the speed competitive ratio of FF-3C also increases.

In Lemma 14, we used β to denote a bound on the utilization of a task set (τ) on which we apply FF-3C and we stated a relation between the utilization of one task set (τ) used for FF-3C and another task set (τ') used for an optimal task assignment algorithm. It is sometimes convenient to express similar relationship but with an expression of a bound on the utilization of a task set on which we apply the optimal algorithm. For this purpose, we use α to denote a bound on the utilization of a task set (τ') on which the optimal algorithm is applied. Let $\alpha = \frac{\beta}{1-\beta}$. Algebraic rewriting gives us $\beta = \frac{\alpha}{1+\alpha}$. With this α, note that the expression $u_i^{1'} = \frac{u_i^1}{1-\beta}$ can be rewritten as: $u_i^1 = u_i^{1'} \times \left(1 - \frac{\alpha}{1+\alpha}\right)$ which in turn can be rewritten as: $u_i^1 = \frac{u_i^{1'}}{1+\alpha}$. Also, with this α, the expression $u_i^1 \leq \beta$ can be rewritten as: $u_i^{1'} \leq \alpha$. Applying this on Lemma 14 gives us:

Lemma 15. *Let α denote a real number:*

$$0 < \alpha \leq 1 \tag{4.65}$$

Let us derive a new task set τ from the task set τ' as follows:

$$\forall \tau_i \in \tau' : \;\; u_i^1 = \frac{u_i^{1'}}{1+\alpha} \;\; \wedge \;\; u_i^2 = \frac{u_i^{2'}}{1+\alpha} \tag{4.66}$$

***If** for τ', it holds that:*

$$\begin{aligned} & \forall \tau_i \in \tau' : \quad (u_i^{1'} \leq \alpha) \;\vee\; (1 < u_i^{1'}) \\ \textit{and} \quad & \forall \tau_i \in \tau' : \quad (u_i^{2'} \leq \alpha) \;\vee\; (1 < u_i^{2'}) \end{aligned} \tag{4.67}$$

then
sched(nmo-feasible, τ', $\Pi(|P^1|, |P^2|)$) $\Rightarrow$ sched(FF-3C, τ, $\Pi(|P^1|, |P^2|)$)

Proof. The proof follows from the discussion above. □

The above result can also be expressed in terms of the additional processor speed required by FF-3C as compared to that of an optimal algorithm for scheduling a given task set.

Theorem 8. *Let α denote a real number: $0 < \alpha \leq 1$.*
***If** for a task set τ', it holds that:*

$$\begin{aligned} & \forall \tau_i \in \tau' : \quad (u_i^{1'} \leq \alpha) \;\vee\; (1 < u_i^{1'}) \\ \textit{and} \quad & \forall \tau_i \in \tau' : \quad (u_i^{2'} \leq \alpha) \;\vee\; (1 < u_i^{2'}) \end{aligned}$$

***then** sched(nmo-feasible, τ', $\Pi(|P^1|, |P^2|)$) $\Rightarrow$ sched(FF-3C, τ', $\Pi(|P^1|, |P^2|) \times \langle 1+\alpha, 1+\alpha \rangle$)*

Proof. The theorem directly follows from Lemma 15. □

Note: The value of α must depend on the utilization of tasks in the task set (τ') on which the optimal algorithm is applied. To apply the above results for a task assignment problem, α must be assigned the smallest value so that Expression 4.67 holds for a given task set.

Theorem 9. *The speed competitive ratio of FF-3C is at most* 2.

Proof. This is trivial to see from Theorem 8 when α takes the maximum possible value of 1. □

Remark: Our results continue to hold if we replace first-fit with any *reasonable allocation algorithm* that has a resource augmentation bound of $1-\beta$. Another example of such an algorithm is best-fit. We have used first-fit for ease of explanation.

4.3.7 Time-complexity of FF-3C algorithm

We show that the time-complexity of FF-3C is a low-degree polynomial function of the number of tasks (n) and processors (m). By inspection of the pseudo-code for FF-3C (Algorithm 3), the function *first-fit* is invoked at most 5 times. Within each of those invocations:

- Sorting is performed over a subset of τ (i.e., at most n tasks). The time-complexity of this operation is $O(n \cdot \log n)$ e.g., using Heapsort.
- Sorting is performed over all processors, (i.e., m processors). The time complexity of this operation is $O(m \cdot \log m)$.
- First-fit bin-packing is performed whose time complexity is $O(n \cdot m)$.

Thus the time-complexity of the algorithm is at most

$$\begin{aligned} 5 \cdot \Big(\underbrace{O(n \cdot \log n)}_{\text{sort tasks}} + \underbrace{O(m \cdot \log m)}_{\text{sort processors}} + \underbrace{O(n \cdot m)}_{\text{bin-packing}} \Big) &= O(n \cdot \max(m, \log n) + m \cdot \log m) \\ &\overset{n \geq m}{=} O(n \cdot \max(m, \log n)) \end{aligned}$$

4.3.8 Variants of FF-3C algorithm

We now extend FF-3C to obtain a couple of its variants with the objective of achieving better average-case performance.

4.3.8.1 The FF-4C algorithm

One drawback of FF-3C is the early declaration of failure while trying to assign heavy tasks. If heavy tasks could not be assigned to their favorite processor type then FF-3C declares failure (on Line 4 and 5 in Algorithm 3) without even trying to assign them on their non-favorite processor type. In an extreme case, FF-3C would fail with a system composed of (i) a heavy task of type H1 (respectively, of type H2) that could fit on a processor of type-2 (respectively, type-1) and (ii) zero processors of type-1 (respectively, type-2) and infinite processors of type-2 (respectively,

type-1). FF-4C, an enhanced version of FF-3C, overcomes this drawback and hence gives better average-case performance than FF-3C. The FF-4C algorithm, upon failing to assign tasks in H1 (respectively, H2) on processors of type-1 (respectively, type-2), tries to assign those unassigned tasks onto their non-favorite processors of type-2 (respectively, type-1).

The pseudo-code of FF-4C is shown in Algorithm 5. Lines 1-3 of FF-4C are the same as that of Lines 1-3 of FF-3C (shown in Algorithm 3) and Lines 21-40 of FF-4C are same as that of Lines 6-25 of FF-3C. Lines 4-5 of FF-3C are replaced as shown in Lines 4-20 of FF-4C.

4.3.8.2 The speed competitive ratio of FF-4C algorithm

We first prove the superiority of FF-4C in terms of the task sets that it can successfully schedule as compared to that of FF-3C and then we prove the speed competitive ratio of FF-4C.

Theorem 10. *The task sets that are schedulable by FF-4C are a strict superset of those that are schedulable by FF-3C.*

Proof. To prove that the claim is true, we need to show that:

1. whenever FF-4C fails, FF-3C would also fail and
2. there is at least one task set τ for which FF-3C fails to assign τ on Π whereas FF-4C succeeds in assigning τ on Π

The intuition for proving (1) is that if $H11 = H1$ and $H22 = H2$ (i.e., the code between Lines 7-11 and 15-19 are not executed in FF-4C) then the behavior of FF-4C is exactly the same as that of FF-3C. For proving (1), we consider all the cases where FF-4C declares FAILURE and show that FF-3C will also declare FAILURE in each of those cases.

Failure on Line 10 in FF-4C.
This implies that FF-4C could not assign all the tasks in $H1$ to their favorite processor type P^1 and hence only few tasks ($H11$) were assigned to P^1 and the rest were attempted to be assigned to their non-favorite processors P^2 and failed. In such a case, FF-3C would have declared failure on Line 4 (in Algorithm 3) itself as it would also fail to assign all the tasks in $H1$ to P^1 since it also uses the same *first-fit* algorithm (of Algorithm 4) that is used by FF-4C.

Failure on Line 18 in FF-4C.
When the algorithm fails here, there are two scenarios that need to be considered with respect to the assignment of tasks in $H1$ (earlier in the algorithm): (i) all the tasks in $H1$ were successfully assigned to P^1 (indicated by $boolH1 = FALSE$, i.e., Lines 7-11 were not executed at all) and (ii) only few tasks from $H1$ could be assigned to P^1 and hence the rest were assigned to P^2 (indicated by $boolH1 = TRUE$). For the first scenario, the reasoning is *symmetric* to the previous case (i.e., the reasoning given for 'Failure on Line 10 in FF-4C' — FF-3C would have declared FAILURE on Line 5 itself as it would also fail to assign all the tasks in $H2$ to processors in P^2). For the second scenario, the proof is analogous to the previous case as FF-3C would have declared FAILURE on Line 4 (in Algorithm 3) itself as soon as a task from H1 was failed to be assigned to P^1.

Algorithm 5: FF-4C: A variant of FF-3C algorithm for assigning tasks on two-type heterogeneous multiprocessors.

Input : τ denotes set of tasks; Π denotes set of processors
Output: τ[p] specifies the tasks assigned to processor p

```
1  Form sets H1,H2,F1,F2 as defined by Expressions 4.25–4.28
2  ∀p: U[p] := 0
3  ∀p: τ[p] := ∅
4  boolH1 := FALSE; boolH2 := FALSE
5  H11 := first-fit(H1,P^1)
6  if (H11 ≠ H1) then
7  |   boolH1 := TRUE
8  |   H12 := H1 \ H11
9  |   if (first-fit(H12, P^2) ≠ H12) then
10 |   |   declare FAILURE
11 |   end
12 end
13 H22 := first-fit(H2,P^2)
14 if (H22 ≠ H2) then
15 |   boolH2 := TRUE
16 |   H21 := H2 \ H22
17 |   if (first-fit(H21, P^1) ≠ H21) then
18 |   |   declare FAILURE
19 |   end
20 end
21 F11 := first-fit(F1,P^1)
22 F22 := first-fit(F2,P^2)
23 if (F11 = F1) ∧ (F22 = F2) then declare SUCCESS;
24 if (F11 ≠ F1) ∧ (F22 ≠ F2) then declare FAILURE;
25 if (F11 ≠ F1) ∧ (F22 = F2) then
26 |   F12 := F1 \ F11
27 |   if (first-fit(F12, P^2) = F12) then
28 |   |   declare SUCCESS
29 |   else
30 |   |   declare FAILURE
31 |   end
32 end
33 if (F11 = F1) ∧ (F22 ≠ F2) then
34 |   F21 := F2 \ F22
35 |   if (first-fit(F21, P^1) = F21) then
36 |   |   declare SUCCESS
37 |   else
38 |   |   declare FAILURE
39 |   end
40 end
```

Failure on Lines 24, 30 and 38 in FF-4C.

When the algorithm fails on one of these lines, our proof depends on the assignment of tasks in

boolH1	**boolH2**	**Description of the scenario**	**Use the reasoning provided in**
FALSE	FALSE	All the tasks of $H1$ and $H2$ are assigned to their favorite processors P^1 and P^2 respectively. This indicates that the behavior of FF-4C is same as that of FF-3C in this case (i.e., code on Lines 7-11 and 15-19 of FF-4C is not executed). Hence, the reason for failure of FF-4C on Line 24, 30 and 38 is same as that of failure of FF-3C on Line 9, 15 and 23.	Proof of Lemma 14, 'Failure on Line 9, 15 and 23' respectively.
FALSE	TRUE	Only few tasks of $H2$ ($H22$) could be assigned on P^2 and the rest ($H21$) are assigned to P^1. In such a case, FF-3C would have failed on Line 5 itself during the assignment of $H2$ on P^2 as it fails to assign all the tasks from $H2$ on P^2 and does not even try to assign the failed tasks of $H2$ on P^1.	Proof of Theorem 10, 'Failure on Line 18 in FF-4C'.
TRUE	FALSE	This case is analogous to the previous case where only few tasks of $H1$ ($H11$) could be assigned to P^1 and rest ($H12$) are assigned to P^2. In this case, FF-3C would have failed on Line 4 itself during the assignment of $H1$ on P^1 as it fails to assign all the tasks from $H1$ on P^1 and does not even try to assign the failed tasks of $H1$ on P^2.	Proof of Theorem 10, 'Failure on Line 10 in FF-4C'.
TRUE	TRUE	This case is similar to one of the two previous cases, i.e., boolH1=FALSE $\wedge$ boolH2=TRUE and boolH1=TRUE $\wedge$ boolH2=FALSE	Proof of Theorem 10, 'Failure on Line 10 in FF-4C' and 'Failure on Line 18 in FF-4C' respectively.

Table 4.4: Summary of proof of speed competitive ratio of FF-4C for different scenarios.

$H1$ (respectively, $H2$) earlier in the algorithm, i.e., whether all the tasks of $H1$ (respectively, $H2$) have been successfully assigned to their favorite processors, i.e., P^1 (respectively, P^2) or only few tasks could be assigned to their favorite processors and the rest to the non-favorite processors, i.e., $H11$ on P^1 and $H12$ on P^2 (respectively, $H22$ on P^2 and $H21$ on P^1). In the FF-4C algorithm, this information is captured using the boolean variable, $boolH1$ (respectively, $boolH2$). For example, $boolH1$ = $FALSE$ indicates that all the tasks of $H1$ are assigned on their favorite processors, P^1, and $boolH1$ = TRUE implies that only few tasks from $H1$, i.e., $H11$, could be assigned on their favorite processors, P^1, and the rest, i.e., $H12$, are assigned on their non-favorite processors, P^2. Analogous explanation holds for boolean variable $boolH2$. Hence, with the help of these two boolean variables we have captured all the possible scenarios for FF-4C to fail (on one of the Lines 24, 30 or 38 in Algorithm 5) in Table 4.4 along with the corresponding proof to look for in the chapter (as the proofs provided earlier in the chapter can be reused for these scenarios).

For proving (2), we illustrate the superiority of FF-4C over FF-3C with an example task set.

Example 9. *Consider a platform comprising a processor π_1 of type-1 and a processor π_2 of type-2, a task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ shown in Table 4.5.*

τ_i	u_i^1	u_i^2	belongs to
τ_1	$\frac{1}{2}+\varepsilon$	$\frac{1}{2}+2\varepsilon$	H1
τ_2	$\frac{1}{2}+\varepsilon$	$\frac{1}{2}+2\varepsilon$	H1
τ_3	$\frac{1}{2}-\varepsilon$	$\frac{1}{2}$	F1

Table 4.5: An example task set schedulable by FF-4C but not by FF-3C.

It is trivial to observe that a schedulable assignment exists for this task set on the given platform: assign τ_1 and τ_3 to π_1 and τ_2 to π_2. We now simulate the behavior of FF-4C and FF-3C for this task set on the given platform and show that FF-4C succeeds whereas FF-3C fails.

First, let us look at FF-4C. On Line 1 (see Algorithm 5), FF-4C groups the tasks as follows: $H1 = \{\tau_1, \tau_2\}$ and $F1 = \{\tau_3\}$.

On Line 5, FF-4C calls first-fit sub-routine to assign tasks in $H1$ to processor π_1 of type-1. The sub-routine succeeds in assigning task τ_1 to π_1 and fails to assign the other task τ_2 to π_1 as there is not enough capacity left on π_1. Hence, after executing Line 5 of FF-4C, we have: $H11 = \{\tau_1\}$. After assigning τ_1 to π_1, the remaining utilization on π_1 is $\frac{1}{2}-\varepsilon$.

On Line 8, it creates $H12 = \{\tau_2\}$.

On Line 9, it successfully assigns τ_2 to processor π_2 using first-fit sub-routine. After assigning τ_2 to processor π_2; the remaining utilization on π_2 is $\frac{1}{2}-2\varepsilon$.

On Line 21, it successfully assigns τ_3 (of $F1$) to processor π_1 using first-fit sub-routine. After assigning τ_3 to π_1, the remaining utilization on π_1 is 0.

So, the final assignment of tasks is as follows: τ_1 and τ_3 are assigned to π_1 and τ_2 is assigned to π_2 — hence, FF-4C succeeds.

Now let us look at FF-3C. FF-3C groups the tasks as follows: $H1 = \{\tau_1, \tau_2\}$ and $F1 = \{\tau_3\}$. FF-3C fails to assign both the tasks in $H1$ to processor π_1 of type-1 since the sum of their utilization ($(\frac{1}{2}+\varepsilon)+(\frac{1}{2}+\varepsilon) = 1+2\varepsilon$) exceeds 1.0.

Hence FF-3C declares FAILURE on Line 4 (see Algorithm 3).

Thus, we showed that: (1) whenever FF-4C fails, FF-3C also fails and (2) there is at least one task set τ for which FF-3C fails to assign τ on platform Π whereas FF-4C succeeds in assigning τ on Π. Hence the theorem holds. □

Now we prove the speed competitive ratio of FF-4C algorithm.

Lemma 16. *Let β denote a real number: $0 < \beta \leq \frac{1}{2}$.*
Let us derive a new task set τ' from the task set τ as follows:

$$\forall \tau_i \in \tau: \;\; u_i^{1'} = \frac{u_i^1}{1-\beta} \;\; \wedge \;\; u_i^{2'} = \frac{u_i^2}{1-\beta}$$

If *for* τ*, it holds that:*

$$\begin{aligned} \forall \tau_i \in \tau: \quad & (u_i^1 \leq \beta) \ \vee \ (1-\beta < u_i^1) \\ \textit{and} \quad \forall \tau_i \in \tau: \quad & (u_i^2 \leq \beta) \ \vee \ (1-\beta < u_i^2) \end{aligned}$$

then
sched(*nmo-feasible,* τ'*,* $\Pi(|P^1|,|P^2|)) \Rightarrow$ *sched*(*FF-4C,* τ*,* $\Pi(|P^1|,|P^2|))$

Proof. We know from Lemma 14 that

$$sched(\text{nmo-feasible}, \tau', \Pi(|P^1|,|P^2|)) \Rightarrow sched(\text{FF-3C}, \tau, \Pi(|P^1|,|P^2|)) \tag{4.68}$$

Also, from Theorem 10 we know that if FF-3C succeeds to assign a task set τ on a computing platform $\Pi(|P^1|,|P^2|)$ then FF-4C succeeds as well (on the same platform). Formally, this can be stated as:

$$sched(\text{FF-3C}, \tau, \Pi(|P^1|,|P^2|)) \Rightarrow sched(\text{FF-4C}, \tau, \Pi(|P^1|,|P^2|)) \tag{4.69}$$

Combining Expression 4.68 and 4.69 gives us:

$$sched(\text{nmo-feasible}, \tau', \Pi(|P^1|,|P^2|)) \Rightarrow sched(\text{FF-4C}, \tau, \Pi(|P^1|,|P^2|))$$

Hence, the proof. □

Similar to Lemma 14, the above lemma uses β to denote a bound on the utilization of a task set (τ) on which we apply FF-4C and states a relation between the utilization of one task set (τ) used for FF-4C and another task set (τ') used for an optimal task assignment algorithm. Now, similar to Lemma 15, let us express this relationship with α, an expression of a bound on the utilization of a task set (τ') on which we apply the optimal algorithm.

Lemma 17. *Let* α *denote a real number:* $0 < \alpha \leq 1$.
Let us derive a new task set τ *from the task set* τ' *as follows:*

$$\forall \tau_i \in \tau': \ u_i^1 = \frac{u_i^{1'}}{1+\alpha} \ \wedge \ u_i^2 = \frac{u_i^{2'}}{1+\alpha}$$

If *for* τ'*, it holds that:*

$$\begin{aligned} \forall \tau_i \in \tau': \quad & (u_i^{1'} \leq \alpha) \ \vee \ (1 < u_i^{1'}) \\ \textit{and} \quad \forall \tau_i \in \tau': \quad & (u_i^{2'} \leq \alpha) \ \vee \ (1 < u_i^{2'}) \end{aligned}$$

then
sched(*nmo-feasible,* τ'*,* $\Pi(|P^1|,|P^2|)) \Rightarrow$ *sched*(*FF-4C,* τ*,* $\Pi(|P^1|,|P^2|))$

Proof. The reasoning is analogous to the proof of Lemma 15. □

Now, we express the above result in terms of the additional processor speed required by FF-4C as compared to that of an optimal algorithm for scheduling a given task set.

Theorem 11. *Let α denote a real number $0 < \alpha \leq 1$.*
***If** for a task set τ', it holds that:*

$$\forall \tau_i \in \tau' : \quad (u_i^{1'} \leq \alpha) \ \vee \ (1 < u_i^{1'})$$
$$\textit{and} \quad \forall \tau_i \in \tau' : \quad (u_i^{2'} \leq \alpha) \ \vee \ (1 < u_i^{2'})$$

***then** sched(nmo-feasible, τ', $\Pi(|P^1|,|P^2|)) \Rightarrow$ sched(FF-4C, τ', $\Pi(|P^1|,|P^2|) \times \langle 1+\alpha, 1+\alpha \rangle$)*

Proof. The theorem directly follows from Lemma 17. □

Theorem 12. *The speed competitive ratio of FF-4C is at most* 2.

Proof. The proof follows from applying $\alpha = 1$ in Theorem 11. □

4.3.8.3 Time-complexity of FF-4C algorithm

We can use the same reasoning provided for the time-complexity of FF-3C in Section 4.3.7 for FF-4C as well. FF-4C uses the first-fit sub-routine at most seven times (see Algorithm 5) and each time (i) sorting is performed over at most n tasks whose complexity is $O(n \cdot \log n)$ (ii) sorting is performed over m processors whose complexity is $O(m \cdot \log m)$ and (iii) first-fit bin-packing takes $O(n \cdot m)$ time. Hence, the time-complexity of FF-4C is: $O(n \cdot \max(m, \log n))$.

4.3.8.4 The FF-4C-NTC algorithm

In FF-3C (and also in FF-4C), tasks are categorized as H1, F1, H2 and F2 and this makes it possible to prove the speed competitive ratio the way we do it. Unfortunately, this categorization can misguide the algorithm to assign a task in a way which causes a failure later on. For example, consider a task set with two tasks τ_1 with u_1^1=0.5, u_1^2=1.0 and τ_2 with u_2^1=1.0, u_2^2=$1.0+\varepsilon$ and a two-type platform comprising a processor π_1 of type-1 and π_2 of type-2. Clearly, there exists a schedulable assignment of the given task set on the given platform: assign τ_1 to π_2 and τ_2 to π_1. Now let us see what FF-3C does for this problem instance. FF-3C classifies τ_1 and τ_2 as H1 and assigns τ_1 to π_1 and then tries to assign τ_2 to π_1 but fails and hence declares FAILURE. FF-4C also exhibits similar behavior: it assigns τ_1 to π_1 and then it attempts to assign τ_2 to π_2 after an unsuccessful attempt to assign it to π_1 and fails and hence declares FAILURE. Hence, both FF-3C and FF-4C declare FAILURE for this task set. Therefore, we present a new algorithm namely, FF-4C-NTC to handle such cases.

The algorithm FF-4C-NTC classifies tasks as τ^1 and τ^2 as defined by Inequalities 4.5 and 4.6 (on page 84), and for each class, assigns tasks in order of decreasing u_i^2/u_i^1 for type-1 processors and decreasing u_i^1/u_i^2 for type-2 processors, respectively with ties broken favoring the task with lower identifier. FF-4C-NTC does not classify τ^1 into H1 and F1 nor τ^2 into H2 and F2 (as

was the case with FF-3C and FF-4C): It only considers favorite/non-favorite processor types and disregards the information (used by both FF-3C and FF-4C) whether a task is heavy or not. The pseudo-code of FF-4C-NTC is shown in Algorithm 6. The algorithm first tries to assign tasks from τ^1 on their favorite processors, set P^1, using first-fit and if any of these tasks could not be assigned then it tries to assign them on their non-favorite processors, set P^2, and analogously for τ^2. For the above example, FF-4C-NTC assigns τ_1 to π_2 and τ_2 to π_1.

FF-4C-NTC also has the same time-complexity of $O(n \cdot \max(m, \log n))$ as the previously discussed algorithms.

Algorithm 6: FF-4C-NTC: new algorithm for assigning tasks on two-type heterogeneous multiprocessors — does not make use of the *heavy* task concept.

```
   Input  : τ denotes set of tasks; Π denotes set of processors
   Output: τ[p] specifies the tasks assigned to processor p
 1 Form sets τ^1, τ^2 as defined by Eq. 4.5 and 4.6
 2 ∀p: U[p] := 0
 3 ∀p: τ[p] := ∅
 4 τ11 := first-fit(τ^1, P^1)
 5 if (τ11 ≠ τ^1) then
 6 |   τ12 := τ^1 \ τ11
 7 |   if (first-fit(τ12, P^2) ≠ τ12) then
 8 |   |   declare FAILURE
 9 |   end
10 end
11 τ22 := first-fit(τ^2, P^2)
12 if (τ22 ≠ τ^2) then
13 |   τ21 := τ^2 \ τ22
14 |   if (first-fit(τ21, P^1) ≠ τ21) then
15 |   |   declare FAILURE
16 |   end
17 end
18 declare SUCCESS
```

This algorithm will be used as a sub-routine in our next algorithm, namely FF-4C-COMB, which is discussed next. We will not use FF-4C-NTC as a stand-alone algorithm and hence we will not discuss its speed competitive ratio.

4.3.8.5 The FF-4C-COMB algorithm

As discussed in earlier sections, for some task sets FF-4C succeeds whereas FF-4C-NTC fails and for other task sets FF-4C-NTC succeeds whereas FF-4C fails. FF-4C-COMB exploits this fact by making use of both the algorithms to get the best out of two — its pseudo-code is listed in Algorithm 7. It first attempts to assign the task set with FF-4C and, upon failing, it tries with FF-4C-NTC.

4.3.8.6 The speed competitive ratio of FF-4C-COMB algorithm

In this section, we establish the speed competitive ratio of FF-4C-COMB.

Algorithm 7: FF-4C-COMB: new algorithm for assigning tasks on two-type heterogeneous multiprocessors — combination of FF-4C and FF-4C-NTC.

```
   Input  : τ denotes set of tasks; Π denotes set of processors
   Output: returns SUCCESS or FAILURE
 1 status := FF-4C(τ, Π)
 2 if (status = FAILURE) then
 3 |   status := FF-4C-NTC(τ, Π)
 4 |   if (status = FAILURE) then
 5 |   |   declare FAILURE
 6 |   else
 7 |   |   declare SUCCESS
 8 |   end
 9 else
10 |   declare SUCCESS
11 end
```

Lemma 18. *Let β denote a real number: $0 < \beta \leq \frac{1}{2}$.*
Let us derive a new task set τ' from the task set τ as follows:

$$\forall \tau_i \in \tau: \;\; u_i^{1'} = \frac{u_i^1}{1-\beta} \;\; \wedge \;\; u_i^{2'} = \frac{u_i^2}{1-\beta}$$

***If** for τ, it holds that:*

$$\forall \tau_i \in \tau: \quad (u_i^1 \leq \beta) \;\; \vee \;\; (1-\beta < u_i^1)$$
$$\textit{and} \quad \forall \tau_i \in \tau: \quad (u_i^2 \leq \beta) \;\; \vee \;\; (1-\beta < u_i^2)$$

then
sched(nmo-feasible, τ', $\Pi(|P^1|,|P^2|)$) $\Rightarrow$ sched(FF-4C-COMB, τ, $\Pi(|P^1|,|P^2|)$)

Proof. An equivalent claim is that, if a task set τ is not schedulable under FF-4C-COMB over a computing platform Π then the task set τ' would likewise be unschedulable, using any algorithm, over computing platform Π. We will prove this by contradiction.

Assume that FF-4C-COMB has failed to assign τ on Π but it is possible (using an algorithm OPT) to assign τ' on Π. Since FF-4C-COMB failed to assign τ on Π, it follows that FF-4C-COMB declared FAILURE. We explore the only possibility for this to occur:

Failure on Line 5 in FF-4C-COMB.
For FF-4C-COMB to declare FAILURE on this line, FF-4C must have failed on Line 1 (in Algorithm 7). But, from Lemma 16 we know that

$$\textit{sched}(\text{nmo-feasible}, \tau', \Pi(|P^1|,|P^2|)) \Rightarrow \textit{sched}(\text{FF-4C}, \tau, \Pi(|P^1|,|P^2|))$$

Since FF-4C declared FAILURE, it must hold that τ' is (nmo-) infeasible on Π. Hence, OPT produces an infeasible assignment — this is a contradiction. □

As done previously for FF-3C and FF-4C, the following lemma expresses this relationship

with α, an expression of a bound on the utilization of a task set (τ') on which we apply the optimal algorithm.

Lemma 19. *Let α denote a real number: $0 < \alpha \leq 1$.*
Let us derive a new task set τ from the task set τ' as follows:

$$\forall \tau_i \in \tau' : \; u_i^1 = \frac{u_i^{1'}}{1+\alpha} \;\wedge\; u_i^2 = \frac{u_i^{2'}}{1+\alpha}$$

***If** for τ', it holds that:*

$$\begin{aligned} & \forall \tau_i \in \tau' : \quad (u_i^{1'} \leq \alpha) \;\vee\; (1 < u_i^{1'}) \\ \textit{and} \quad & \forall \tau_i \in \tau' : \quad (u_i^{2'} \leq \alpha) \;\vee\; (1 < u_i^{2'}) \end{aligned}$$

then
sched(nmo-feasible, τ', $\Pi(|P^1|,|P^2|)$) $\Rightarrow$ sched(FF-4C-COMB, τ, $\Pi(|P^1|,|P^2|)$)

Proof. The reasoning is analogous to the proof of Lemma 15. □

The following theorem expresses the above result in terms of the additional processor speed required by FF-4C-COMB as compared to that of an optimal algorithm for scheduling a given task set.

Theorem 13. *Let α denote a real number $0 < \alpha \leq 1$.*
***If** for a task set τ', it holds that:*

$$\begin{aligned} & \forall \tau_i \in \tau' : \quad (u_i^{1'} \leq \alpha) \;\vee\; (1 < u_i^{1'}) \\ \textit{and} \quad & \forall \tau_i \in \tau' : \quad (u_i^{2'} \leq \alpha) \;\vee\; (1 < u_i^{2'}) \end{aligned}$$

then
sched(nmo-feasible, τ', $\Pi(|P^1|,|P^2|)$) $\Rightarrow$
sched(FF-4C-COMB, τ', $\Pi(|P^1|,|P^2|) \times \langle 1+\alpha, 1+\alpha \rangle$)

Proof. The theorem directly follows from Lemma 19. □

Theorem 14. *The speed competitive ratio of FF-4C-COMB is at most* 2.

Proof. The proof follows from applying $\alpha = 1$ in Theorem 13. □

4.3.8.7 Time-complexity of FF-4C-COMB algorithm

We know that both FF-4C and FF-4C-NTC have the same time-complexity of $O(n \cdot \max(m, \log n))$. FF-4C-COMB (pseudo-code in Algorithm 6) calls FF-4C first and upon failing it calls FF-4C-NTC. Hence, time-complexity of FF-4C-COMB is also $O(n \cdot \max(m, \log n))$.

4.3.9 Average-case performance evaluations

After seeing the theoretical bounds of our algorithms, we wanted to evaluate their average-case performance and compare it with state-of-the-art. For this purpose, we looked at the following issues: (i) how well our algorithms perform compared to state-of-the-art in successfully assigning the tasks to processors, i.e., how much faster processors our algorithms need in order to assign a task set compared to state-of-the-art algorithms? (i.e., comparison of their necessary multiplication factors), (ii) how fast our algorithms run compared to state-of-the-art algorithms? and (iii) how much pessimism is there in the theoretically derived performance bounds of our algorithms (i.e., in speed competitive ratio)?

In order to answer these questions, we performed two sets of experiments. First, we compared the average-case performance of our algorithms with two state-of-the-art algorithms [Bar04b, Bar04c]. Both [Bar04b, Bar04c] proposed solutions with a speed competitive ratio of 2 against non-migrative adversary. Hence, we evaluated the average-case performance of our algorithms with [Bar04b, Bar04c] by setting $\alpha = 1$ for our algorithms since their speed competitive ratios become 2 as well with this setting. In our evaluations with randomly generated task sets, we observed that, our algorithms exhibit a better average-case performance than state-of-the-art algorithms [Bar04b, Bar04c]. We also observed that our algorithms run significantly faster compared to those algorithms. Then, we simulated our algorithms for different values of α. We observed that even for this improved analysis case (where the speed competitive ratio is quantified with task set parameters as opposed to a constant number [ARB10]), they still perform better than indicated by their respective speed competitive ratios. We now discuss both the cases in detail.

4.3.9.1 Comparison with state-of-the-art

We implemented two versions of [Bar04c] (referred to as SKB-RTAS and SKB-RTAS-IMP) and two versions of [Bar04b] (referred to as SKB-ICPP and SKB-ICPP-IMP). SKB-RTAS and SKB-ICPP follow from the corresponding papers; the -IMP variants are our improved versions of the respective algorithms (see description below). We implemented all algorithms using C on Windows XP on an Intel Core2 (2.80 GHz) machine. For SKB- algorithms, we also used a state-of-art LP/ILP solver, IBM ILOG CPLEX [IBM12].

In [Bar04c], a two step algorithm for assigning tasks on heterogeneous multiprocessors is proposed. The algorithm is as follows:

1. The assignment problem is formulated as ILP and then relaxed to LP. The LP formulation is solved using an LP solver. Tasks are then assigned to the processors according to the values of the respective indicator variables in the solution. Using certain tricks [Pot85], it is shown that there exists a solution (for example, the solution that lies on the vertex of the feasible region) to the LP formulation in which all but at most $m-1$ tasks are integrally assigned to processors, where m is the number of processors.

2. The remaining at most $m-1$ tasks are integrally assigned on the remaining capacity of the processors using "exhaustive enumeration".

While assigning the remaining tasks in Step 2, the author illustrates [Bar04c] with an example that the utilization of the task under consideration is compared against the value $1-z$ for assignment decisions *on any processor*, where variable z (returned by the LP solver) is the maximum utilized fraction of any processor — SKB-RTAS implements this (pessimistic) rule. Since the actual remaining capacity of each processor[3] can easily be computed from the LP solver solution, SKB-RTAS-IMP uses that, instead of $1-z$, to test assignments, for improved average-case performance.

In [Bar04b], author proposes a two step algorithm, namely *taskPartition*, to assign tasks on a heterogeneous platform. The algorithm is as follows:

1. This step is similar to Step 1 of [Bar04c] as described above.

2. The remaining at most $m-1$ tasks are assigned using the bipartite matching technique such that at most one task from the $m-1$ remaining tasks is assigned to each processor.

Let $r_1, r_2, \ldots, r_k$ denote the distinct utilization values in the given task set sorted in the increasing order, where $1 \leq k \leq m*n$. The two step algorithm is called repeatedly by a procedure, namely *optSrch*, with different values of r_i, $1 \leq i \leq k$. When *taskPartition* is called by *optSrch* with a r_i, all the utilizations that are greater than r_i are set to ∞. The procedure *optSrch* checks for the condition $U_{OPT}^{r_i} \leq 1-r_i$ in order to determine whether a feasible mapping has been obtained by $taskPartition$ where $U_{OPT}^{r_i}$ denotes the value of objective function of the vertex solution returned by LP solver — SKB-ICPP implements this feasibility test. This pessimistic condition severely impacts performance. Hence, SKB-ICPP-IMP implements a better feasibility condition which checks that the sum of utilizations of all the tasks assigned to each processor does not exceed its computing capacity thereby improving its performance significantly in practice.

We assess the average-case performance of algorithms by (i) creating a histogram of necessary multiplication factor and (ii) comparing the average running time of each algorithm. Since all the SKB- algorithms use CPLEX, an external program, for assigning tasks to processors (for solving LP), they are penalized by the startup time and reading of the problem instance from an input file — we refer to this overhead as *CPLEX overhead*. We deal with this issue by measuring the average time for CPLEX overhead and subtract it from the measured running time of those algorithms that rely on CPLEX. In particular, SKB-ICPP and SKB-ICPP-IMP invoke CPLEX multiple times for a single task set. So, we record, for such algorithms for each task set how many times CPLEX was invoked and subtract as many times the average CPLEX overhead.

We have considered the following as CPLEX overhead: (i) starting CPLEX from our program through a system call and (ii) reading of an input file (i.e., problem instance) by CPLEX. We

[3]The actual remaining capacity on processor p is $1-\sum_{i:x_{i,p}=1} u_{i,p}$ where $u_{i,p}$ represents the utilization of task τ_i on processor p [Bar04c]. The symbol $x_{i,p}$ represents the indicator variable and the value of $0 \leq x_{i,p} \leq 1$ indicates how much fraction of task τ_i must be assigned to processor p. The term $1-\sum_{i:x_{i,p}=1} u_{i,p}$ gives an accurate estimation of the remaining capacity on processor p as it ignores the fractionally assigned tasks on that processor whereas z is pessimistic since it includes those tasks as well.

measured the total time that CPLEX takes to start and read the largest input file possible for our simulation (i.e., problem involving 12 tasks and 6 processors). We measured this time for 200 iterations (same as the number of task sets for which we have computed the average execution times) and took the average of these measurements. This average value was subtracted (i) once for every measurement of SKB-RTAS and SKB-RTAS-IMP and (ii) r times for every measurement of SKB-ICPP and SKB-ICPP-IMP where r is the number of different $u_{i,j}$ values the algorithm tries for each task set.

The problem instances (number of tasks, their utilizations and number of processors of each type) were generated randomly. Each problem instance had at most 12 tasks and at most 3 processors of each type. We term a task set *critically feasible non-migrative task set* if it is feasible on a given heterogeneous multiprocessor platform but rendered infeasible if u_i^1 and u_i^2 of all the tasks in the system are increased by an arbitrarily small factor. To obtain critically feasible non-migrative task sets from randomly generated task sets, we perform the assignment with ILP as discussed in [Bar04b] and obtain z — the utilization of the most utilized processor, and then multiply all the task utilizations by a factor of $\frac{1}{z}$ and repeatedly feed back to CPLEX till $0.99 < z \leq 1$.

We ran each algorithm on 15000 critically feasible non-migrative task sets to obtain the necessary multiplication factors of each algorithm for every task set. Recall that, an algorithm is said to have a good average-case performance, if for vast majority of task sets, it has a low necessary multiplication factor. Figure 4.6 shows the comparison of all the versions of SKB- algorithms. The SKB-RTAS-IMP and SKB-ICPP-IMP with their improved tests (to check the feasibility of task assignment to processors) give better average-case performance compared to their counterparts. As we can see, SKB-RTAS-IMP gives the best average-case performance among all the SKB- algorithms.

Figure 4.7 shows the performance of all our FF- algorithms. As we can see, FF-3C performs poorly compared to the other three, and FF-4C-COMB gives the best average-case performance among all the FF- algorithms as it makes use of both FF-4C and FF-4C-NTC algorithms (whose performance lies between FF-3C and FF-4C-COMB).

Since SKB-RTAS-IMP offered the best necessary multiplication factor among all the SKB-algorithms and FF-4C-COMB offered the best necessary multiplication factor among all the FF-algorithms, we only depict these along with FF-3C since it is the baseline of all our algorithms in Figure 4.8. As can be seen, in our evaluations, the necessary multiplication factor of FF-4C-COMB never exceeded 1.35 whereas for FF-3C and SKB-RTAS-IMP this factor is close to 2.00 and 1.60, respectively. Therefore, FF-4C-COMB offers significantly better average-case performance compared to state-of-the-art.

We also measured the running times of each algorithm for the same task set. Table 4.6 shows the average running time of FF- algorithms, Table 4.7 shows the average running time of SKB-algorithms with CPLEX overhead and finally Table 4.8 shows the average running time of SKB-algorithms after subtracting the measured CPLEX overhead (from the values shown in Table 4.7). We deal with the CPLEX overhead in the SKB- algorithms for fair evaluation. We can see that, in the evaluations, our proposed algorithms all run in less than 1.1 μs (Table 4.6) but SKB- algorithms

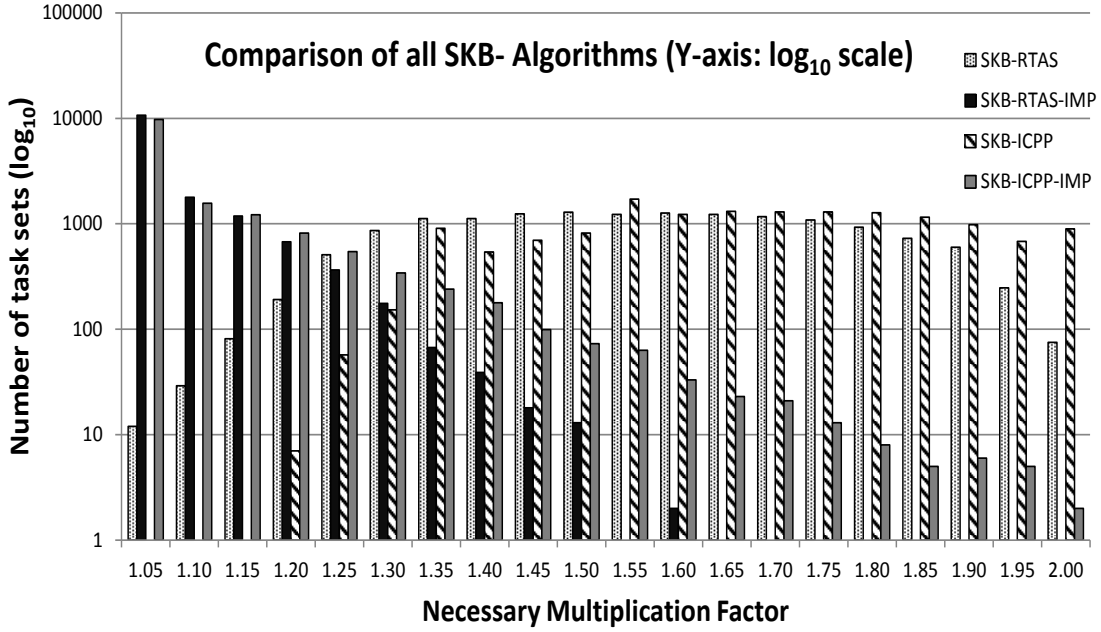


Figure 4.6: Comparison of the necessary multiplication factors for all the SKB- algorithms (if an algorithm has low necessary multiplication factor for many task sets then the algorithm is said to perform well).

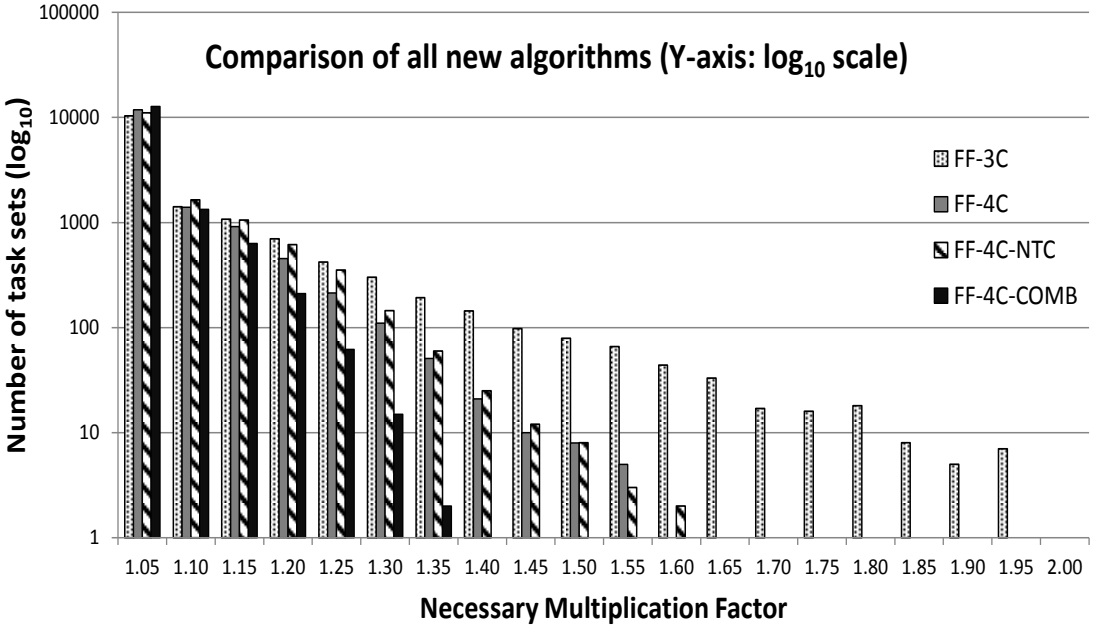


Figure 4.7: Comparison of the necessary multiplication factors for all of our FF- algorithms (if an algorithm has low necessary multiplication factor for many task sets then the algorithm is said to perform well).

have running times in the range of 13500 to 160000 μs (Table 4.8). Hence all of our algorithms

run at least 12000 times faster.

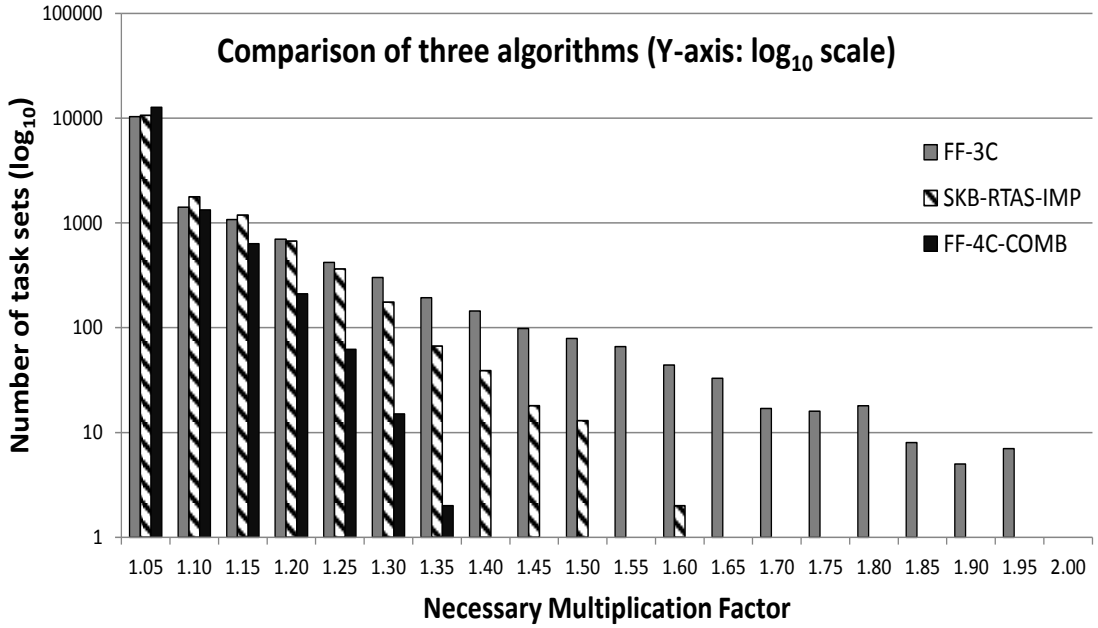


Figure 4.8: Comparison of the necessary multiplication factors for three algorithms — FF-3C, the baselines of all our algorithms, SKB-RTAS-IMP, the better one among the previously known algorithms and FF-4C-COMB, the best of our algorithms.

Multiplication factor	**New (FF-) Algorithms**			
	Measured average execution time			
	FF-3C	**FF-4C**	**FF-4C-NTC**	**FF-4C-COMB**
1.00	0.84	0.73	0.97	1.06
1.25	0.53	0.55	0.54	0.56
1.50	0.49	0.48	0.46	0.48
1.75	0.49	0.46	0.40	0.42
2.00	0.51	0.47	0.43	0.50

Table 4.6: Measured average execution times of our algorithms (in microseconds)

4.3.9.2 Evaluation of our algorithms for different values of α

We evaluated the average-case performance of our algorithms for different values of α. We generated 100000 critically feasible non-migrative task sets. Each critically feasible non-migrative task set had at most 25 tasks and at most 2 processors of each type[4]. We then classified the critically feasible task sets based on the value of α of each task set into ten groups — for a given critically

[4]Since we only evaluate FF- algorithms in this batch of experiments and do not run SKB- algorithms which make use of linear programming solvers thereby taking much longer to output the solution, we could afford to set a higher bound on the number of tasks in each problem instance compared to previous set of experiments.

Multiplication factor	Previous (SKB-) Algorithms			
	Average execution time including CPLEX overhead			
	SKB-RTAS	SKB-RTAS-IMP	SKB-ICPP	SKB-ICPP-IMP
1.00	32477.35	32562.27	394753.66	369170.79
1.25	31665.74	31525.82	393745.52	325010.43
1.50	31747.28	31740.34	381912.81	297383.55
1.75	31749.19	31598.63	337205.23	290102.20
2.00	31752.65	31781.70	291689.45	287692.93

Table 4.7: Measured average execution times of SKB- algorithms (in microseconds) with the CPLEX overhead

Multiplication factor	Previous (SKB-) Algorithms			
	Average execution time excluding CPLEX overhead			
	SKB-RTAS	SKB-RTAS-IMP	SKB-ICPP	SKB-ICPP-IMP
1.00	14263.68	14348.60	164551.87	161689.21
1.25	13452.07	13312.15	163565.96	149459.82
1.50	13533.61	13526.67	161373.08	140211.38
1.75	13535.52	13384.96	151003.87	137302.53
2.00	13538.98	13568.03	137989.63	136490.37

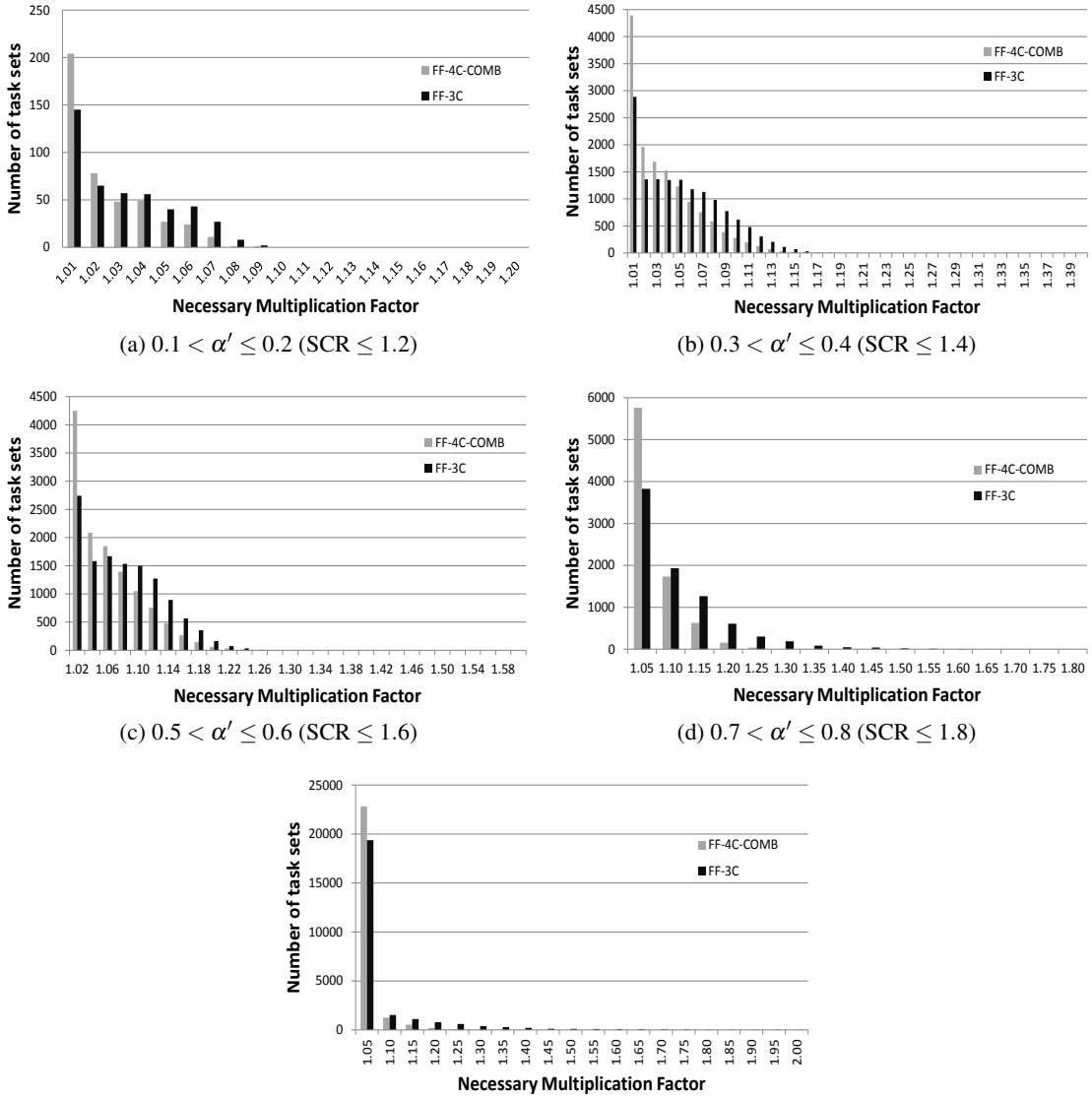
Table 4.8: Measured average execution times of SKB- algorithms (in microseconds) after subtracting the CPLEX overhead

feasible task set, if $\alpha \leq 0.1$ then the task set belongs to the first group, if $0.1 < \alpha \leq 0.2$ then the task set belongs to the second group, ..., and finally if $0.9 < \alpha \leq 1.0$ then the task set belongs to the tenth group. Then, we ran all our FF- algorithms, i.e., FF-3C, FF-4C, FF-4C-NTC and FF-4C-COMB, for the above generated critically feasible non-migrative task sets and observed their necessary multiplication factors. We plotted the histogram of necessary multiplication factors for each of these algorithms for task sets in each of the groups. Since the evaluations in previous subsection have confirmed that FF-4C-COMB performs better compared to all other algorithms and since FF-3C is the baseline of all our algorithms, we only depict these two.

Figure 4.9a to Figure 4.9e shows the performance of FF-3C and FF-4C-COMB algorithms. We only show the results obtained for five cases, i.e., $0.1 < \alpha' \leq 0.2$, $0.3 < \alpha' \leq 0.4$, ..., $0.9 < \alpha' \leq 1.0$. The observations for other cases follow the same trend. As we can see from the graphs, for the vast majority of task sets, the algorithms exhibit much better average-case performance than indicated by their speed competitive ratio, even when we consider the speed competitive ratio as a function of task set parameters.

4.3.10 Summary

In this section, for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors, we presented a low-degree polynomial time-complexity algorithm, FF-3C, and a



(a) $0.1 < \alpha' \leq 0.2$ (SCR ≤ 1.2)

(b) $0.3 < \alpha' \leq 0.4$ (SCR ≤ 1.4)

(c) $0.5 < \alpha' \leq 0.6$ (SCR ≤ 1.6)

(d) $0.7 < \alpha' \leq 0.8$ (SCR ≤ 1.8)

(e) $0.9 < \alpha' \leq 1.0$ (SCR ≤ 2.0)

Figure 4.9: Average-case performance of FF-3C and FF-4C-COMB algorithm in terms of necessary multiplication factors for different values of α' (if an algorithm has low necessary multiplication factor for many task sets then the algorithm performs well)

couple of its variants. These algorithms use bin-packing heuristics (e.g., first-fit) to output the solution. We proved that the speed competitive ratio of each of these algorithms is 2 against an equally powerful non-migrative adversary. We also evaluated their average-case performance. This is done by generating random task sets and converting these task sets into critically feasible non-migrative task sets and then measuring the necessary multiplication factor of the algorithms for each of these critically feasible non-migrative task sets and by measuring their average running times. The proposed FF-3C algorithm (and its variants) is shown to outperform the state-of-the-art algorithms either in terms of (i) the speed competitive ratio or (ii) the time-complexity or (iii) the average-case performance (which is characterized by *necessary multiplication factor* and average

running time of the algorithm in the simulations) or (iv) a combination of these factors.

In the next section, we propose another non-migrative algorithm and prove its speed competitive ratio against a more powerful intra-migrative adversary.

4.4 SA-P algorithm

4.4.1 Introduction

In this section, we present our second *non-migrative* task assignment algorithm, SA-P, an enhanced version of SA algorithm (discussed in Chapter 3), for assigning tasks in τ to individual processors on a two-type platform π. We also prove its speed competitive ratio against a *more powerful* intra-migrative adversary.

Related Work. As discussed at the beginning of this chapter, the problem of non-migrative task assignment on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, LST90, HS76, JP99, WBB13, CSV12, RAB13]. However, most of these approaches provide a performance guarantee in terms of speed competitive ratio against equally powerful non-migrative adversary including our FF-3C algorithm discussed in Section 4.3 — see Table 4.9. Also, most of these solutions (except FF-3C) have a high-degree polynomial time-complexity. Hence, we propose a non-migrative task assignment algorithm of low-degree polynomial time-complexity and prove its speed competitive ratio against a more powerful intra-migrative adversary.

Computing Platform	**Adversary**	**Task Assignment Algorithms**			
	Task migration	**Algorithm**	**Task migration**	**Speed competitive ratio**	**Complexity**
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	non-migrative	FF-3C (Section 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA-P	non-migrative	$1+\alpha \leq 2$	low-degree polynomial

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 4.9: Summary of state-of-the-art task assignment algorithms along with the SA-P algorithm proposed in this section.

Contributions and Significance of the work discussed in this section. We present a non-migrative task assignment algorithm, namely SA-P, of $O(n \log n)$ time-complexity which offers the following guarantee. For a given task set τ and a two-type platform π, if there exists a feasible *intra-migrative* assignment of τ on π then SA-P succeeds in finding a feasible *non-migrative*

assignment of τ but on a platform $\pi^{(1+\alpha)}$ in which every processor is $1+\alpha$ times faster than the corresponding processor in π. In other words, the speed competitive ratio of our non-migrative algorithm, SA-P, is $1+\alpha$ against a more powerful intra-migrative adversary. We also evaluate the average-case performance of our new algorithm by generating task sets randomly and measuring the necessary multiplication factors for each of these task sets.

We believe that the significance of this work is two-fold. First, for the problem of non-migrative task assignment, our algorithm, SA-P, has superior performance compared to state-of-the-art. This can be seen from Table 4.9 since (i) SA-P has the same speed competitive ratio as FF-3C [ARB10, RAB13] and other algorithms in [Bar04b, Bar04c, LST90] but with a stronger adversary and also a better time-complexity, (ii) compared to the algorithms whose speed competitive ratio have been proven against an adversary with a migration model of intra-migrative or greater power [CSV12], SA-P offers the best speed competitive ratio and (iii) compared to PTAS algorithms [HS76, JP99, WBB13] that offer better speed competitive ratios (for lower values of ε) but whose practical significance is severely limited as they incur a very high time-complexity (i.e., exponential in number processors or exponential in $1/\varepsilon$), our algorithm offers a significantly lower (i.e., low-degree polynomial) time-complexity. Second, in our average-case performance evaluations with randomly generated task sets, for the vast majority of task sets, our algorithm requires significantly smaller processor speedup than what is indicated by its theoretical bound.

A global view. The context of the new algorithm, SA-P, can be visualized as shown in Figure 4.10.

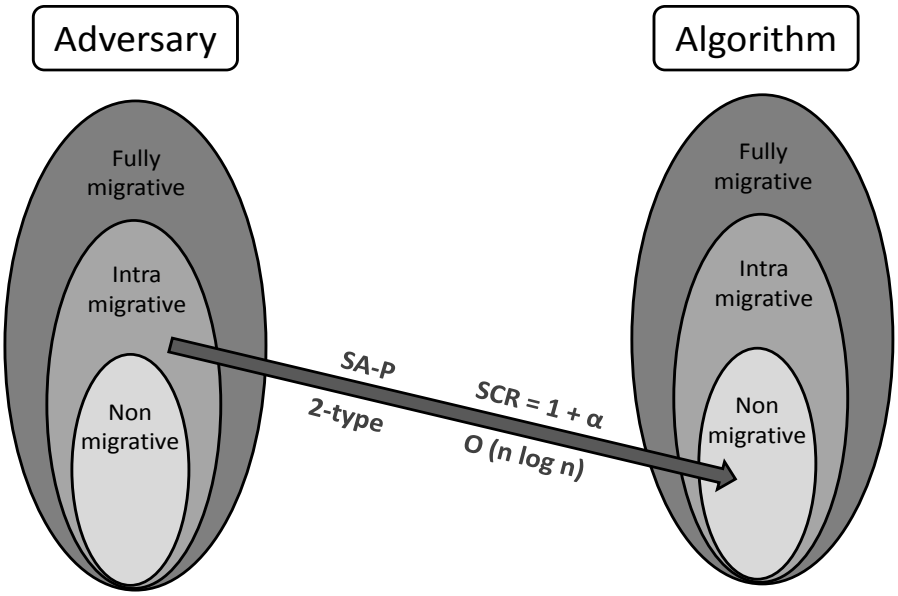


Figure 4.10: A global view of the new algorithm, SA-P, proposed in this section. Here, SCR denotes the "speed competitive ratio", α is a property of the task set — it is the maximum of all the task utilizations that are no greater than one (and hence can take a value in the range $(0,1]$) and n denotes the number of tasks.

Organization of Section 4.4 The rest of the section is organized as follows. Section 4.4.2 briefs the system model. The description of SA-P algorithm is given in Section 4.4.3 and its time-complexity is discussed in Section 4.4.4. Section 4.4.5 proves the speed competitive ratio of SA-P and also shows that this proven bound of SA-P is indeed a tight bound. Section 4.4.6 offers the average-case performance evaluations and Section 4.4.7 concludes.

4.4.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks on a two-type heterogeneous multiprocessor platform comprising m processors, of which m_1 are of type-1 and m_2 are of type-2.

On a two-type platform, we denote by u_i^1 and u_i^2 the utilizations of task τ_i on type-1 and type-2 processors, respectively. A task that cannot be executed upon a certain processor type is modeled by setting its worst-case execution time (and thus its utilization) on that processor type to ∞.
Let α be a real number defined as follows:

$$\alpha \stackrel{\text{def}}{=} \max_{\forall \tau_i \in \tau, t \in \{1,2\}} \left\{ u_i^t : u_i^t \leq 1 \right\}$$

Then it holds that the utilization of any task on any processor type is either no greater than α or is greater than 1, i.e.,

$$\begin{aligned} &\forall \tau_i \in \tau: \quad (u_i^1 \leq \alpha) \ \vee \ (u_i^1 > 1) \quad \text{and} \\ &\forall \tau_i \in \tau: \quad (u_i^2 \leq \alpha) \ \vee \ (u_i^2 > 1) \end{aligned}$$

We assume that all the tasks assigned to a processor are scheduled on this processor using an algorithm that is optimal for the problem of scheduling tasks on a uniprocessor (e.g., EDF [LL73]).

4.4.3 The description of SA-P algorithm

For this algorithm, we consider that the processors are indexed in some order and this indexing is maintained throughout the algorithm. The new algorithm, SA-P, for assigning tasks to processors, works as follows.

1. Assign tasks in τ to processor *types* on π using algorithm SA (discussed in Section 3.6 of Chapter 3 on page 57).

   - SA assigns tasks to only processor types (and not to individual processors); let τ^1 (respectively, τ^2) be the subset of tasks assigned to type-1 (respectively, type-2) processors.
   - SA guarantees that, for an intra-migrative feasible task set, at most one task is fractionally assigned to both processor types; let τ_f be this task and let fraction x_f^1 of τ_f be assigned to type-1 and fraction $x_f^2 = 1 - x_f^1$ be assigned to type-2.

2. Assign tasks from τ^1 (respectively, τ^2) to individual processors of type-1 (respectively, type-2) using next-fit but allowing *splitting* of tasks between consecutive processors (also referred to as "wrap-around" assignment in literature). Assign the fraction, x_f^1 of τ_f, to the last processor (i.e., the m_1^{th} processor) of type-1 and the fraction, x_f^2, to the last processor (i.e., the m_2^{th} processor) of type-2. It is trivial to see that such an assignment ensures following properties:

   - at most $m_1 - 1$ tasks are *split* between processors of type-1 with one task split between each pair of consecutive processors
   - at most $m_2 - 1$ tasks are *split* between processors of type-2 with one task split between each pair of consecutive processors and
   - at most *one* task, τ_f, is fractionally assigned between processors of type-1 and type-2; specifically, τ_f is split between the m_1^{th} processor of type-1 and the m_2^{th} processor of type-2

3. Copy this assignment of tasks onto a faster platform π' (we show in Theorem 15 that a platform in which every processor is $1+\alpha$ times faster than the corresponding processor in π is sufficient).

4. On platform π', assign a task split between processor p and $p+1$ of type-1 to processor p, where $1 \leq p < m_1$; similarly, assign a task split between processor q and $q+1$ of type-2 to processor q, where $1 \leq q < m_2$. Finally, assign the task τ_f to the m_1^{th} processor of type-1 (or to the m_2^{th} processor of type-2).

SA-P is named so because it is the "**P**artitioned" (i.e., non-migrative) version of SA algorithm.

4.4.4 Time-complexity of SA-P algorithm

We now show that the time-complexity of SA-P is a low-degree polynomial function of the number of tasks (n). By inspecting the four steps of SA-P algorithm, we know that:

- In Step 1, tasks are assigned to processor types using SA. The time-complexity of this operation is $O(n \cdot \log n)$ (shown in previous chapter — See Section 3.6.2 on page 58).
- In Step 2, tasks that are assigned to type-1 (respectively, type-2) processors by SA (at most n) are assigned to individual processors of type-1 (respectively, type-2) using "wrap-around" technique. The time-complexity of each of these operations is $O(n)$.
- In Step 3, the assignment (of n tasks) is copied onto a faster platform. The time-complexity of this operation is $O(n)$.
- In Step 4, tasks that are fractionally assigned (at most m) are integrally assigned. The time-complexity of this operation is $O(n)$ since the number of fractionally assigned tasks is upper bounded by n.

Thus, the time-complexity of the algorithm is at most

$$\underbrace{O(n \cdot \log n)}_{\text{Step 1}} + \underbrace{O(n)}_{\text{Step 2}} + \underbrace{O(n)}_{\text{Step 3}} + \underbrace{O(n)}_{\text{Step 4}} = O(n \cdot \log n)$$

4.4.5 Speed competitive ratio of SA-P algorithm

In this section, we derive the speed competitive ratio of SA-P algorithm.

Theorem 15. *If there exists a feasible intra-migrative assignment of a task set τ on a two-type platform π then* SA-P *is guaranteed to find a feasible non-migrative assignment of τ but on a platform $\pi^{(1+\alpha)}$ in which every processor is $1+\alpha \leq 2$ times faster than the corresponding processor in π.*

Proof. We know from Theorem 4 of Chapter 4 (see page 63) that if τ is intra-migrative feasible on π then SA succeeds in returning an assignment of tasks in τ to processor types on π in which at most one task from set L (recall from Expression 3.9 on page 50 that, L is defined as: $\mathrm{L} \stackrel{\text{def}}{=} \left\{\tau_i \in \tau : u_i^1 \leq \alpha \ \wedge \ u_i^2 \leq \alpha\right\}$) is fractionally assigned and the rest are integrally assigned to type-1 and type-2 processors. Hence, we only need to show that, if SA assigns tasks in τ to processor types on π with at most one fractional task then SA-P can assign tasks in τ to individual processors on $\pi^{(1+\alpha)}$ in which the speed of each processor is $1+\alpha$ times that of the corresponding processor in π.

Let us consider the assignment of tasks in τ to processor types in π returned by SA with at most one fractional task. We know that, SA assigns tasks to processor types (and not to individual processors) — let τ^1 (respectively, τ^2) denote the subset of tasks that are assigned to processors of type-1 (respectively, type-2). Let τ_f denote the task that is fractionally assigned to both the processor types — fraction x_f^1 to type-1 and fraction $x_f^2 = 1 - x_f^1$ to type-2. Clearly, $\tau = \tau^1 \cup \tau^2 \cup \left\{\tau_f\right\}$, $\tau^1 \cap \left\{\tau_f\right\} = \emptyset$, $\tau^2 \cap \left\{\tau_f\right\} = \emptyset$ and finally $\tau^1 \cap \tau^2 = \emptyset$. We also know that:

$$\forall \tau_i \in \tau^1 : \ u_i^1 \leq \alpha \qquad \text{and} \tag{4.70}$$

$$\forall \tau_i \in \tau^2 : \ u_i^2 \leq \alpha \qquad \text{and} \tag{4.71}$$

$$\tau_f \in \tau : \ u_f^1 \leq \alpha \ \wedge \ u_f^2 \leq \alpha \tag{4.72}$$

SA-P uses this assignment information and assigns tasks to individual processors (using "wrap-around" technique which allows splitting of tasks between processors of same type) as described earlier in Step 2 of SA-P algorithm. After this step, it must hold that:

$$\forall p \in \pi : U[p] \leq 1 \tag{4.73}$$

where $U[p]$ is the utilization of tasks assigned to processor p. Let $\tau^1_{p_1,p_1+1}$ denote the task split between the p_1^{th} processor and the $(p_1+1)^{th}$ processor of type-1 where $1 \leq p_1 < m_1$. Analogously, let $\tau^2_{p_2,p_2+1}$ denote the task split between the p_2^{th} processor and the $(p_2+1)^{th}$ processor of type-2 where $1 \leq p_2 < m_2$.

On step 3, SA-P copies this assignment onto the faster platform $\pi^{(1+\alpha)}$. Let $u_i^{1'}$ and $u_i^{2'}$ denote the utilizations of task τ_i on platform $\pi^{(1+\alpha)}$. Then, it holds that:

$$\forall \tau_i \in \tau : \frac{u_i^{2'}}{u_i^2} = \frac{u_i^{1'}}{u_i^1} = \frac{1}{1+\alpha} \tag{4.74}$$

Combining Expression (4.73) and (4.74) gives us:

$$\forall p \in \pi^{(1+\alpha)} : U[p] \leq \frac{1}{1+\alpha} \tag{4.75}$$

Also, combining Expressions (4.70)-(4.72) and (4.74), we get:

$$\forall \tau_i \in \tau^1 : \ u_i^{1'} \leq \frac{\alpha}{1+\alpha} \qquad \text{and} \tag{4.76}$$

$$\forall \tau_i \in \tau^2 : \ u_i^{2'} \leq \frac{\alpha}{1+\alpha} \qquad \text{and} \tag{4.77}$$

$$\tau_f \in \tau : \ u_f^{1'} \leq \frac{\alpha}{1+\alpha} \ \wedge \ u_f^{2'} \leq \frac{\alpha}{1+\alpha} \tag{4.78}$$

On step 4, SA-P assigns the split tasks integrally. So, $\forall p_1 \in$ type-1 of $\pi^{(1+\alpha)}$, it moves the fraction of the task $\tau^1_{p_1,p_1+1}$ that is assigned to the $(p_1+1)^{th}$ processor of type-1 to the p_1^{th} processor of type-1. After this re-assignment, it follows from Expressions (4.75) and (4.76) that:

$$\forall p_1 \in \text{type-1 of } \pi^{(1+\alpha)} \ \wedge \ p_1 \neq m_1 : \ U[p_1] \leq 1.0 \tag{4.79}$$

Note that the m_1^{th} processor of type-1 is still utilized at most $\frac{1}{1+\alpha}$ of its capacity as no fraction of any task is moved to this processor in the above step.

Analogously, $\forall p_2 \in$ type-2 of $\pi^{(1+\alpha)}$, SA-P moves the fraction of the task $\tau^2_{p_2,p_2+1}$ that is assigned to the $(p_2+1)^{th}$ processor of type-2 to the p_2^{th} processor of type-2. After this re-assignment, it follows from Expressions (4.75) and (4.77) that:

$$\forall p_2 \in \text{type-2 of } \pi^{(1+\alpha)} \ \wedge \ p_2 \neq m_2 : \ U[p_2] \leq 1.0 \tag{4.80}$$

Once again, since no fraction of any task is moved to the m_2^{th} processor of type-2 in the above step, this processor is still utilized at most $\frac{1}{1+\alpha}$ of its capacity.

Finally, the task τ_f (split between the m_1^{th} processor and the m_2^{th} processor) remains to be integrally assigned. It turns out that this task can be *entirely* assigned to either the m_1^{th} processor of type-1 or the m_2^{th} processor of type-1. Consider the case that it is integrally assigned to the m_1^{th} processor of type-1. Since, this processor is used at most $\frac{1}{1+\alpha}$ of its capacity and since $u_f^{1'} \leq \frac{\alpha}{1+\alpha}$ (see Expression (4.78)), this re-assignment does not allow the used capacity of the m_1^{th} processor to exceed one. Combining this with the fact that the m_2^{th} processor of type-2 is still utilized at most $\frac{1}{1+\alpha}$ of its capacity and with Expression (4.79) and Expression (4.80), we obtain:

$$\forall p \in \pi^{(1+\alpha)} : U[p] \leq 1.0 \tag{4.81}$$

(Analogous reasoning holds for the case when τ_f is integrally assigned to the m_2^{th} processor of type-2.)

Since Expression (4.81) is a necessary and sufficient feasibility condition for task assignment on a uniprocessor [LL73], the non-migrative assignment of τ on $\pi^{(1+\alpha)}$ returned by SA-P is feasible. Hence the proof. □

We now show that the proven speed competitive ratio of SA-P is a tight bound.

Theorem 16 (Speed competitive ratio of SA-P is tight). *The proven speed competitive ratio* $1+\alpha \leq 2$ *of algorithm* SA-P *is a tight bound.*

Proof. In order to show that, the proven speed competitive ratio is tight for SA-P algorithm, it is sufficient to show that, there exists a (feasible intra-migrative) problem instance for which SA-P needs 2 times faster processors to output a feasible non-migrative assignment. We now show that such a problem instance exists.

Consider a problem instance with a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ comprising n tasks and a two-type platform $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprising m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. Also, let $n = m_1 + m_2 + 2$. The task set τ can be partitioned into two subsets, τ^1 of m_1+1 tasks and τ^2 of m_2+1 tasks, such that:

$$\tau^1 \cup \tau^2 = \tau$$
$$\tau^1 \cap \tau^2 = \emptyset$$
$$\forall \tau_i \in \tau^1 : \; u_i^1 = \frac{m_1}{m_1+1} \quad \text{and} \quad u_i^2 = \frac{m_1}{m_1+1} + \frac{1}{(m_1+1)^2}$$
$$\forall \tau_i \in \tau^2 : \; u_i^1 = \frac{m_2}{m_2+1} + \frac{1}{(m_2+1)^2} \quad \text{and} \quad u_i^2 = \frac{m_2}{m_2+1}$$

Now consider SA-P algorithm. Initially, the task set τ is partitioned as follows using Expressions (3.6)–(3.9): $H12 = \emptyset$, $H1 = \emptyset$, $H2 = \emptyset$ and $L = \{\tau_1, \tau_2, \ldots, \tau_n\}$. As a consequence, it holds that $L = \tau^1 \cup \tau^2$. Since all the tasks in the task set are light, SA-P sorts the tasks in non-increasing order of $\frac{u_i^2}{u_i^1}$. From the utilizations of the tasks, it can be seen that, in such a sorted order, all the tasks from τ^1 precede all the tasks from τ^2 (i.e., all the tasks from τ^1 appear before any task from τ^2 in the list). Since $\forall \tau_i \in \tau^1 : u_i^1 = \frac{m_1}{m_1+1}$ and $|\tau^1| = m_1 + 1$, it can be seen that: $\sum_{\tau_i \in \tau^1} u_i^1 = m_1$. Combining this with the fact that, all the tasks of τ^1 appear before any task of τ^2 in the sorted order and the fact that, there are m_1 processors of type-1, it can be seen that SA-P assigns all the tasks of τ^1 to type-1 processors. Analogously, it can be seen that SA-P assigns all the tasks of τ^2 to type-2 processors. Note that, at this stage, tasks have been assigned to processor types and not to individual processors. Now, the tasks need to be assigned to individual processors.

Consider tasks of τ^1 that are assigned to type-1 processors. We know that $|\tau^1| = m_1 + 1$ and there are m_1 processors of type-1 (i.e., one processor less than the number of tasks). Hence, to obtain a non-migrative assignment, SA-P must assign two tasks of τ^1 to at least one processor of type-1. Since, $\forall \tau_i \in \tau^1 : u_i^1 = \frac{m_1}{m_1+1}$, we need to speedup at least one processor of type-1 (which

is the processor to which two tasks from τ^1 will be assigned) to $\frac{2m_1}{m_1+1}$. Analogously, we need to speedup at least one processor of type-2 to $\frac{2m_2}{m_2+1}$. By the definition of speed competitive ratio, we need to speedup every processor by the same factor. Therefore, we need to speedup every processor by a factor of:

$$\max\left\{\frac{2m_1}{m_1+1}, \frac{2m_2}{m_2+1}\right\}$$

Rewriting the above max term gives us: we need to speedup every processor by a factor of:

$$2 \times \max\left\{\frac{m_1}{m_1+1}, \frac{m_2}{m_2+1}\right\}$$

In the above expression, the maximum value that the max term can take is 1 when either m_1 tends to an infinitely large value or when m_2 tends to an infinitely large value. Therefore, we need to speedup every processor by a factor of 2.
Hence the proof. □

Let $\pi(m_1, m_2)$ denote a two-type platform in which $m_1 > 0$ processors are of type-1 and $m_2 > 0$ processors are of type-2. We now state the performance of LP-Algo in terms of additional number of processors.

Corollary 5. *If there exists a feasible intra-migrative assignment of τ on $\pi(m_1, m_2)$ then* SA-P *is guaranteed to obtain a feasible non-migrative assignment of τ on $\pi'(2m_1, 2m_2)$.*

Proof. We know from Theorem 15 that, after executing Step 1 in SA-P, it holds that:

- the utilization of any task that is assigned to processors of type-1 (respectively, type-2) does not exceed α on processors of type-1 (respectively, type-2) — see Expression (4.70) and Expression (4.71) and
- the utilization of the task split between processors of type-1 and type-2 does not exceed α on both processor types — see Expression (4.72)

Also, we know from Theorem 15 that, after executing Step 2 in SA-P, it holds that:

- every processor is utilized at most 100% of its capacity (see Expression (4.73)) and
- at most $m_1 - 1$ (respectively, $m_2 - 1$) tasks are *split* between processors of type-1 (respectively, type-2) with one task split between each pair of consecutive processors and at most 1 task is split between processors of type-1 and type-2

Hence, if such fractional tasks exist then

- the $m_1 - 1$ (respectively, $m_2 - 1$) tasks that are fractionally assigned between processors of type-1 (respectively, type-2) can be integrally assigned to the additional $m_1 - 1$ (respectively, $m_2 - 1$) processors of type-1 (respectively, type-2) in π'.

- the single task that is fractionally assigned between processors of type-1 and type-2 can be integrally assigned to yet another additional processor of either type-1 or type-2 in π' since only $m_1 - 1$ (respectively, $m_2 - 1$) additional processors of type-1 (respectively, type-2) were used in the previous step out of m_1 (respectively, m_2) additional processors.

From earlier observations about the capacity used on each processor and the utilizations of the tasks assigned on each processor type, it is trivial to see that, the above re-assignment satisfies the uniprocessor feasibility test on every processor in π'. Hence the proof. □

4.4.6 Average-case performance evaluation SA-P algorithm

After studying the theoretical bound of SA-P algorithm (i.e., its speed competitive ratio), we evaluate its average-case performance by measuring how well it performs compared to its theoretical bound. We assess its average-case performance by measuring its *necessary multiplication factor* for various randomly generated task sets. For a given task set, we define the necessary multiplication factor of SA-P as the *minimum* amount of extra speed of processors that SA-P needs, so as to succeed in finding a feasible *non-migrative* assignment as compared to an optimal *intra-migrative* task assignment algorithm[5]. For each task set, we evaluate the performance of SA-P algorithm by comparing the *necessary multiplication factor* (computed via simulations) with the *speed competitive ratio* (derived theoretically). In our evaluations, we observed that, for the vast majority of task sets, our algorithm performs significantly better by succeeding in finding a feasible non-migrative assignment with necessary multiplication factor much smaller than the speed competitive ratio. We now discuss these evaluations in detail.

The problem instances (number of tasks, their utilizations and the number of processors of each type) were generated randomly. Each problem instance had at most 25 tasks and at most 3 processors of each type. We generated 100000 task sets, denoted as $\{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(100000)}\}$, which we transformed into "critically feasible intra-migrative task sets" as described in Section 3.8 in Chapter 4 (see page 68).

For each critically feasible task set $\tau^{(k)}_{\text{crit}}$, we measure the *necessary multiplication factor* of algorithm SA-P, denoted by $\text{NMF}^{(k)}_{\text{SA-P}}$. We then compare $\text{NMF}^{(k)}_{\text{SA-P}}$ with the speed competitive ratio denoted by $\text{SCR}^{(k)}_{\text{SA-P}}$[6]. The pseudo-code to compute $\text{NMF}^{(k)}_{\text{SA-P}}$ for every intra-migrative critically feasible task set, $\tau^{(k)}_{\text{crit}}$, is obtained by replacing all the occurrences of SA with SA-P in Algorithm 1 in Section 3.8 (see page 70).

Recall that we want to evaluate the average-case performance of our algorithm by measuring how well it performs compared to its theoretical bound. In this regard, for each critically feasible

[5]Note the subtle difference in this version of the necessary multiplication factor definition compared to the earlier/standard definition. This is due to the fact that, here we are comparing a *non-migrative* algorithm with an optimal *intra-migrative* algorithm.

[6]Note that, as opposed to the generic definition of the speed competitive ratio provided in Section 2.5.1 of Chapter 2 on page 16 which says that the speed competitive ratio is a property of the algorithm alone, the speed competitive ratio of SA-P algorithm which is shown to be $1 + \alpha \leq 2$, is not only a property of the algorithm but also a property of the task set as it depends on the parameter $0 < \alpha \leq 1$ whose value in turn depends on the (utilization values of the tasks in the) task set.

intra-migrative task set, $\tau^{(k)}_{\text{crit}}$, we compute the *performance ratio* $\text{PR}^{(k)}_{\text{SA-P}}$ (in %) of SA-P algorithm as follows (similar to the definition in Section 3.8 of Chapter 3):

$$\text{PR}^{(k)}_{\text{SA-P}} \stackrel{\text{def}}{=} \frac{\text{NMF}^{(k)}_{\text{SA-P}} - 1}{\text{SCR}^{(k)}_{\text{SA-P}} - 1} \times 100 \tag{4.82}$$

Note that both $\text{NMF}^{(k)}_{\text{SA-P}}$ and $\text{SCR}^{(k)}_{\text{SA-P}}$ are numbers that take a value of $1.x$ where the integral part 1 can be seen as the speed of the processors on which an optimal intra-migrative algorithm succeeds to find a feasible intra-migrative task assignment and the fractional part x can be seen as the increase in the speed of processors that algorithm SA requires (compared to the optimal algorithm) in order to find a feasible non-migrative task assignment. Hence, 1 is subtracted from both $\text{NMF}^{(k)}_{\text{SA-P}}$ and $\text{SCR}^{(k)}_{\text{SA-P}}$ in the above expression. The multiplication factor 100 converts the ratio in percentage. *This expression enables us to compare the average-case performance of* SA-P *algorithm for task sets with different values of* α *on a same scale.* For example, for a given critically feasible intra-migrative task set, $\tau^{(k)}_{\text{crit}}$, with $\alpha = 0.1$, if SA-P succeeds in finding a feasible non-migrative task assignment with $\text{NMF}^{(k)}_{\text{SA-P}} = 1.01$ then the value of the above ratio is 10% (since $\text{SCR}^{(k)}_{\text{SA-P}}$ of SA-P for this task set is $1 + \alpha = 1.10$) indicating that SA-P required only 10% faster processors than indicated by the theoretical estimate. Similarly, for a given task set in which $\alpha = 0.2$, if SA-P succeeds in finding a feasible non-migrative task assignment with $\text{NMF}^{(k)}_{\text{SA-P}} = 1.02$ then the value of the above ratio is again 10% (since $\text{SCR}^{(k)}_{\text{SA-P}}$ of SA-P for this task set is $1 + \alpha = 1.20$) indicating that SA-P required only 10% faster processors than indicated by the theoretical estimate.

In general, for a given task set and a given algorithm, the smaller the *performance ratio*, the better the average-case performance of the algorithm. For example, if this ratio takes a value of 100% then it implies that the algorithm is not performing any better than what is indicated by its theoretical bound and if this ratio takes a smaller value, say 10%, then it implies that the algorithm is performing much better (to be precise, 90% better) than its theoretical bound. Hence, an algorithm is said to perform better if this ratio is less for many task sets.

We plot the histogram of the performance ratios of SA-P algorithm in Figure 4.11. As we can see from Figure 4.11, for approximately 70% of the task sets, SA-P succeeded in finding a feasible non-migrative assignment within $(0-10]$% of its theoretical bound, for approximately 20% of the task sets, SA-P succeeded in finding a feasible non-migrative assignment within $(10-20]$% of its theoretical bound, and so on.

To summarize, in our evaluations, for the vast majority of task sets, the SA-P algorithm performed significantly better than indicated by its theoretical bound.

4.4.7 Summary

In this section, for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors, we presented a low-degree polynomial time-complexity algorithm, SA-P. This algorithm is an extension of the intra-migrative algorithm, SA, that was discussed earlier in Chapter 3.

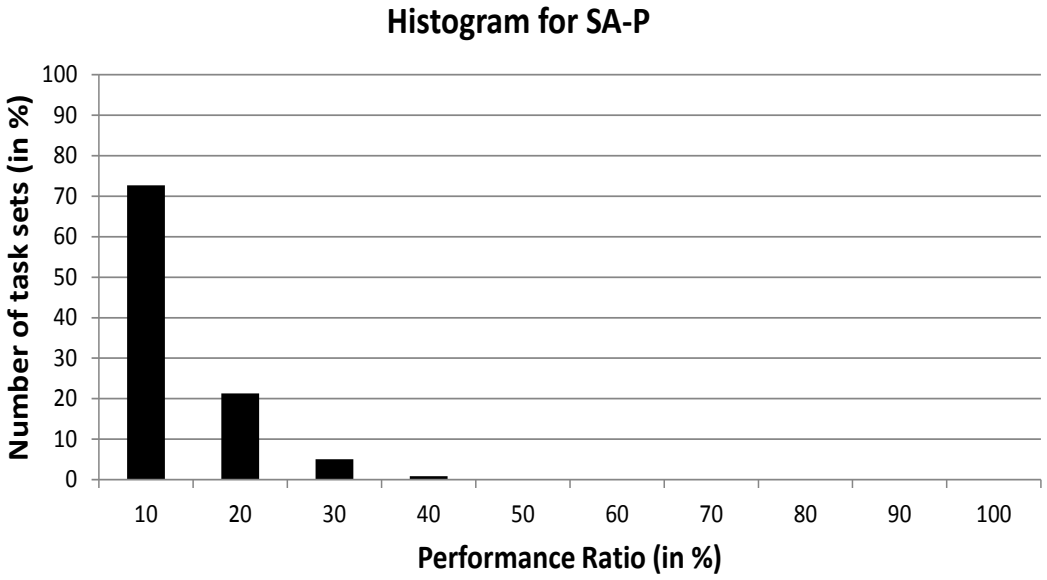


Figure 4.11: Performance of algorithm, SA-P, in terms of *performance ratio* for task sets with different values of α (if an algorithm has low performance ratio for many task sets then the algorithm is said to perform well).

We showed that SA-P algorithm has a time-complexity of $O(n \log n)$ and proved that its speed competitive ratio is $1 + \alpha \leq 2$ against a more powerful intra-migrative adversary, where the parameter $0 < \alpha \leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than one. The proposed algorithm, SA-P, is shown to be better than the state-of-the-art either in terms of the speed competitive ratio or the time-complexity or a combination of both. We also evaluated the average-case performance of SA-P by randomly generating task sets, converting these task sets to critically feasible intra-migrative task sets and then measuring the necessary multiplication factor of SA-P algorithm for each of these critically feasible task sets. In our evaluations, we observed that, for the vast majority of task sets, SA-P algorithm performed significantly better by succeeding in finding a feasible non-migrative assignment with necessary multiplication factor much smaller than the speed competitive ratio.

In the next section, we propose another non-migrative algorithm, LPC, that relies on solving a linear program. We prove its speed competitive ratio against equally powerful non-migrative adversary.

4.5 Cutting plane algorithm

4.5.1 Introduction

In this section, we propose a non-migrative task assignment algorithm, LPC, for assigning tasks in τ to processors in π. This task assignment algorithm is based on solving a Linear Program with Cutting planes. We also prove the speed competitive ratio of LPC algorithm against an equally powerful non-migrative adversary.

Related Work. As discussed in Section 4.1, the problem of assigning tasks to processors on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, LST90, HS76, JP99, WBB13, CSV12, RAB13, RABN12] — summarized in Table 4.10. However, as can be seen in Table 4.10, most of these approaches [Bar04c, Bar04b, LST90] have a speed competitive ratio of 2 or higher [CSV12] (except for PTAS algorithms [HS76, JP99, WBB13] which have a better speed competitive ratio but incur a very high time-complexity) including our FF-3C [RAB13] and SA-P [RABN12] algorithms discussed in Section 4.3 and Section 4.4, respectively.

Computing Platform	**Adversary**	**Task Assignment Algorithms**			
	Task migration	**Algorithm**	**Task migration**	**Speed competitive ratio**	**Complexity**
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	non-migrative	FF-3C (Section 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA-P (Section 4.4)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	non-migrative	LPC	non-migrative	1.5 (and 3 extra processors)	$O(P)$

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 4.10: Summary of state-of-the-art task assignment algorithms along with the LPC algorithm proposed in this section.

Contributions and Significance of the work discussed in this section. We present a non-migrative algorithm, LPC (task assignment based on solving a **L**inear **P**rogram with **C**utting planes), for assigning implicit-deadline sporadic tasks to processors on a two-type heterogeneous

multiprocessor platform, which offers the following guarantee. If there exists a feasible non-migrative assignment of a task set τ on a two-type platform π then, LPC succeeds in finding such a feasible non-migrative assignment of τ but on a platform $\pi^{(1.5x+3p)}$ in which (i) each processor is 1.5 times faster than the corresponding processor in π and (ii) there are 3 additional processors than π.

The significance of this work is two-fold. First, for the problem of non-migrative task assignment, our algorithm, has superior performance compared to state-of-the-art. This can be seen from Table 4.10 since, for systems with large number of processors, our algorithm offers a better speed competitive ratio than all the previous algorithms. This is because (i) for systems with large number of processors, the additional 3 processors that our algorithm requires become negligible and hence its speed competitive ratio tends to $1.5x$ which is better than the algorithms in [Bar04b, Bar04c, LST90, CSV12, RAB13, RABN12] including the FF-3C [RAB13] and SA-P [RABN12] algorithms proposed in previous sections and (ii) compared to PTAS algorithms [HS76, JP99, WBB13] which incur a very high time-complexity (i.e., exponential in processors or exponential in $1/\varepsilon$), our algorithm offers a lower (i.e., polynomial) time-complexity. Second, although task assignment schemes with provably good performance have previously been developed by relaxing an MILP formulation to an LP formulation (e.g., [Bar04c, Bar04b, LST90]) and cutting planes have been used to solve such formulations in different efforts, no work in the past has shown how cutting planes can be used to improve the speed competitive ratio of algorithms for provably good algorithms for assigning real-time tasks to processors. Hence, to the best of our knowledge, this is the first work to do so.

A global view. The context of the new algorithm LPC can be visualized as shown in Figure 4.12.

Organization of Section 4.5 The rest of the section is organized as follows. Section 4.5.2 briefs the system model. Section 4.5.3 discusses task assignment using Integer Linear Program, Linear Program relaxation and cutting planes. Section 4.5.4 presents our new algorithm, LPC, and Section 4.5.5 derives its speed competitive ratio. Finally, Section 4.5.6 concludes.

4.5.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks on a two-type heterogeneous multiprocessor platform $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprising m processors, of which $|P^t(\pi)|$ are of type-t; where $t \in \{1,2\}$. The set of processors of type-t is represented by $P^t(\pi)$. Note that $P^1(\pi) \bigcup P^2(\pi) = \pi$. We assume that an optimal scheduling algorithm (such as EDF [LL73]) is used to schedule the tasks on each processor.

On a two-type platform, the WCET of a task depends on the type of processor on which the task executes. We denote by $C_{i,1}$ and $C_{i,2}$ the WCET of task τ_i when executed on a processor of type-1 and type-2, respectively. The minimum inter-arrival time of task τ_i is denoted by T_i. We denote by $u_{i,1} \stackrel{\text{def}}{=} \frac{C_{i,1}}{T_i}$ and $u_{i,2} \stackrel{\text{def}}{=} \frac{C_{i,2}}{T_i}$ the utilizations of task τ_i on type-1 and type-2 processors,

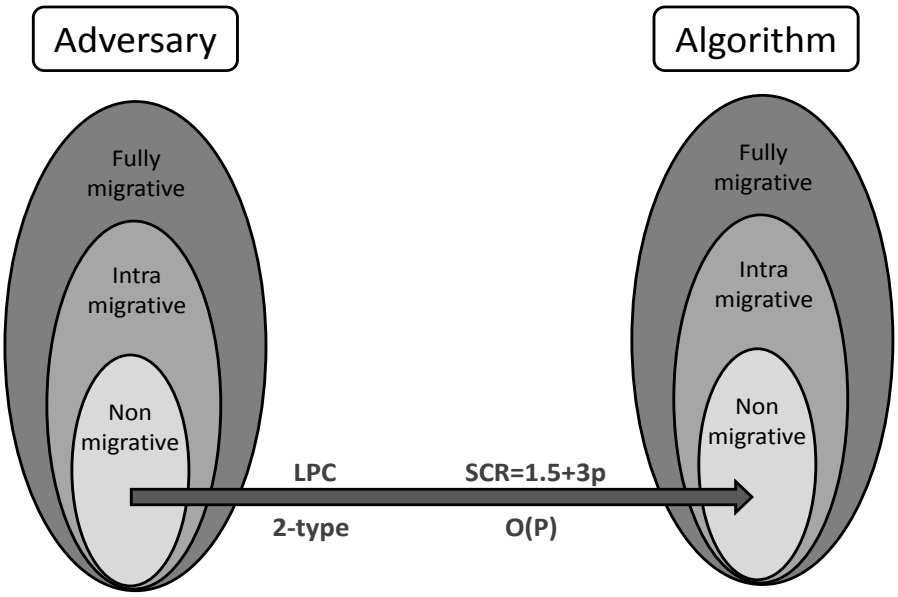


Figure 4.12: A global view of the new algorithm, LPC, proposed in this section. Here, SCR denotes the "speed competitive ratio", the term "+3p" in the speed competitive ratio signifies that, it needs "three extra processors" (in addition to 1.5 times faster processors) and $O(P)$ indicates that the algorithm relies on solving a Linear Program formulation.

respectively. A task that cannot be executed upon a certain processor type is modeled by setting its utilization on that processor type to ∞[7].

We now define a couple of auxiliary functions that are used in the rest of the discussion related to LPC algorithm.

Let aot $\big($ts : `set of tasks`, t : `type`$\big)$ be a function that returns the subset of tasks in set ts such that $u_{i,t} > 1/3$. Similarly, let ah (ts, t) be a function which returns the subset of tasks in set ts such that $u_{i,t} > 1/2$. Intuitively, "aot" means "above one third".

Let `solve`$\big($lp:`linear program`$\big)$ be a function which solves the linear program lp and if this solution is not a vertex optimal solution then it converts this solution into a vertex optimal solution (previous work [Bar04c] did such a transformation). It returns the values assigned to variables and the value of the objective function.

Let mp $\big(\big|P^1(\text{pl})\big|$: #`processors`, $\big|P^2(\text{pl})\big|$: #`processors`, s:`relative speed of processors`, pl:`two-type platform`$\big)$ denote a function that returns a computing platform with $|P^1(\text{pl})|$ (respectively, $|P^2(\text{pl})|$) processors of type-1 (respectively, type-2) that are $s > 0$ times as fast as the corresponding processors of type-1 (respectively, type-2) in computing platform pl. Intuitively, "mp" means "make platform". This function is never called by our algorithm; it is only used in proofs.

[7]Later in the paper, we will solve LPs and MILPs and unfortunately, solvers for these problems typically do not allow coefficients to be ∞. This can be dealt with, however, by assigning utilization of a task on a certain processor to $\max(\big|P^1(\pi)\big|, \big|P^2(\pi)\big|)$. We will see, later in the paper, that this gives the same result as assigning ∞.

Minimize Z_{MILP} subject to the following constraints:

I1.	$\forall p \in P^1(\text{pl}) : \sum_{\tau_i \in \text{ts}} xv_{i,p} \times u_{i,1} \leq Z_{\text{MILP}}$
I2.	$\forall p \in P^2(\text{pl}) : \sum_{\tau_i \in \text{ts}} xv_{i,p} \times u_{i,2} \leq Z_{\text{MILP}}$
I3.	$\forall \tau_i \in \text{ts} : \sum_{p \in P^1(\text{pl})} xv_{i,p} + \sum_{p \in P^2(\text{pl})} xv_{i,p} = 1$
I4.	$\forall \tau_i \in \text{ts}$ and $\forall p \in P^1(\text{pl}) : xv_{i,p}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau_i \in \text{ts}$ and $\forall p \in P^2(\text{pl}) : xv_{i,p}$ is an integer $\in \{0,1\}$

Figure 4.13: $\text{MILP}_{\text{OPT}}(\text{ts},\text{pl})$– MILP formulation for assigning tasks in task set ts to processors in computing platform pl.

Let $\text{sched}(A,\tau,\pi)$ denote a predicate to signify that the task-to-processor assignment returned by algorithm A for tasks in task set τ onto processors in platform π *meets all the deadlines* when the tasks assigned to each processor are scheduled by an optimal uniprocessor scheduling algorithm (such as EDF [LL73]). The term *meets all the deadlines* in this predicate means 'meets deadlines for every possible arrival of tasks that is valid as per the given parameters of τ'. The predicates with $A = \text{OPT}$ imply that there exists a feasible task-to-processor assignment of tasks in τ onto processors in π.

4.5.3 Task assignment, MILP, LP and cutting planes

In this section, we describe how the task assignment problem under consideration can be formulated as MILP. Recall that, we mentioned in Section 4.5.1 that, given a task set and a computing platform, the problem of deciding if a feasible non-migrative task assignment exists is NP-complete in the strong sense. Then it clearly follows that, for any MILP formulation of this problem, deciding if the MILP is feasible is NP-complete in the strong sense as well. Since deciding if our MILP formulation of task assignment is NP-complete, we also discuss how it can be relaxed to LP (because LP can be solved in polynomial time).

Recall that, once the tasks are assigned to processors, we assume that an optimal scheduling algorithm (such as EDF [LL73]) is used on each processor to schedule the respective tasks. From the uniprocessor feasibility test, the following necessary and sufficient condition must hold $\forall t \in \{1,2\}$ in order for the non-migrative task assignment to be feasible:

$$\forall \pi_p \in P^t(\pi) : \quad \sum_{\tau_i \in \tau[\pi_p]} u_{i,t} \;\leq\; 1 \tag{4.83}$$

where $\tau[\pi_p]$ denotes the set of tasks assigned to processor $\pi_p \in \pi$.

The problem of assigning tasks in τ to processors in π can be formulated as MILP using the function $\text{MILP}_{\text{OPT}}(\tau,\pi)$ which returns an MILP formulation as defined by Figure 4.13. In this MILP formulation, every indicator variable, $xv_{i,p}$, indicates the assignment of task τ_i to processor π_p, i.e., $xv_{i,p} = 1$ implies that τ_i is entirely assigned to processor π_p, $xv_{i,p} = 0$ implies that τ_i is not assigned to processor π_p. The variable Z_{MILP} denotes the maximum capacity of any processor that is used and is set as the objective function (to be minimized). If $Z_{\text{MILP}} \leq 1$ then it implies that the sum of utilization of tasks assigned to any processor is less than or equal to the available

	Type-1 (π_1, π_2)	**Type-2** (π_3)
Task	$\mathbf{u_{i,1}}$	$\mathbf{u_{i,2}}$
τ_1	0.51	1.1
τ_2	0.51	1.1
τ_3	0.51	1.1
τ_4	1.1	0.5

Table 4.11: An example task set.

capacity on that processor and hence the assignment is feasible. If $Z_{\text{MILP}} > 1$ then it implies that the condition in Expression (4.83) is violated and hence the task set is *non-migrative infeasible*, i.e., no algorithm will be able to assign the given tasks on the given processors such that all the deadlines are met.

We now illustrate this with an example. Consider a task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ comprising four tasks and a two-type platform $\pi = \{\pi_1, \pi_2, \pi_3\}$ comprising three processors of which π_1 and π_2 are of type-1 and π_3 is of type-2. The utilizations of these tasks on type-1 and type-2 processors are shown in Table 4.11. Observe that this task set is non-migrative infeasible on the given platform.

Solving the MILP formulation, $\text{MILP}_{\text{OPT}}(\tau, \pi)$, for this example outputs $Z_{\text{MILP}} = 1.02$ (corresponding to the assignment in which τ_1 and τ_2 are assigned to π_1 of type-1, τ_3 is assigned to π_2 of type-1 and τ_4 is assigned to π_3 of type-2). Since $Z_{\text{MILP}} > 1$, it rightly indicates that the task set is non-migrative infeasible on the given platform.

As stated earlier (in Section 4.5.1), the problem of finding a feasible task-to-processor assignment on two-type heterogeneous multiprocessors is NP-Complete in the strong sense. Since $\text{MILP}_{\text{OPT}}(\text{ts}, \text{pl})$, shown in Figure 4.13, is the MILP formulation for this problem, it holds that $\text{MILP}_{\text{OPT}}(\text{ts}, \text{pl})$ is NP-Complete in the strong sense as well. It has been shown in the past that, via relaxation of (M)ILP formulation to LP (by allowing a certain number of tasks to be fractionally assigned to processors initially) and certain *rounding tricks* [Pot85] (for integrally assigning the fractionally assigned tasks), polynomial time-complexity can be attained [Bar04b, Bar04c, LST90] at the expense of potentially non-optimal value for the objective function. In another recent development (SA-P algorithm discussed in Section 4.4), it was shown that assigning tasks to *processor types* first and then assigning them to individual *processors* lead to a better performance [RABN12] than [Bar04b, Bar04c, LST90]. Hence, in addition to using *cutting planes* in this work, we also use the above mentioned two tricks, i.e., (i) assigning tasks to processor types first and then assigning them to individual processors and (ii) relaxing MILP to LP and then integrally assigning the fractional tasks.

As discussed in Chapter 3, in intra-migrative task assignment, once tasks are assigned to processor types, we can use an optimal identical multiprocessor scheduling algorithm (e.g., ER-fair [AS00], DP-Fair [LFS+10], U-EDF [NBN+12]) to schedule them on processors of each type. From the feasibility tests of identical multiprocessor scheduling, the following conditions must

hold $\forall t \in \{1,2\}$ in order for intra-migrative task assignment to be feasible:

$$\forall \tau_i \in \tau^t : \quad u_{i,t} \leq 1 \tag{4.84}$$

$$\sum_{\tau_i \in \tau^t} u_{i,t} \leq \left|P^t(\pi)\right| \tag{4.85}$$

where τ^t denotes the tasks assigned to processors of type-t. Given these necessary and sufficient feasibility conditions, we now describe how to obtain a task-to-processor-type assignment of τ on π.

We partition the task set τ into four subsets $\mathrm{H12}(\tau,1)$, $\mathrm{H1}(\tau,1)$, $\mathrm{H2}(\tau,1)$ and $\mathrm{L}(\tau,1)$ as defined below.

$$\mathrm{H12}(\mathrm{ts},\theta) = \{\tau_i \in \mathrm{ts} : u_{i,1} > \theta \wedge u_{i,2} > \theta\} \tag{4.86}$$

$$\mathrm{H1}(\mathrm{ts},\theta) = \{\tau_i \in \mathrm{ts} : u_{i,1} \leq \theta \wedge u_{i,2} > \theta\} \tag{4.87}$$

$$\mathrm{H2}(\mathrm{ts},\theta) = \{\tau_i \in \mathrm{ts} : u_{i,1} > \theta \wedge u_{i,2} \leq \theta\} \tag{4.88}$$

$$\mathrm{L}(\mathrm{ts},\theta) = \{\tau_i \in \mathrm{ts} : u_{i,1} \leq \theta \wedge u_{i,2} \leq \theta\} \tag{4.89}$$

$\mathrm{H12}(\tau,1)$ is the set of tasks whose utilization exceeds one on both processor types. These tasks cannot be assigned to any of the processor types as assigning them in such a manner violates the condition in Expression (4.84). Hence, these tasks make the task set infeasible and thus we assume this set to be empty in the rest of this section. $\mathrm{H1}(\tau,1)$ is the set of tasks that must be assigned to type-1 processors as their utilization on type-2 processors exceeds one and hence assigning them to type-2 processors violates the condition in Expression (4.84). Analogously, $\mathrm{H2}(\tau,1)$ is the set of tasks that must be assigned to type-2 processors as their utilization on type-1 processors exceeds one and hence assigning them to type-1 processors violates the condition in Expression (4.84). Finally, $\mathrm{L}(\tau,1)$ is the set of tasks that can be assigned on either processor type as their utilizations on both processor types do not exceed one. In these definitions, we can intuitively understand the meaning of "H" as "heavy" and "L" as "light" tasks. Now, to obtain an intra-migrative task assignment, do the following.

First, assign the tasks in $\mathrm{H1}(\tau,1)$ to type-1 (respectively, $\mathrm{H2}(\tau,1)$ to type-2) processors. Let U^1 refer to the capacity used on type-1 processors after assigning $\mathrm{H1}(\tau,1)$ tasks, i.e., $U^1 = \sum_{\tau_i \in \mathrm{H1}(\tau,1)} u_{i,1}$. Analogously, let $U^2 = \sum_{\tau_i \in \mathrm{H2}(\tau,1)} u_{i,2}$. If $U^1 > \left|P^1(\pi)\right|$ or $U^2 > \left|P^2(\pi)\right|$ then the task set is intra-migrative infeasible as this violates the condition in Expression (4.85).

Second, solve the formulation, $\mathrm{MILP}_{\mathrm{TYPE}}(\mathrm{L}(\tau,1),\pi,U^1,U^2)$, of Figure 4.14 for assigning tasks in $\mathrm{L}(\tau,1)$. In this formulation, each indicator variable, $yv_{i,t}$ ($t \in \{1,2\}$), indicates the assignment of task τ_i to type-t processors. The variable Z denotes the average used capacity of either type-1 or type-2 processors, whichever is greater, and is set as the objective function to be minimized. If $Z \leq 1$ then a successful intra-migrative assignment is obtained else the task set is intra-migrative infeasible as it violates Expression (4.85).

Recall that, our end goal is to obtain a non-migrative (i.e., task-to-processor) assignment. However, this two-step algorithm where the "Heavy" tasks are assigned first and then the "Light"

Minimize Z subject to the following constraints:

I1.	$U^1 + \sum_{\tau_i \in \text{ts}} yv_{i,1} \times u_{i,1} \leq Z \times \lvert P^1(\text{pl}) \rvert$
I2.	$U^2 + \sum_{\tau_i \in \text{ts}} yv_{i,2} \times u_{i,2} \leq Z \times \lvert P^2(\text{pl}) \rvert$
I3.	$\forall \tau_i \in \text{ts}$: $yv_{i,1} + yv_{i,2} = 1$
I4.	$\forall \tau_i \in \text{ts}$: $yv_{i,1}$ is an **integer** $\in \{0,1\}$
I5.	$\forall \tau_i \in \text{ts}$: $yv_{i,2}$ is an **integer** $\in \{0,1\}$

Figure 4.14: $\text{MILP}_{\text{TYPE}}(\text{ts}, \text{pl}, U^1, U^2)$ — MILP formulation for assigning tasks in ts to processor types in pl.

tasks are assigned by solving the MILP formulation (of Figure 4.14) gives us intra-migrative (i.e., task-to-processor-type) assignment. Hence, we need to convert this task-to-processor-type assignment into a task-to-processor assignment. However, for some task sets, it may be the case that a feasible task-to-processor-type assignment exists but not a feasible task-to-processor assignment. As a result of this, the two-step algorithm can sometimes indicate that a feasible task-to-processor-type assignment exist for those task sets which do not have a feasible task-to-processor assignment. To illustrate this, let us apply this two-step algorithm on our earlier example (see Table 4.11). It first partitions the tasks as follows: $\text{H1}(\tau, 1) = \{\tau_1, \tau_2, \tau_3\}$ and $\text{H2}(\tau, 1) = \{\tau_4\}$. Then, it assigns all the $\text{H1}(\tau, 1)$ tasks to type-1 processors and $\text{H2}(\tau, 1)$ tasks to type-2 processors. As a result, we obtain: $Z = 0.765$ indicating that a feasible task-to-processor-type assignment exists. But, we *cannot* convert this assignment into a feasible task-to-processor assignment (since for this task set there is no feasible task-to-processor assignment as illustrated earlier). To avoid such undesirable scenarios, we use *cuts*.

Observe that, for the example under consideration, the problem with the returned task-to-processor-type assignment (considering the fact that this must be converted to a task-to-processor assignment) is that *three* tasks with utilization 0.51 on type-1 processors are assigned to *two* type-1 processors. We know that such an assignment is task-to-processor infeasible as the number of tasks assigned on type-1 processors with their utilizations greater than 0.5 cannot exceed the number of processors of type-1. Analogous property holds for type-2 processors. Hence, we add these two observations as two separate constraints in the MILP formulation (of Figure 4.14) — these constraints *cut* the feasible region of the optimization problem without losing any solution that is of interest to us (which is a feasible task-to-processor assignment).

Also, as described earlier, solving an MILP formulation is time consuming. However, an LP formulation can be solved in polynomial time though [Kar84]. So, the MILP formulation for assigning tasks in L is relaxed to an LP formulation to be able to solve it in polynomial-time. This relaxed LP formulation along with the two cuts is obtained by the function $\text{TLP}_{\text{CUT}}\big(\text{L}(\tau, 1), \pi, \text{H1}(\tau, 1), \text{H2}(\tau, 1), \text{ah}\big)$ as shown in Figure 4.15. In this LP formulation, variables zv and $yv_{i,t}$ have the same meaning as the corresponding variables, Z and $yv_{i,t}$, in the MILP formulation (of Figure 4.14) and the first three constraints are the same as well. The fourth and fifth constraints represent the cuts that we have added and the sixth and seventh constraints (are *relaxed* versions of fourth and fifth constraints in Figure 4.14) assert that a task can either be *integrally* or *fractionally*

Minimize zv subject to the following constraints:

C1. $\sum_{\tau_i \in \text{ts}} yv_{i,1} \times u_{i,1} + \sum_{\tau_i \in \text{pa1}} u_{i,1} \leq \left|P^1(\text{pl})\right| \times zv$
C2. $\sum_{\tau_i \in \text{ts}} yv_{i,2} \times u_{i,2} + \sum_{\tau_i \in \text{pa2}} u_{i,2} \leq \left|P^2(\text{pl})\right| \times zv$
C3. $\forall \tau_i \in \text{ts} : yv_{i,1} + yv_{i,2} = 1$
C4. $\sum_{\tau_i \in \text{fun}(\text{ts} \cup \text{pa1},\ 1)} yv_{i,1} \leq \left|P^1(\text{pl})\right|$
C5. $\sum_{\tau_i \in \text{fun}(\text{ts} \cup \text{pa2},\ 2)} yv_{i,2} \leq \left|P^2(\text{pl})\right|$
C6. $\forall \tau_i \in \text{ts}$: $yv_{i,1}$ is a real number ≥ 0
C7. $\forall \tau_i \in \text{ts}$: $yv_{i,2}$ is a real number ≥ 0

Figure 4.15: $\text{TLP}_{\text{CUT}}(\text{ts}, \text{pl}, \text{pa1}, \text{pa2}, \text{fun})$ — LP formulation with *cuts* for assigning tasks in ts to processor types in pl.

assigned to processor types.

The proposed algorithm which is discussed in the next section uses this Linear Program formulation (which is based on *cuts*).

4.5.4 The new algorithm: LPC

The pseudo-code for the proposed algorithm, LPC, is listed in Algorithm 8. LPC uses a variant of First-Fit bin-packing scheme where heavy tasks are assigned first — pseudo-code for this First-Fit bin-packing variant, FF_{hf}, is shown in Algorithm 9.

The algorithm, LPC, for assigning tasks in τ to processors in π works as follows.

1. Partition the task set τ into $\text{H12}(\tau, 2/3)$, $\text{H1}(\tau, 2/3)$, $\text{H2}(\tau, 2/3)$ and $\text{L}(\tau, 2/3)$ as shown in Expression (4.86)–(4.89).

2. Set aside three processors of type-1 and let rp denote this set of three processors. Then solve the LP formulation, $\text{TLP}_{\text{CUT}}\big(\text{L}(\tau, 2/3), \pi', \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}\big)$, for assigning tasks in $\text{L}(\tau, 2/3)$ to processor types, where $\pi' = \pi \setminus \text{rp}$. In the solution returned by the LP solver, let (i) L1 and L2 denote the subset of tasks in $\text{L}(\tau, 2/3)$ that are integrally assigned to type-1 and type-2 processors, respectively and (ii) τ^F denote the subset of tasks in $\text{L}(\tau, 2/3)$ that are fractionally assigned between processors of type-1 and type-2 (we later show that $|\tau^F| \leq 3$).

3. Assign the tasks in $\text{H1}(\tau, 2/3) \cup \text{L1}$ to type-1 processors and $\text{H2}(\tau, 2/3) \cup \text{L2}$ to type-2 processors using the First-Fit bin-packing variant, FF_{hf}.

4. Assign each of the (at most three) tasks in τ^F to a unique processor in rp.

Informally, choosing $\theta = 2/3$ for partitioning the task set τ into four subsets (Step 1) and then assigning the tasks in $\text{H1}(\tau, 2/3)$ and $\text{H2}(\tau, 2/3)$ to type-1 and type-2 processors, respectively (Step 3), facilitates in creating an algorithm with the desired speed competitive ratio. Since we will compare the performance of our new algorithm versus every other algorithm that uses processors of at most $2/3$ the speed, it ensures that each of the tasks in $\text{H1}(\tau, 2/3)$ and $\text{H2}(\tau, 2/3)$ is assigned to the same corresponding processor type as under every other successful assignment algorithm.

Algorithm 8: LPC: The non-migrative task assignment algorithm for two-type heterogeneous multiprocessors based on linear program with *cuts*.

Input : A task set τ and a two-type platform π
Output: An assignment of tasks to processors indicated by matrix X
`// Let Y denote a matrix in which the algorithm stores the information about the assignment of tasks to processor types`
Set each element in X and Y to zero
Select any subset of three processors of type-1 from π and let rp denote this set of processors
Let π' denote a platform $\pi \setminus \text{rp}$
Partition the task set τ into subsets $\text{H12}(\tau,2/3)$, $\text{H1}(\tau,2/3)$, $\text{H2}(\tau,2/3)$ and $\text{L}(\tau,2/3)$ as shown in Expressions (4.86)–(4.89).
if $(\text{H12}(\tau,2/3) = \emptyset)$ **then**
  foreach $(\tau_i \in \text{H1}(\tau,2/3))$ **do** $y_{i,1} := 1$;
  foreach $(\tau_i \in \text{H2}(\tau,2/3))$ **do** $y_{i,2} := 1$;
  $\langle YV, zv, f \rangle :=$ solve$(\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3), \pi', \text{H1}(\tau,2/3), \text{H2}(\tau,2/3), \text{aot}))$
  if (f = feasible) **then**
    foreach $(\tau_i \in \text{L}(\tau,2/3))$ **do** $y_{i,1} := yv_{i,1}$ **end** ;
    foreach $(\tau_i \in \text{L}(\tau,2/3))$ **do** $y_{i,2} := yv_{i,2}$ **end** ;
    $z := zv$
    if $(z \leq 2/3)$ **then**
      $\tau^{\text{F}} := \{\tau_i \in \text{L}(\tau,2/3) : y_{i,1} > 0 \wedge y_{i,2} > 0\}$
      $\tau^{\text{A}} := \text{FF}_{\text{hf}}(\tau^{\text{F}}, \pi, \text{rp}, 1)$
      if $(\tau^{\text{A}} = \tau^{\text{F}})$ **then**
        $\text{L1} := \{\tau_i \in \text{L}(\tau,2/3) : y_{i,1} = 1\}$; $\text{L2} := \{\tau_i \in \text{L}(\tau,2/3) : y_{i,2} = 1\}$
        $\tau^1 := \text{L1} \cup \text{H1}(\tau,2/3)$; $\tau^2 := \text{L2} \cup \text{H2}(\tau,2/3)$
        if $(\text{aot}(\tau^1,1) \leq |\text{P}^1(\pi)| - |\text{rp}|)$ **then**
          if $(\text{aot}(\tau^2,2) \leq |\text{P}^2(\pi)|)$ **then**
            $\tau^{\text{A1}} := \text{FF}_{\text{hf}}(\tau^1, \pi, P^1(\pi) \setminus \text{rp}, 1)$
            $\tau^{\text{A2}} := \text{FF}_{\text{hf}}(\tau^2, \pi, P^2(\pi), 2)$
            if $(\tau^{\text{A1}} = \tau^1)$ **then**
              if $(\tau^{\text{A2}} = \tau^2)$ **then**
                declare SUCCESS
              else
                declare FAILURE
              end
            else
              declare FAILURE
            end
          else
            declare FAILURE
          end
        else
          declare FAILURE
        end
      else
        declare FAILURE
      end
    else
      declare FAILURE
    end
  else
    declare FAILURE
  end
else
  declare FAILURE
end

Algorithm 9: FF_{hf}: A variant of First-Fit bin-packing (in which **h**eavy utilization tasks are assigned **f**irst)

Input : ts : a set of tasks, pl : a two-type platform, *ps* : a set of processors to assign the tasks in ts to, t : type-id

Output: A task-to-processor assignment of tasks in ts to processors in *ps* of type t of platform pl

```
// Assumption:  |aot(ts,t)| ≤ |ps|
// This algorithm modifies the variable X in the task assignment
   algorithm, LPC.
// pso is a local variable, a tuple that stores the set of
   processors in ps in a certain order.  The function first(pso)
   returns the first processor in pso and the function
   next(pso,p) returns NULL if p is the last processor in pso,
   otherwise it returns the processor after p in pso.  tso is a
   local variable, a tuple that stores the set of tasks in ts in
   a certain order.
```

1 at := ∅ // set 'assigned tasks' to empty
2 Order the processors in the set *ps* in some order and assign it to the tuple *pso*
3 p := `first(pso)`
4 Order the tasks in the set aot(ts,t) in some order and assign it to the tuple *tso*
5 **foreach** ($\tau_i \in$ tso), *in order* **do**
6 $\quad x_{i,p} := 1$
7 $\quad$ at := at$\cup\{\tau_i\}$
8 $\quad p$:= `next(pso,p)`
$\quad$ `// We will not run out of processors here because of the assumption` $|\text{aot}(\text{ts},t)| \leq |ps|$`. Also, note that the main algorithm checks to ensure that when we call this algorithm, this assumption is true`
9 **end**
10 Order the tasks in the set ts $\setminus$ aot(ts,t) in some order and assign it to the tuple *tso*
11 **foreach** ($\tau_i \in$ tso), *in order* **do**
12 $\quad p$:= `first(pso)`
13 $\quad$ **while** (τ_i *is not in* at) **do**
14 $\quad\quad$ **if** $\left(\sum_{\tau_j \in \text{at}} x_{j,p} \times u_{j,t} + u_{i,t} \leq 1\right)$ **then**
15 $\quad\quad\quad x_{i,p} := 1$
16 $\quad\quad\quad$ at := at$\cup\{\tau_i\}$
17 $\quad\quad$ **else**
18 $\quad\quad\quad$ **if** (`next(pso,p)`=*NULL*) **then**
19 $\quad\quad\quad\quad$ return at
20 $\quad\quad\quad$ **else**
21 $\quad\quad\quad\quad p$:= `next(pso,p)`
22 $\quad\quad\quad$ **end**
23 $\quad\quad$ **end**
24 $\quad$ **end**
25 **end**
26 return at

Also, using the function aot while formulating the LP formulation (Step 2) serves the same purpose of achieving the desired speed competitive ratio — details are provided later in the proofs.

4.5.5 The speed competitive ratio of LPC algorithm

In this section, we show that if there exists a feasible non-migrative assignment of a task set τ on a two-type platform π, then LPC succeeds in finding such a feasible non-migrative assignment as well for τ but on a platform $\pi^{(1.5x+3p)}$ in which each processor is 1.5 times faster than the corresponding processor in π and in addition it has 3 extra processors than π. We prove this via a series of intermediate results.

Let $Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})}$ denote the value of the objective function obtained by solving the LP formulation, $\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})$.

Lemma 20. *Consider a task set τ and a two-type platform π. Let τ' be defined as:*

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \ \wedge \ u'_{i,2} = u_{i,2} \times 3/2$$

It then holds that:

$$\text{sched}(\text{OPT}, \tau', \pi) \Rightarrow Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$$

Proof. We assume that the left-hand side predicate is true and show that the right-hand side predicate is true as well. Since the predicate $\text{sched}(\text{OPT}, \tau', \pi)$ is true, it holds that:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\forall p \in P^1(\pi) : \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,1} \leq zv$
I2.	$\forall p \in P^2(\pi) : \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,2} \leq zv$
I3.	$\forall \tau'_i \in \tau' : \sum_{p \in P^1(\pi)} xv_{i,p} + \sum_{p \in P^2(\pi)} xv_{i,p} = 1$
I4.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^1(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^2(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$

We can observe that there can be at most $\left|P^1(\pi)\right|$ tasks (respectively, at most $\left|P^2(\pi)\right|$ tasks), $\tau'_i \in \tau'$, with $u'_{i,1} > 1/2$ (respectively, $u'_{i,2} > 1/2$) that are assigned to type-1 processors (respectively, type-2 processors). This gives us:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\forall p \in P^1(\pi) : \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,1} \leq zv$
I2.	$\forall p \in P^2(\pi) : \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,2} \leq zv$
I3.	$\forall \tau'_i \in \tau' : \sum_{p \in P^1(\pi)} xv_{i,p} + \sum_{p \in P^2(\pi)} xv_{i,p} = 1$
I4.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^1(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^2(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$
I6.	$\sum_{p \in P^1(\pi)} \sum_{\tau'_i \in \tau' : u'_{i,1} > 1/2} xv_{i,p} \leq \left\lvert P^1(\pi) \right\rvert$
I7.	$\sum_{p \in P^2(\pi)} \sum_{\tau'_i \in \tau' : u'_{i,2} > 1/2} xv_{i,p} \leq \left\lvert P^2(\pi) \right\rvert$

Let us rewrite the two last constraints by changing the order of summation on the left-hand side. Also, for each of the first two constraints, let us add up the constraints. This may change the feasible region but the feasible region increases in the sense that each point that was feasible before is still feasible. This gives us:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\sum_{p \in P^1(\pi)} \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
I2.	$\sum_{p \in P^2(\pi)} \sum_{\tau'_i \in \tau'} xv_{i,p} \times u'_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
I3.	$\forall \tau'_i \in \tau' : \sum_{p \in P^1(\pi)} xv_{i,p} + \sum_{p \in P^2(\pi)} xv_{i,p} = 1$
I4.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^1(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau'_i \in \tau'$ and $\forall p \in P^2(\pi) : xv_{i,p}$ is an integer $\in \{0,1\}$
I6.	$\sum_{\tau'_i \in \tau' : u'_{i,1} > 1/2} \sum_{p \in P^1(\pi)} xv_{i,p} \leq \left\lvert P^1(\pi) \right\rvert$
I7.	$\sum_{\tau'_i \in \tau' : u'_{i,2} > 1/2} \sum_{p \in P^2(\pi)} xv_{i,p} \leq \left\lvert P^2(\pi) \right\rvert$

Once again, let us reorder the summation on the left-hand side of the first two constraints. Also, extracting the utilization terms outside one of the summations in the first two constraints and then replacing (i) $\sum_{p \in P^1(\pi)} xv_{i,p}$ with $yv_{i,1}$ and (ii) $\sum_{p \in P^2(\pi)} xv_{i,p}$ with $yv_{i,2}$ gives us:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\sum_{\tau'_i \in \tau'} u'_{i,1} \times yv_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
I2.	$\sum_{\tau'_i \in \tau'} u'_{i,2} \times yv_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
I3.	$\forall \tau'_i \in \tau' : yv_{i,1} + yv_{i,2} = 1$
I4.	$\forall \tau'_i \in \tau'$: $yv_{i,1}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau'_i \in \tau'$: $yv_{i,2}$ is an integer $\in \{0,1\}$
I6.	$\sum_{\tau'_i \in \tau' : u'_{i,1} > 1/2} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
I7.	$\sum_{\tau'_i \in \tau' : u'_{i,2} > 1/2} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$

We partition the task set τ' into H12(τ',1), H1(τ',1), H2(τ',1) and L(τ',1) as shown in Expressions (4.86)–(4.89). Rewriting the previous formulation based on these partitions gives us:

The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\sum_{\tau'_i \in \tau'} u'_{i,1} \times yv_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
I2.	$\sum_{\tau'_i \in \tau'} u'_{i,2} \times yv_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
I3.	$\forall \tau'_i \in \mathrm{H12}(\tau', 1) : yv_{i,1} + yv_{i,2} = 1$
I4.	$\forall \tau'_i \in \mathrm{H1}(\tau', 1) : yv_{i,1} + yv_{i,2} = 1$
I5.	$\forall \tau'_i \in \mathrm{H2}(\tau', 1) : yv_{i,1} + yv_{i,2} = 1$
I6.	$\forall \tau'_i \in \mathrm{L}(\tau', 1) : yv_{i,1} + yv_{i,2} = 1$
I7.	$\forall \tau'_i \in \mathrm{H12}(\tau', 1) : yv_{i,1}, yv_{i,2}$ are integers $\in \{0,1\}$
I8.	$\forall \tau'_i \in \mathrm{H1}(\tau', 1) : yv_{i,1}, yv_{i,2}$ are integers $\in \{0,1\}$
I9.	$\forall \tau'_i \in \mathrm{H2}(\tau', 1) : yv_{i,1}, yv_{i,2}$ are integers $\in \{0,1\}$
I10.	$\forall \tau'_i \in \mathrm{L}(\tau', 1) : yv_{i,1}, yv_{i,2}$ are integers $\in \{0,1\}$
I11.	$\sum_{\tau'_i \in \tau' : u'_{i,1} > 1/2} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
I12.	$\sum_{\tau'_i \in \tau' : u'_{i,2} > 1/2} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$

Since $zv \leq 1$, it follows that, $\forall \tau_i \in \mathrm{H1}(\tau', 1)$: $yv_{i,1} = 1$. Analogously, it follows that, $\forall \tau_i \in \mathrm{H2}(\tau', 1)$: $yv_{i,2} = 1$. Also, because $zv \leq 1$, the set $\mathrm{H12}(\tau', 1)$ must be empty. These observations and rearrangement of the terms in the first two constraints gives us:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize zv subject to the following constraints:

I1.	$\sum_{\tau'_i \in \mathrm{L}(\tau', 1)} u'_{i,1} \times yv_{i,1} + \sum_{\tau'_i \in \mathrm{H1}(\tau', 1)} u'_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
I2.	$\sum_{\tau'_i \in \mathrm{L}(\tau', 1)} u'_{i,2} \times yv_{i,2} + \sum_{\tau'_i \in \mathrm{H2}(\tau', 1)} u'_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
I3.	$\forall \tau'_i \in \mathrm{L}(\tau', 1) : yv_{i,1} + yv_{i,2} = 1$
I4.	$\forall \tau'_i \in \mathrm{L}(\tau', 1)$: $yv_{i,1}$ is an integer $\in \{0,1\}$
I5.	$\forall \tau'_i \in \mathrm{L}(\tau', 1)$: $yv_{i,2}$ is an integer $\in \{0,1\}$
I6.	$\sum_{\tau'_i \in \mathrm{H1}(\tau', 1) \cup \mathrm{L}(\tau', 1) : u'_{i,1} > 1/2} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
I7.	$\sum_{\tau'_i \in \mathrm{H2}(\tau', 1) \cup \mathrm{L}(\tau', 1) : u'_{i,2} > 1/2} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$

We can observe that if a task $\tau'_i \in \mathrm{H1}(\tau', 1)$ then it follows that the corresponding task $\tau_i \in \mathrm{H1}(\tau, 2/3)$. Analogously for tasks in $\mathrm{H2}(\tau, 2/3)$, $\mathrm{H12}(\tau, 2/3)$ and $\mathrm{L}(\tau, 2/3)$. Also, doing the following substitution: $u'_{i,1} = u_{i,1} \times \frac{3}{2}$ and $u'_{i,2} = u_{i,2} \times \frac{3}{2}$ and then rewriting the objective function and the first two and the last two constraints gives us:
The value of the objective function for an optimal solution of the following optimization problem is ≤ 1:

Minimize $(\frac{2}{3} \times zv) \times \frac{3}{2}$ subject to the following constraints:

I1. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,1} \times yv_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq \frac{2}{3} \times zv \times \left|P^1(\pi)\right|$
I2. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,2} \times yv_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq \frac{2}{3} \times zv \times \left|P^2(\pi)\right|$
I3. $\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
I4. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is an integer $\in \{0,1\}$
I5. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is an integer $\in \{0,1\}$
I6. $\sum_{\tau_i \in \mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,1} > 1/3} yv_{i,1} \leq \left|P^1(\pi)\right|$
I7. $\sum_{\tau_i \in \mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,2} > 1/3} yv_{i,2} \leq \left|P^2(\pi)\right|$

Substituting $\frac{2}{3} \times zv$ by zt and since $zt \times \frac{3}{2} \leq 1$ is same as $zt \leq \frac{2}{3}$, we obtain:
The value of the objective function for an optimal solution of the following optimization problem is $\leq 2/3$:

Minimize zt subject to the following constraints:

I1. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,1} \times yv_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq zt \times \left|P^1(\pi)\right|$
I2. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,2} \times yv_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq zt \times \left|P^2(\pi)\right|$
I3. $\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
I4. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is an integer $\in \{0,1\}$
I5. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is an integer $\in \{0,1\}$
I6. $\sum_{\tau_i \in \mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,1} > 1/3} yv_{i,1} \leq \left|P^1(\pi)\right|$
I7. $\sum_{\tau_i \in \mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,2} > 1/3} yv_{i,2} \leq \left|P^2(\pi)\right|$

Substituting zt by zv gives us:
The value of the objective function for an optimal solution of the following optimization problem is $\leq 2/3$:

Minimize zv subject to the following constraints:

I1. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,1} \times yv_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq zv \times \left|P^1(\pi)\right|$
I2. $\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,2} \times yv_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq zv \times \left|P^2(\pi)\right|$
I3. $\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
I4. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is an integer $\in \{0,1\}$
I5. $\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is an integer $\in \{0,1\}$
I6. $\sum_{\tau_i \in \mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,1} > 1/3} yv_{i,1} \leq \left|P^1(\pi)\right|$
I7. $\sum_{\tau_i \in \mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3): u_{i,2} > 1/3} yv_{i,2} \leq \left|P^2(\pi)\right|$

Note that the optimization problem above is an MILP. We can relax the constraint on integrality of $yv_{i,1}$ and $yv_{i,2}$. This gives us a non-decreasing feasible region and hence the value of the objective function at an optimal solution is non-increasing. This gives us:
The value of the objective function for an optimal solution of the following optimization problem is $\leq 2/3$:

Minimize zv subject to the following constraints:

C1.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,1} \times yv_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
C2.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,2} \times yv_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
C3.	$\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
C4.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is a real number in $[0,1]$
C5.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is a real number in $[0,1]$
C6.	$\sum_{\tau_i \in \mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3):u_{i,1}>1/3} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
C7.	$\sum_{\tau_i \in \mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3):u_{i,2}>1/3} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$

Because of $yv_{i,1} + yv_{i,2} = 1$, it follows that, $yv_{i,1} \leq 1$ and $yv_{i,2} \leq 1$. Hence, it is unnecessary to state that $yv_{i,1}$ and $yv_{i,2}$ are real numbers in the range $[0,1]$. Therefore, instead of mentioning this range in the constraint, only mentioning that these variables have to be greater than or equal to zero, does not impact the feasible region of the above problem and also does not impact the value of the objective function at an optimal solution. This gives us:
The value of the objective function for an optimal solution of the following optimization problem is $\leq 2/3$:

Minimize zv subject to the following constraints:

C1.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,1} \times yv_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq zv \times \left\lvert P^1(\pi) \right\rvert$
C2.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} u_{i,2} \times yv_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq zv \times \left\lvert P^2(\pi) \right\rvert$
C3.	$\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
C4.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is a real number ≥ 0
C5.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is a real number ≥ 0
C6.	$\sum_{\tau_i \in \mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3):u_{i,1}>1/3} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
C7.	$\sum_{\tau_i \in \mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3):u_{i,2}>1/3} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$

Consider the following call to the $\mathrm{TLP}_{\mathrm{CUT}}$ function, i.e., $\mathrm{TLP}_{\mathrm{CUT}}\big(\mathrm{L}(\tau,2/3),\ \pi,\ \mathrm{H1}(\tau,2/3),\ \mathrm{H2}(\tau,2/3),\ \mathrm{aot}\big)$. This gives:
$\mathrm{TLP}_{\mathrm{CUT}}(\mathrm{L}(\tau,2/3), \pi, \mathrm{H1}(\tau,2/3), \mathrm{H2}(\tau,2/3), \mathrm{aot})$ =

Minimize zv subject to the following constraints:

C1.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} yv_{i,1} \times u_{i,1} + \sum_{\tau_i \in \mathrm{H1}(\tau,2/3)} u_{i,1} \leq \left\lvert P^1(\pi) \right\rvert \times zv$
C2.	$\sum_{\tau_i \in \mathrm{L}(\tau,2/3)} yv_{i,2} \times u_{i,2} + \sum_{\tau_i \in \mathrm{H2}(\tau,2/3)} u_{i,2} \leq \lvert P^2(\pi) \rvert \times zv$
C3.	$\forall \tau_i \in \mathrm{L}(\tau,2/3) : yv_{i,1} + yv_{i,2} = 1$
C4.	$\sum_{\tau_i \in \mathrm{aot}(\mathrm{H1}(\tau,2/3) \cup \mathrm{L}(\tau,2/3))} yv_{i,1} \leq \left\lvert P^1(\pi) \right\rvert$
C5.	$\sum_{\tau_i \in \mathrm{aot}(\mathrm{H2}(\tau,2/3) \cup \mathrm{L}(\tau,2/3))} yv_{i,2} \leq \left\lvert P^2(\pi) \right\rvert$
C6.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,1}$ is a real number ≥ 0
C7.	$\forall \tau_i \in \mathrm{L}(\tau,2/3)$: $yv_{i,2}$ is a real number ≥ 0

Note that the earlier optimization problem is same as $\mathrm{TLP}_{\mathrm{CUT}}(\mathrm{L}(\tau),\ \pi,\ \mathrm{H1}(\tau),\ \mathrm{H2}(\tau),\ \mathrm{aot})$. This gives us:
The value of the objective function for an optimal solution of the following optimization problem

is $\leq 2/3$:
$\text{TLP}_{\text{CUT}}(\text{L}(\tau, 2/3), \pi, \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot})$.

This gives us: $Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$.

Hence, we have shown that:

$$\text{sched}(\text{OPT}, \tau', \pi) \Rightarrow Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$$

This states the lemma. □

Corollary 6. *Consider a task set τ and a two-type platform π. Let τ' be defined as:*

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \ \wedge \ u'_{i,2} = u_{i,2} \times 3/2$$

It then holds that:

$$\text{sched}(\text{OPT}, \tau', \pi) \Rightarrow \text{TLP}_{\text{CUT}}(\text{L}(\tau, 2/3), \pi, \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}) \textit{ is feasible}.$$

Proof. This follows from Lemma 20. □

The following lemma follows from a well-known result about vertex solutions in Linear Programming [Bar04c].

Lemma 21. *For each input τ and π to Algorithm 8 it holds that: When line 9 has finished execution, if the optimization problem is feasible then it holds that there are at most three tasks in τ that are fractionally assigned between processor types (referred to as fractionally type-assigned).*

Proof. Suppose that the claim is false. Then it holds that there is a τ and π such that if they are input to the function `solve(`TLP_{CUT}`)` then `solve(`TLP_{CUT}`)` outputs a solution in which four or more tasks are fractionally type-assigned. Then it must have been that in the solution output by `solve(`TLP_{CUT}`)`, there were four or more tasks τ_i for which it holds that $0 < yv_{i,1} < 1$ and $0 < yv_{i,2} < 1$. Considering TLP_{CUT}, we can observe that it has $2|\text{L}|+1$ variables and 2L non-negativity constraints and $|\text{L}|+4$ other constraints. Hence, in the vertex solution, there are at most $|\text{L}|+4$ non-zero variables [Bar04c]. Let us explore two cases:
Case 1: $zv = 0$**.** If this is the case then the vertex optimal solution produced by `solve(`TLP_{CUT}`)` has all type-integral assignments and hence this contradicts the claim that there were four of more fractionally type-assigned tasks.
Case 2: $zv > 0$**.** Since $zv > 0$, it follows that, there are at most $|\text{L}|+3$ non-zero $yv_{i,t}$ values. Let Q denote the number of tasks that are fractionally type-assigned. From our assumption that the lemma is false, it follows that $Q \geq 4$. The number of non-zero values of Y is exactly $Q \times 2 + (|\text{L}| - Q)$ because each fractionally type-assigned task provides us with two non-zero variables in Y and each integrally type-assigned task provides us with one non-zero variable in Y. Hence, we

have:

$$Q \geq 4 \tag{4.90}$$

and

$$Q \times 2 + (\mathrm{L} - Q) \leq \mathrm{L} + 3 \tag{4.91}$$

Rewriting Expression (4.91) gives us:

$$Q \leq 3 \tag{4.92}$$

Expression (4.92) contradicts Expression (4.90). Thus it is impossible for the claim of the lemma to be false and hence the lemma holds. □

Lemma 22. *Consider* $\mathrm{FF_{hf}}$*(ts, pl, ps, t) and assume that* $|\mathrm{aot}(\mathrm{ts},t)| \leq |ps|$*. If* $\sum_{\tau_i \in \mathrm{ts}} u_{i,t} \leq (2/3) \times |ps|$ *then it holds that the execution of* $\mathrm{FF_{hf}}$ *returns* $\mathrm{at} = \mathrm{ts}$

Proof. We prove the claim by contradiction. Suppose that the lemma was incorrect. Then there is a set of tasks (ts), a two-type platform (pl), a set of processors (ps) and a type-id (t) for which it holds that:

$$\sum_{\tau_i \in \mathrm{ts}} u_{i,t} \leq (2/3) \times |ps| \tag{4.93}$$

and

$$\mathrm{FF_{hf}} \text{ returns a set 'at' that is a strict subset of 'ts'} \tag{4.94}$$

Let us explore two cases:

Case (i): $\mathrm{aot}(\mathrm{at},t) \neq \mathrm{aot}(\mathrm{ts},t)$**.** Considering the execution of lines 1-9 and our assumption that $|\mathrm{aot}(\mathrm{ts},t)| \leq |ps|$, we can see that this cannot happen.

Case (ii): $\mathrm{aot}(\mathrm{at},t) = \mathrm{aot}(\mathrm{ts},t)$**.** If this case would have happened then there must have been a task $\tau_i \in \mathrm{ts} \setminus \mathrm{aot}(\mathrm{ts},t)$ such that when executing line 14, it was the case that:

$$\forall p \in ps, \text{it holds that} \sum_{\tau_j \in \mathrm{at}} (x_{j,p} \times u_{j,t}) + u_{i,t} > 1 \tag{4.95}$$

Because of Case (ii), it holds that when this line executed, τ_i has $u_{i,t} \leq 1/3$. Applying it on Expression (4.95) yields:

$$\forall p \in ps, \text{it holds that} \sum_{\tau_j \in \mathrm{at}} (x_{j,p} \times u_{j,t}) + 1/3 > 1 \tag{4.96}$$

Rewriting Expression (4.96) and adding them yields:

$$\sum_{p \in ps} \sum_{\tau_j \in \mathrm{at}} (x_{j,p} \times u_{j,t}) > 2/3 \times |ps|$$

Further rewriting of the above expression yields:

$$\sum_{\tau_j \in \text{at}} u_{j,t} \sum_{p \in ps} x_{j,p} > 2/3 \times |ps| \tag{4.97}$$

Observe that, for each task, $\tau_j \in$ at, it holds that, there is exactly one $p \in ps$ such that $x_{j,p} = 1$. Hence, for each task, $\tau_j \in$ ts, it holds that: $\sum_{p \in ps} x_{j,p} = 1$. Applying this on Expression (4.97) yields:

$$\sum_{\tau_j \in \text{at}} u_{j,t} > 2/3 \times |ps| \tag{4.98}$$

Combining Expression (4.94) and Expression (4.98) yields:

$$\sum_{\tau_j \in \text{ts}} u_{j,t} > 2/3 \times |ps|$$

This contradicts Expression (4.93).

Therefore, regardless of which case is true, it holds that, we obtain a contradiction. Hence, the statement of the lemma is true. □

Lemma 23. *There is no π and τ and τ' such that*

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \text{ and } u'_{i,2} = u_{i,2} \times 3/2$$

and

$$\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

and

$$\text{LPC } \textit{declares FAILURE for inputs } \tau \textit{ and } \pi$$

Proof. If the lemma would be incorrect then it holds that there is a π and τ and τ' such that

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \text{ and } u'_{i,2} = u_{i,2} \times 3/2 \tag{4.99}$$

and

$$\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)\right) \tag{4.100}$$

and

$$\text{LPC declares FAILURE for inputs } \tau \text{ and } \pi \tag{4.101}$$

Because of Expression (4.101), it must have been that one of the lines where the algorithm declares FAILURE has been executed. We will first make a general remark about a class of these failures and then explore each failure individually. For the case that a failure happened at line 27,30,33,36,39,42 (Cases (1)-(6) below), we can reason as follows:

LPC must have executed line 9; so, it must hold that, f='feasible'. Hence, $\text{TLP}_{\text{CUT}}\left(\text{L}(\tau, 2/3), \pi', \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}\right)$ has a feasible solution. (Recall that $\pi' = \pi \setminus \text{rp}$, where rp is a set of three type-1 processors of π.)

Since the optimization problem is feasible, let us discuss the value of its objective function. Recall that Lemma 20 states that: *Consider a task set τ and a two-type platform π. Let τ' be defined as:*

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \ \wedge \ u'_{i,2} = u_{i,2} \times 3/2$$

It then holds that:

$$\text{sched}(\text{OPT}, \tau', \pi) \Rightarrow Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\pi,\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$$

Applying Lemma 20 on a platform with three fewer processors of type-1 gives us: *Consider a task set τ and a two-type platform π. Let τ' be defined as:*

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \ \wedge \ u'_{i,2} = u_{i,2} \times 3/2$$

It then holds that:

$$\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)\right) \Rightarrow$$
$$Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\text{mp}(|P^1(\pi)|-3,|P^2(\pi)|,1,\pi),\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$$

We know that Expression (4.100) is true and since the left-hand side predicate of the above implication is Expression (4.100), it follows that the right-hand side predicate of the implication is true. This gives us:

$$Z_{\text{TLP}_{\text{CUT}}(\text{L}(\tau,2/3),\text{mp}(|P^1(\pi)|-3,|P^2(\pi)|,1,\pi),\text{H1}(\tau,2/3),\text{H2}(\tau,2/3),\text{aot})} \leq 2/3$$

Hence, we have: $z \leq 2/3$.

Therefore, for the case of failure on any of the lines 27, 30, 33, 36, 39 and 42, we have:

$$\text{If the algorithm declares failure on line 27, 30, 33, 36, 39, 42}$$
$$\text{then it holds that: } z \leq 2/3 \tag{4.102}$$

Let us now explore the individual cases:

Case (1): The algorithm declares failure on line 27. If this case would have happened then τ^{A2} is a strict subset of τ^2. Let us explore two cases:

Case (1a): $\sum_{\tau_i \in \tau^2} u_{i,2} > (2/3) \times \left|P^2(\pi)\right|$. Since we experienced Case (1), it holds that we have executed line 13 and evaluated its condition to true. Hence, we have:

$$z \leq 2/3 \tag{4.103}$$

Inspecting TLP_{CUT} and knowing that $z \leq 2/3$ gives us:

$$\sum_{\tau_i \in \text{L}(\tau,2/3) \cup \text{H2}(\tau,2/3)} y_{i,2} \times u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right| \tag{4.104}$$

Let us partition $\text{L}(\tau, 2/3)$ into L1 and L2 and applying it on Expression (4.104) gives us:

$$\sum_{\tau_i \in \text{L1} \cup \text{L2} \cup \text{H2}(\tau,2/3)} y_{i,2} \times u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right| \tag{4.105}$$

Recall that the definition of L1 and L2 it holds that:

$$\forall \tau_i \in \text{L1} : y_{i,1} = 1 \tag{4.106}$$

$$\forall \tau_i \in \text{L2} : y_{i,2} = 1 \tag{4.107}$$

Recall in TLP_{CUT} we have a constraint $y_{i,1} + y_{i,2} = 1$ and clearly our values of Y satisfies that constraint. Applying this on Expression (4.106) gives us:

$$\forall \tau_i \in \text{L1} : y_{i,2} = 0 \tag{4.108}$$

Using Expression (4.108) on Expression (4.105) gives us:

$$\sum_{\tau_i \in \text{L2} \cup \text{H2}(\tau,2/3)} y_{i,2} \times u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right| \tag{4.109}$$

Because of Expression (4.107) and because of line 7 in our algorithm we obtain:

$$\sum_{\tau_i \in \text{L2} \cup \text{H2}(\tau,2/3)} u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right| \tag{4.110}$$

Since $\tau^2 = \text{H2}(\tau, 2/3) \cup \text{L2}$, we get:

$$\sum_{\tau_i \in \tau^2} u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right| \tag{4.111}$$

This contradicts the assumption of Case (1a).

Case (1b): $\sum_{\tau_i \in \tau^2} u_{i,2} \leq (2/3) \times \left|P^2(\pi)\right|$. From Lemma 22, we obtain that the bin-packing scheme in FF_{hf} algorithm would succeed to assign all the tasks and then we would have $\tau^{\text{A2}} = \tau^2$. This contradicts the Case (1).

Case (2): The algorithm declares failure on line 30. If this case would have happened then τ^{A1} is a strict subset of τ^1. The reasoning for this case is similar to the above case (replace $\left|P^2(\pi)\right|$ with $\left|P^1(\pi)\right| - 3$).

Case (3): The algorithm declares failure on line 33. If this case would have happened then $\left|\text{aot}(\tau^2, 2)\right| > \left|P^2(\pi)\right|$. But then TLP_{CUT} would be infeasible. And this contradicts Expression (4.102).

Case (4): The algorithm declares failure on line 36. If this case would have happened then $|\text{aot}(\tau^1, 1)| > \left|P^1(\pi)\right| \setminus \text{rp}$. But then TLP_{CUT} would be infeasible. And this contradicts Expression (4.102).

Case (5): The algorithm declares failure on line 39. If this case would have happened then

$$\tau^{\text{A}} \text{ is a strict subset of } \tau^{\text{F}} \tag{4.112}$$

We know from Lemma 21 that in TLP_{CUT} there are at most three fractionally type-assigned tasks from $\text{L}(\tau, 2/3)$. And we know that the set rp has three processors of type-1. Hence, it is possible to assign each task in τ^{F} to a unique processor in rp. And indeed the execution of line 15, would therefore succeed and hence we would have:

$$\tau^{\text{A}} = \tau^{\text{F}} \tag{4.113}$$

This contradicts Expression (4.112).

Case (6): The algorithm declares failure on line 42. From this case we obtain $z > 2/3$. From Expression (4.102), we have, $z \leq 2/3$. This contradicts the case.

Case (7): The algorithm declares failure on line 45. If this case would have happened then $f \neq$ 'feasible' and hence the optimization problem

$$\text{TLP}_{\text{CUT}}(\text{L}(\tau, 2/3), \pi', \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}) \text{ is infeasible} \tag{4.114}$$

Recall from Expression (4.100) that the following predicate holds true:

$$\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

Applying this on Lemma 6 gives us:

$$\begin{aligned}&\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)\right) \Rightarrow \\ &\text{TLP}_{\text{CUT}}\left(\text{L}(\tau, 2/3), \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right), \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}\right) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{is feasible.}\end{aligned}$$

This gives us that:

$$\begin{aligned}&\text{TLP}_{\text{CUT}}\left(\text{L}(\tau, 2/3), \text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right), \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}\right) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{is feasible.}\end{aligned}$$

Note that $\text{mp}\left(\left|P^1(\pi)\right| - 3, \left|P^2(\pi)\right|, 1, \pi\right)$ and π' have the same number of processors of each type and these processors are from π. Applying this on the above expression gives us:

$$\text{TLP}_{\text{CUT}}\left(\text{L}(\tau, 2/3), \pi', \text{H1}(\tau, 2/3), \text{H2}(\tau, 2/3), \text{aot}\right) \text{ is feasible.}$$

This contradicts Expression (4.114).

Case (8): The algorithm declares failure on line 48. If this case would have happened then there was a task in H12 and this would contradict Expression (4.100).

We see that all cases where LPC declares FAILURE lead to contradiction. Hence the lemma holds. □

Lemma 24. *There is no π and τ such that*

$$\text{sched}(\text{OPT}, \tau, \pi)$$

and

$$\text{LPC } \textit{declares FAILURE with inputs} \tau \textit{ and } \text{mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 3/2, \pi\right)$$

Proof. This follows from the previous lemma obtained after a series of algebraic manipulations. Recall that Lemma 23 states that:
"There is no π and τ and τ' such that

$$\forall \tau'_i \in \tau' : u'_{i,1} = u_{i,1} \times 3/2 \text{ and } u'_{i,2} = u_{i,2} \times 3/2$$

and

$$\text{sched}\left(\text{OPT}, \tau', \text{mp}\left(\left|P^1(\pi)\right|-3, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

and

the algorithm LPC declares FAILURE for inputs τ and π"

Rewriting this so that it makes a statement about the same task set rather than two different (but related) task sets gives us that:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \text{mp}\left(\left|P^1(\pi)\right|-3, \left|P^2(\pi)\right|, 2/3, \pi\right)\right)$$

and

the algorithm LPC declares FAILURE for inputs τ and π"

Scaling the processor speeds of the two platforms that are compared gives us that:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \text{mp}\left(\left|P^1(\pi)\right|-3, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

and

the algorithm LPC declares FAILURE for inputs τ and $\text{mp}\left(\left|P^1(\pi)\right|, \left|P^2(\pi)\right|, 3/2, \pi\right)$"

Consider the statements above with π being replaced by $\text{mp}\left(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi\right)$. This gives us:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT},\tau,\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|-3,\right.$$
$$\left.|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,1,\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and}$$
$$\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,$$
$$3/2,\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))\text{"}$$

Note that the last parameter indicates the platform from which we get processors to form a new platform. Hence the actual number of processors in the platform of the last parameter does not matter. This gives us:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT},\tau,\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|-3,\right.$$
$$\left.|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,1,\text{mp}(\left|P^1(\pi)\right|,\left|P^2(\pi)\right|,1,\pi))\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and}$$
$$\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,$$
$$3/2,\text{mp}(\left|P^1(\pi)\right|,\left|P^2(\pi)\right|,1,\pi))\text{"}$$

Observe that $\text{mp}\left(\left|P^1(\pi)\right|,\left|P^2(\pi)\right|,1,\pi\right)=\pi$. Applying that on the last parameter yields:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT},\tau,\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|-3,\right.$$
$$\left.|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,1,\pi)\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and}$$
$$\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,|P^2(\text{mp}(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi))|,3/2,\pi)\text{"}$$

Observe that $\left|P^2\left(\text{mp}\left(\left|P^1(\pi)\right|+3,\left|P^2(\pi)\right|,1,\pi\right)\right)\right|=\left|P^2(\pi)\right|$. Applying that yields:

"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 1, \pi))|-3, \left|P^2(\pi)\right|, 1, \pi)\right)$$

and

$$\begin{aligned}&\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and}\\ &\text{mp}(|P^1(\text{mp}(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 1, \pi))|, \left|P^2(\pi)\right|, 3/2, \pi)\text{"}\end{aligned}$$

Analogously, observe that, $\left|P^1\left(\text{mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 1, \pi\right)\right)\right| = \left|P^1(\pi)\right|+3$. Applying this yields:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \text{mp}\left(\left|P^1(\pi)\right|+3-3, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 3/2, \pi\right)\text{"}$$

Clearly, $\left|P^1(\pi)\right|+3-3 = \left|P^1(\pi)\right|$. Using it yields:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \text{mp}\left(\left|P^1(\pi)\right|, \left|P^2(\pi)\right|, 1, \pi\right)\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 3/2, \pi\right)\text{"}$$

Observe that $\text{mp}\left(\left|P^1(\pi)\right|, \left|P^2(\pi)\right|, 1, \pi\right) = \pi$. Using it yields:
"There is no π and τ such that

$$\text{sched}\left(\text{OPT}, \tau, \pi\right)$$

and

$$\text{the algorithm LPC declares FAILURE for inputs } \tau \text{ and mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 3/2, \pi\right)\text{"}$$

This states the lemma. □

Theorem 17.

$$\text{sched}\left(\text{OPT}, \tau, \pi\right) \Rightarrow \text{sched}\left(\text{LPC}, \tau, \text{mp}\left(\left|P^1(\pi)\right|+3, \left|P^2(\pi)\right|, 3/2, \pi\right)\right)$$

Proof. Follows from Lemma 24 and the fact that when the algorithm declares success, each processor is utilized to at most 100% and each task is *integrally* assigned to processors. □

Note: The additional three processors in the platform that the algorithm, LPC, uses can either be of type-1 or type-2 or a combination of these two types. We have chosen all the additional processors to be of type-1, for ease of explanation. The result continues to hold for any combination of three additional processors as long as this information is input to the algorithm (so that it can form the remaining set of processors, rp, accordingly — Step 2 in Algorithm 8).

4.5.6 Summary

In this section, for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors, we presented a polynomial time-complexity algorithm, LPC. This algorithm relies on solving a linear program formulation and offers the following guarantee. If a task set has a feasible non-migrative assignment on a two-type platform then, LPC succeeds in finding such a feasible non-migrative assignment as well but on a platform in which each processor is 1.5 times faster and has 3 additional processors. The proposed algorithm, LPC, is shown to be better than the state-of-the-art either in terms of the speed competitive ratio (for systems with large number of processors) or time-complexity or both. To the best of our knowledge, this is the first work to show how cutting planes can be used to improve the speed competitive ratio of algorithms for assigning real-time tasks to heterogeneous processors.

In the next section, we present a polynomial time approximation scheme (PTAS) for the problem of non-migrative task assignment on two-type platforms.

4.6 A polynomial time approximation scheme

4.6.1 Introduction

We now present our fourth and final *non-migrative* task assignment algorithm, PTAS_{NF}. It is a dynamic programming based algorithm with polynomial time-complexity, for assigning tasks in τ to individual processors on a two-type platform π. We also prove its speed competitive ratio against equally powerful non-migrative adversary; its speed competitive ratio depends on a parameter, $\varepsilon > 0$, which is input to the algorithm. Such an algorithm (whose speed competitive ratio is quantified in terms of an input parameter, ε, and whose time-complexity is polynomial) is referred to as a polynomial-time approximation scheme (PTAS).

Definition 19 (PTAS). *A PTAS takes an instance of an optimization problem (for which exact solutions are intractable) and a parameter $\varepsilon > 0$ and, in polynomial time, produces a solution that is within a factor $f(\varepsilon)$ of being optimal, where function $f()$ is independent of the problem instance.*

Related work. As discussed in Section 4.1, the problem of non-migrative task assignment on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, LST90, HS76, JP99, WBB13, CSV12, RAB13, RABN12]. However, as can be seen from Table 4.12 (on page 74), most of these approaches have a speed competitive ratio of at least 1.5 [RA13] or higher [Bar04c, Bar04b, LST90, RAB13, RABN12, CSV12]. The previously proposed PTASs [HS76, JP99, WBB13] have a better speed competitive ratio than these algorithms [Bar04c, Bar04b, LST90, RAB13, RABN12, RA13], in the sense that, these algorithms partition the task set in polynomial time, to any desired degree of accuracy thereby making them theoretically significant results. However, their practical significance is severely limited as these algorithms have a very high run-time complexity — the constants in the run-time expression of these algorithms are prohibitively large. In particular, the PTAS proposed in [WBB13] for assigning tasks to processors on a t-type heterogeneous multiprocessor, has a very high run-time complexity since it "heavily" relies on solving many linear programming formulations. Even on a two-type platform, it has a high run-time complexity which makes its implementation highly inefficient (which is confirmed by our simulations in Section 4.6.9).

Contributions and Significance of the work discussed in this section. We present a polynomial time approximation scheme, PTAS_{NF}, for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors which offers the following guarantee. If there exists a feasible non-migrative assignment of tasks in τ to processors on a two-type platform π then given an $\varepsilon > 0$, PTAS_{NF} succeeds as well, in polynomial time, in finding such a feasible non-migrative task assignment of τ but on a platform $\pi^{(1+3\varepsilon)}$ in which every processor is $1+3\varepsilon$ times faster than the corresponding processor in π.

We believe the significance of this work is as follows. For the problem under consideration, our algorithm, PTAS_{NF}, has superior performance compared to prior state-of-the-art. This can be seen from Table 4.12 since (i) compared to algorithms proposed in [Bar04c, Bar04b, LST90,

Computing Platform	Adversary	Task Assignment Algorithms			
	Task migration	Algorithm	Task migration	Speed competitive ratio	Complexity
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	non-migrative	FF-3C (Section 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA-P (Section 4.4)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	non-migrative	LPC (Section 4.5)	non-migrative	1.5 and 3 extra processors	$O(P)$
2-type	non-migrative	PTAS_{NF}	non-migrative	PTAS	exponential in $1/\varepsilon$

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 4.12: Summary of state-of-the-art task assignment algorithms along with the PTAS_{NF} algorithm proposed in this section.

RAB13, RABN12, RA13] (including FF-3C [RAB13], SA-P [RABN12] and LPC [RA13] proposed in previous sections), it has a better speed competitive ratio and (ii) compared to previous PTASs [HS76, JP99, WBB13], it has a better time-complexity. Specifically, compared to PTAS of [WBB13], referred to as PTAS_{LP} from now on, our PTAS has a much better run-time complexity, in the sense that, it is efficient enough to be usable in practice. We evaluate the average-case performance of PTAS_{NF} and PTAS_{LP} with randomly generated task sets. The evaluation is based on (i) the processor speedup the algorithms need, for a given task set, so as to succeed, compared to an optimal algorithm (i.e., the necessary multiplication factor) and (ii) the average running time. Overall, our algorithm outperforms PTAS_{LP} by requiring much smaller processor speedup and running faster by orders of magnitude. Also, for the vast majority of task sets, it requires significantly smaller necessary multiplication factor than its upper bound of $1+3\varepsilon$.

A global view. The context of the new algorithm, PTAS_{NF}, can be visualized as shown in Figure 4.16.

Organization of Section 4.6. The rest of the section is organized as follows. Section 4.6.2 briefs the system model. Section 4.6.3 gives an overview of our algorithm which categorizes the tasks into *heavy*, *medium* and *light* tasks and makes different provisions for assigning these tasks. Section 4.6.4 discusses the assignment of strictly heavy tasks to processors and presents the corre-

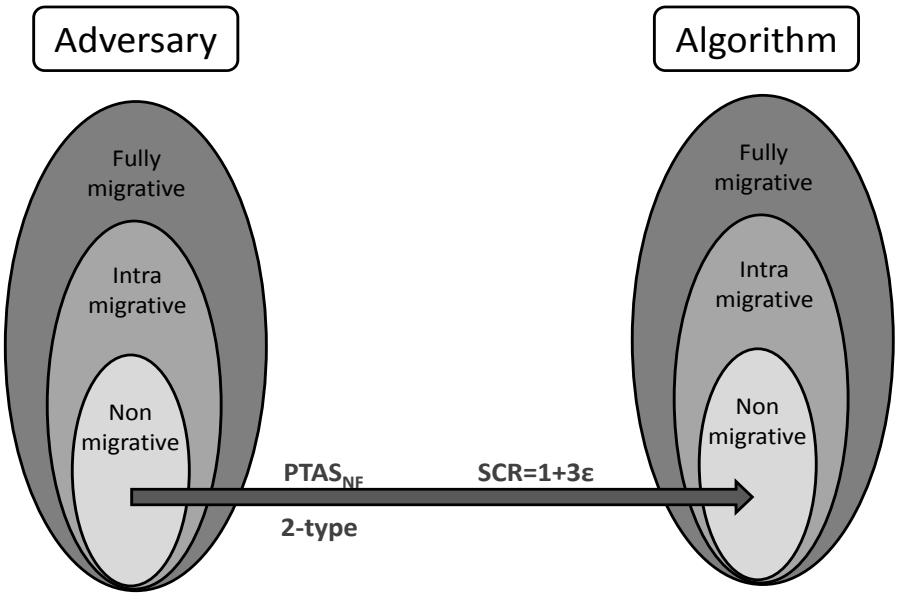


Figure 4.16: A global view of the new algorithm, PTAS_{NF}, proposed in this section. Here, SCR denotes the "speed competitive ratio" and $\varepsilon > 0$ is an input parameter to the algorithm.

sponding analysis. Section 4.6.5 discusses the fractional assignment of medium tasks to processors and presents the corresponding analysis. Analogously, Section 4.6.7 discusses the fractional assignment of light tasks to processors and presents the corresponding analysis. Then, Section 4.6.8 describes the (integral) assignment of both medium and light tasks to processors (which were previously assigned fractionally) and provides analysis for such an assignment. Section 4.6.9 presents the average-case performance evaluation of the proposed algorithm and compares it with the prior state-of-the-art algorithm [WBB13]. Finally, Section 4.6.10 concludes.

4.6.2 System model

We consider the problem of non-migrative assignment of a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks on a two-type heterogeneous multiprocessor platform π comprising m processors, of which m_1 are of type-1 and m_2 are of type-2. We assume that an optimal scheduling algorithm such as EDF is used to schedule the tasks on each processor.

On a two-type platform, the WCET of a task depends on the processor type on which it executes. We denote by C_i^1 and C_i^2 the WCET of a task τ_i on processors of type-1 and type-2, respectively. The minimum inter-arrival time of task τ_i is denoted by T_i. We denote by $u_i \stackrel{\text{def}}{=} C_i^1/T_i$ and $v_i \stackrel{\text{def}}{=} C_i^2/T_i$ its utilizations on type-1 and type-2 processors, respectively. A task τ_i that cannot be executed on processors of type-1 (respectively, type-2) is modeled by setting its $u_i = \infty$ (respectively, $v_i = \infty$).

4.6.3 An overview of our approach

We now give an overview of our algorithm (referred to as PTAS_{NF} since it uses "Next-Fit"). Our PTAS takes $\varepsilon > 0$ as an input parameter and outputs a feasible non-migrative assignment. Let us partition the given task set τ into two subsets as follows:

$$\tau_{\text{hvy}} = \{\tau_i \mid u_i \geq \varepsilon \text{ or } v_i \geq \varepsilon\} \tag{4.115}$$

$$\tau_{\text{lgt}} = \tau \setminus \tau_{\text{hvy}} = \{\tau_i \mid u_i < \varepsilon \text{ and } v_i < \varepsilon\} \tag{4.116}$$

Intuitively, τ_{hvy} refers to "heavy" tasks and τ_{lgt} refers to "light" tasks. Our PTAS, has the following steps:

Step 1. We first approximate the utilizations of every task in τ_{hvy} to some finite number of pre-computed values. The motivation for doing this is twofold: (i) by restricting the number of pre-computed values to a constant, we ensure polynomial complexity for the algorithm and (ii) by choosing these values cleverly, we ensure the speed competitive ratio of the algorithm is bounded. Then, we assign the tasks in τ_{hvy} to processors using the algorithm A_{hvy} described in Section 4.6.4.1. In Section 4.6.4.5, we show that after using A_{hvy}, the sum of the utilizations of the tasks assigned on processors of type-1 (respectively, type-2) does not exceed $(1+\varepsilon) \times m_1$ (respectively, $(1+\varepsilon) \times m_2$).

Step 2. Some tasks from τ_{hvy}, i.e., some tasks with $u_i \geq \varepsilon \ \wedge \ v_i < \varepsilon$ or $u_i < \varepsilon \ \wedge \ v_i \geq \varepsilon$ may remain unassigned after using A_{hvy}. These unassigned tasks form the set, τ_{int} ("intermediate" tasks). Now, A_{int} *fractionally* assigns the tasks (i.e., tasks can be split between processors) with $u_i < \varepsilon \ \wedge \ v_i \geq \varepsilon$ (respectively, $u_i \geq \varepsilon \ \wedge \ v_i < \varepsilon$) to type-1 (respectively, type-2) processors as described in Section 4.6.6. In Section 4.6.6.1, we show that after using A_{int}, the sum of the utilizations of all the tasks assigned so far on processors of type-1 (respectively, type-2) still does not exceed $(1+\varepsilon) \times m_1$ (respectively, $(1+\varepsilon) \times m_2$).

Step 3. Fractionally assign the tasks in τ_{lgt} to processors using the algorithm A_{lgt} (which makes use of a fractional knapsack property) described in Section 4.6.7.1. In Section 4.6.7.2, we show that after using A_{lgt}, the sum of the utilizations of all the tasks assigned so far on processors of type-1 (respectively, type-2) does not exceed $(1+2\varepsilon) \times m_1$ (respectively, $(1+2\varepsilon) \times m_2$).

Step 4. Finally, those tasks from τ_{int} and τ_{lgt} that were assigned fractionally by A_{int} and A_{lgt} are assigned *integrally* using the algorithm, A_{fract}, described in Section 4.6.8.1. In Section 4.6.8.2, we show that after using A_{fract}, the sum of the utilizations of all the tasks assigned so far on processors of type-1 (respectively, type-2) does not exceed $(1+3\varepsilon) \times m_1$ (respectively, $(1+3\varepsilon) \times m_2$). Hence, we conclude that if there exists a feasible non-migrative assignment of the task set τ on the two-type platform π then PTAS_{NF} succeeds as well in finding such a feasible non-migrative assignment of τ but on the platform, $\pi^{(1+3\varepsilon)}$, in which every processor is $1+3\varepsilon$ times faster.

4.6.4 Assigning the tasks in τ_{hvy} (Step 1)

In this section, we describe the algorithm, A_{hvy}, for integrally assigning (a subset of) the tasks in τ_{hvy} to processors and also analyze its returned assignment.

4.6.4.1 Description of A_{hvy} algorithm

It consists of three steps listed below which are in turn discussed next in detail:

Step 1.1. It defines a finite set, $S(\varepsilon)$, of utilization values, based on the value of the input parameter, ε. Then, it computes the "rounded-down utilizations" u_i^{rd} and v_i^{rd} of every task, $\tau_i \in \tau$, by rounding *down* u_i and v_i to one of the quantized values in $S(\varepsilon)$. We will denote by $\tau_{\text{hvy}}^{\text{rd}}$ the set of tasks obtained by rounding down the utilizations of the tasks of τ_{hvy}.

Step 1.2. It uses dynamic programming to determine, in polynomial time, (i) all the subsets of $\tau_{\text{hvy}}^{\text{rd}}$ that can be non-migratively assigned to m_1 processors of type-1 and (ii) all the subsets that can be non-migratively assigned to m_2 processors of type-2.

Step 1.3. It exhaustively considers each pair of subsets such that one subset can be assigned to m_1 processors of type-1 and the other subset can be assigned to m_2 processors of type-2. Using the ordered pair of subsets under consideration, it integrally assigns (a subset of) the tasks from τ_{hvy} to processors (at least all the tasks with $u_i \geq \varepsilon \wedge v_i \geq \varepsilon$).

4.6.4.2 Step 1.1: Rounding-down the utilizations of the tasks

We compute the set $S(\varepsilon)$ of all real numbers ≤ 1 that are of the form $\varepsilon(1+\varepsilon)^k$, for all integers $k \geq 0$. Then, we compute the rounded-down utilizations u_i^{rd} and v_i^{rd} of every task, $\tau_i \in \tau$, by rounding down each of its utilizations (u_i and v_i) to the nearest value present in the set $S(\varepsilon)$. For tasks with $u_i < \varepsilon$ (respectively, $v_i < \varepsilon$), we set $u_i^{\text{rd}} = 0$ (respectively, $v_i^{\text{rd}} = 0$) and for tasks with $u_i = \infty$ (respectively, $v_i <= \infty$), we set $u_i^{\text{rd}} = \infty$ (respectively, $v_i^{\text{rd}} = \infty$). The definition of $S(\varepsilon)$ leads to the following property.

Property 2. *For a task τ_i, if it holds that $\varepsilon \leq u_i \leq 1$ then there exists k such that $\varepsilon(1+\varepsilon)^k \leq u_i < \varepsilon(1+\varepsilon)^{k+1}$ and thus*

$$\frac{u_i}{u_i^{\text{rd}}} = \frac{u_i}{\varepsilon(1+\varepsilon)^k} < \frac{\varepsilon(1+\varepsilon)^{k+1}}{\varepsilon(1+\varepsilon)^k} = (1+\varepsilon) \tag{4.117}$$

The same holds for v_i.

Therefore, if the utilizations of each task is reduced by this maximal factor, it follows that any collection of tasks with their reduced utilizations summing to ≤ 1 would have their original utilizations summing to $\leq (1+\varepsilon)$.

Let us now determine the number L of distinct values in $S(\varepsilon)$. Since only values with $\varepsilon(1+\varepsilon)^k \leq 1$ are included in $S(\varepsilon)$, it holds that, $k\log(1+\varepsilon) \leq \log(1/\varepsilon)$ and thus, $k \leq \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)}$. Then we conclude that:

$$L = \left\lfloor \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)} \right\rfloor + 1$$

For each ℓ, $0 \leq \ell < L$, we denote by X_ℓ (respectively, Y_ℓ) the number of tasks in $\tau^{\text{rd}}_{\text{hvy}}$ with u^{rd}_i (respectively, v^{rd}_i) equal to $\varepsilon(1+\varepsilon)^\ell \in S(\varepsilon)$. The task set, $\tau^{\text{rd}}_{\text{hvy}}$, can thus be represented by $2 \times L$ non-negative integers $X_0, X_1, \ldots, X_{L-1}, Y_0, Y_1, Y_{L-1}$. Note that each X_ℓ and each Y_ℓ is no greater than $\left|\tau_{\text{hvy}}\right|$.

4.6.4.3 Step 1.2: Generating the feasible configurations

The rounding down of the utilizations described in the previous section ensures that the utilizations of the tasks in τ_{hvy} may only take one of the values in $S(\varepsilon)$, resulting in the set $\tau^{\text{rd}}_{\text{hvy}}$. In this section, using dynamic programming, we determine, in polynomial time, all the subsets of $\tau^{\text{rd}}_{\text{hvy}}$ that can be non-migratively assigned to m_1 processors of type-1 (respectively, m_2 processors of type-2). Once all the feasible subsets (also referred to as *feasible configurations*) are determined, we use this information to assign a subset of tasks from τ_{hvy} on type-1 and type-2 processors (described in Section 4.6.4.4).

Definition 20 (feasible configurations). *Consider any L-tuple, $T = (x_0,\ x_1,\ \ldots,\ x_{L-1})$, where $x_\ell \geq 0, \forall \ell \in [0, L-1]$, and let $\tau_{(T)}$ denote a task set containing exactly x_ℓ tasks τ_i of utilization $u_i = \varepsilon(1+\varepsilon)^\ell$ for each ℓ. The L-tuple T is said to be a feasible configuration on m_1 processors of type-1 if and only if there exists a feasible non-migrative assignment for the corresponding task set $\tau_{(T)}$ on m_1 processors of type-1. Analogously, we define an L-tuple, $(y_0, y_1, \ldots, y_{L-1})$, with v_i values that is a feasible configuration on m_2 processors of type-2.*

The algorithm, A_{hvy}, uses the same approach as the one presented in [Bar11] to determine all the configurations, $(x_0, x_1, \ldots, x_{L-1})$, of tasks in $\tau^{\text{rd}}_{\text{hvy}}$ (respectively, $(y_0, y_1, \ldots, y_{L-1})$) that are *feasible* on m_1 processors of type-1 (respectively, m_2 processors of type-2), in which $x_\ell \leq X_\ell \leq \left|\tau_{\text{hvy}}\right|$ (respectively, $y_\ell \leq Y_\ell \leq \left|\tau_{\text{hvy}}\right|$) for each ℓ, $0 \leq \ell < L$. This approach [Bar11] is summarized below. As there are no more than $\Pi_{\ell=0}^{L-1}(1+X_\ell) \leq \Pi_{\ell=0}^{L-1}(1+\left|\tau_{\text{hvy}}\right|) = O(n^L)$ such feasible configurations on type-1 processors (and the same holds for type-2 processors) and since L is a constant for a given value of ε, the time to determine all the feasible configurations is polynomial in n.

Summary of the approach in [Bar11]: It constructs two separate tables: one table each for storing the information about all the configurations on processors of each type. The table for type-1 processors has m_1 rows and $\Pi_{\ell=0}^{L-1}(1+X_\ell)$ columns. Each column corresponds to a different configuration and each cell has a value $\in \{\text{yes}, \text{no}\}$. A cell in the i'th row and the j'th column is an "yes" if the corresponding configuration is feasible on i processors of type-1. This table is filled row-wise starting with the first row. Filling in the first row is straightforward for all the configurations: it is an "yes" if the corresponding configuration, say $(x_0, x_1, \ldots, x_{L-1})$, is feasible on a single processor, i.e., if $\sum_{\ell=0}^{L-1} x_\ell \times \varepsilon(1+\varepsilon)^\ell \leq 1$, it is a "no" otherwise. The i'th row is filled in by using the entries of the $(i-1)$'th row. Specifically, for the configuration corresponding to the j'th column, say $(x_0, x_1, \ldots, x_{L-1})$, the cell at the i'th row is a "yes" if and only if there exists two configurations $(x'_0, x'_1, \ldots, x'_{L-1})$ and $(x''_0, x''_1, \ldots, x''_{L-1})$ such that

1. $(x'_0, x'_1, \ldots, x'_{L-1})$ is a feasible configuration on $(i-1)$ processors of type-1;

2. $(x''_0, x''_1, \ldots, x''_{L-1})$ is a feasible configuration on one processor of type-1; and

3. $x_\ell = x'_\ell + x''_\ell$, for all $0 \leq \ell < L$.

For each cell in the i'th row, there are polynomially many possible candidates for the role of $(x'_0, x'_1, \ldots, x'_{L-1})$; hence, each cell in the i'th row can be filled in polynomial time. Similarly, the second table for type-2 processors is constructed.
Note: By using standard dynamic programming tricks which require storing additional information [Bar11], we can obtain a non-migrative assignment from the feasible configurations.

4.6.4.4 Step 1.3: Determining the partitioning

Using the two configuration tables that were constructed in the previous step, we now determine a non-migrative task assignment for (a subset of) the heavy tasks. The main idea is as follows. Suppose that the task set τ can indeed be non-migratively assigned so as to meet all deadlines on the given platform and let $\mathscr{H}_{\text{feas}}$ denote one such feasible non-migrative assignment. For each ℓ, $0 \leq \ell < L$, let x_ℓ^{feas} denote the number of tasks τ_i satisfying $\varepsilon(1+\varepsilon)^\ell \leq u_i < \varepsilon(1+\varepsilon)^{\ell+1}$ that are assigned to type-1 processors in $\mathscr{H}_{\text{feas}}$. Since $\mathscr{H}_{\text{feas}}$ is a feasible non-migrative assignment, the configuration $(x_0^{\text{feas}}, x_1^{\text{feas}}, \ldots, x_{L-1}^{\text{feas}})$ must appear in the table constructed in the previous step for type-1 processors and the cell at the m_1'th row of the corresponding column must contain "yes". Analogously, the configuration $(y_0^{\text{feas}}, y_1^{\text{feas}}, \ldots, y_{L-1}^{\text{feas}})$ must appear in the table constructed for type-2 processors and the cell at the m_2'th row of the corresponding column must contain "yes". However, since we do not know which of the feasible configurations in our tables correspond to $\mathscr{H}_{\text{feas}}$, we consider every ordered pair of configurations that are feasible on m_1 and m_2 processors of type-1 and type-2 respectively. Since there are only polynomially (i.e., $O(n^L)$) many distinct feasible configurations in each table, it follows that there are at most polynomially many such ordered pairs of feasible configurations to consider.

For each considered ordered pair of configurations, by assuming that they are the ones corresponding to $\mathscr{H}_{\text{feas}}$, we attempt to construct a *similar* non-migrative assignment for the tasks in τ_{hvy} as that of $\mathscr{H}_{\text{feas}}$. The assignment obtained will be *similar* to $\mathscr{H}_{\text{feas}}$ in the following sense: although the tasks assigned in both the assignments may not be the same, it holds that (as we show later), the sum of utilizations of the tasks assigned by our algorithm on each processor type does not exceed that of $\mathscr{H}_{\text{feas}}$ by a factor of $1+\varepsilon$.

Let $\{(x_0, x_1, \ldots, x_{L-1}), (y_0, y_1, \ldots, y_{L-1})\}$ denote the currently considered ordered pair of feasible configurations on m_1 and m_2 processors of type-1 and type-2, respectively. The algorithm, A_{hvy}, to determine the corresponding task-to-processor assignment of tasks from τ_{hvy} is as follows.
Step 1.3.1. For each ℓ, $0 \leq \ell \leq L-1$, A_{hvy} assigns *exactly* x_ℓ tasks τ_i satisfying $u_i^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ to type-1 processors. Specifically, for each ℓ,

1.3.1.1 If there are *fewer* than x_ℓ such tasks in τ_{hvy}, then A_{hvy} declares failure with respect to this particular ordered pair of feasible configurations, and moves on to the next ordered pair of feasible configurations.

1.3.1.2 If there are *exactly* x_ℓ such tasks then A_{hvy} assigns all of them to type-1 processors.

1.3.1.3 If there are *more* than x_ℓ such tasks, it assigns x_ℓ of them to type-1 processors by favoring those with larger v_i.

Step 1.3.2. After assigning tasks to processors of type-1, A_{hvy} assigns the remaining tasks to processors of type-2 as follows. For each ℓ, starting with $\ell = L-1$ and repeatedly decreasing ℓ by one until ℓ equals 0,

1.3.2.1 If there are *less* than y_ℓ unassigned tasks τ_i satisfying $v_i^{rd} = \varepsilon(1+\varepsilon)^\ell$ (say, n_1 tasks), then A_{hvy} assigns these n_1 tasks to type-2 processors. Then, A_{hvy} assigns $y_\ell - n_1$ other (unassigned) tasks τ_j with smaller utilization on type-2 processors (i.e., $v_j^{rd} < \varepsilon(1+\varepsilon)^\ell$), by favoring those with larger v_j and within these tasks that are favored, those with larger u_i are favored.

1.3.2.2 If there are *exactly* y_ℓ unassigned tasks τ_i satisfying $v_i^{rd} = \varepsilon(1+\varepsilon)^\ell$ then all of them are assigned to type-2 processors.

1.3.2.3 If there are *more* than y_ℓ unassigned tasks τ_i satisfying both (i) $v_i^{rd} = \varepsilon(1+\varepsilon)^\ell$ and (ii) $u_i^{rd} > 0$, then A_{hvy} declares failure with respect to this particular ordered pair of feasible configurations and moves on to the next ordered pair of feasible configurations.

1.3.2.4 If there are more than y_ℓ unassigned tasks τ_i satisfying $v_i^{rd} = \varepsilon(1+\varepsilon)^\ell$ but not more than y_ℓ of these tasks have $u_i^{rd} > 0$, then A_{hvy} assigns y_ℓ of these tasks by favoring those with larger u_i.

Step 1.3.3. If any task τ_i remains unassigned with both $u_i^{rd} > 0$ and $v_i^{rd} > 0$, A_{hvy} declares failure with respect to this particular ordered pair of feasible configurations, and moves on to the next ordered pair of feasible configurations.

If A_{hvy} did not declare failure in any of the above steps, implying that *all* the tasks with $u_i \geq \varepsilon \wedge v_i \geq \varepsilon$ are assigned (and may be *few* other tasks from τ_{hvy} with $u_i \geq \varepsilon \ \wedge \ v_i < \varepsilon$ or $u_i < \varepsilon \ \wedge \ v_i \geq \varepsilon$) then algorithm A_{int} is called with the ordered pair of feasible configurations under consideration. This algorithm, A_{int}, is presented in Section 4.6.5.

4.6.4.5 Assignment analysis

Let $\mathscr{H}_{hvy}$ denote the assignment of the heavy tasks returned by A_{hvy}. In this section, we show that in $\mathscr{H}_{hvy}$, the subset of tasks assigned to each processor consumes no more than $1+\varepsilon$ of the capacity of that processor.

Definition 21 (**The subsets Γ^1_{hvy} and Γ^2_{hvy}**). *We denote by $\Gamma^1_{hvy}, \Gamma^2_{hvy} \subseteq \tau_{hvy}$ the subsets of tasks assigned to the processors of type-1 (respectively, type-2) in the assignment $\mathscr{H}_{hvy}$ returned by the algorithm, A_{hvy}.*

Remark about notation. Hereafter, we use the notation τ for the subsets of tasks that we explicitly define (τ_{hvy} and τ_{lgt}, for example), Γ for the subsets of tasks returned by the different steps of our PTAS and Φ for the subsets of tasks assigned in $\mathscr{H}_{\text{feas}}$.

We know that the ordered pair of feasible configurations $\{(x_0^{\text{feas}}, x_1^{\text{feas}}, \ldots, x_{L-1}^{\text{feas}}), (y_0^{\text{feas}}, y_1^{\text{feas}}, \ldots, y_{L-1}^{\text{feas}})\}$ corresponding to the feasible partitioning $\mathscr{H}_{\text{feas}}$ must be present in the tables constructed in Step 1.2 (in Section 4.6.4.3). Therefore, this particular ordered pair of feasible configurations (denoted by P^{feas} hereafter) will come to be considered by A_{hvy}.

Lemma 25. *If P^{feas} is the ordered pair of feasible configurations currently under consideration by A_{hvy}, then A_{hvy} successfully terminates (i.e., without declaring failure) and it holds that every task, $\tau_i \in \Gamma^1_{\text{hvy}}$, can be 1:1 mapped to exactly one task, τ_k, that is assigned to a type-1 processor in $\mathscr{H}_{\text{feas}}$ such that $u_i \leq (1+\varepsilon)u_k$. An analogous property holds for the tasks in Γ^2_{hvy} (such that $v_i \leq (1+\varepsilon)v_k$).*

Proof. First, let us focus on the tasks in Γ^1_{hvy}. In Step 1.3.1, for each $\ell \in [0, L-1]$, it is straightforward (from the fact that we consider the ordered pair P^{feas}) to see that A_{hvy} successfully assigns *exactly* x_ℓ^{feas} tasks τ_i satisfying $\varepsilon(1+\varepsilon)^\ell \leq u_i < \varepsilon(1+\varepsilon)^{\ell+1}$ to type-1 processors (through either case 1.3.1.2 or 1.3.1.3). While these may not be the same tasks as those that are assigned to these processors in $\mathscr{H}_{\text{feas}}$, the utilization of each task does not exceed that of the corresponding task assigned in $\mathscr{H}_{\text{feas}}$ by more than a factor of $(1+\varepsilon)$. Hence the lemma holds for the heavy tasks in Γ^1_{hvy}.

Now, let us focus on Step 1.3.2, i.e., on the tasks in Γ^2_{hvy}. If A_{hvy} terminates without declaring failure then it means that for each $\ell \in [0, L-1]$, A_{hvy} went through either case 1.3.2.1, 1.3.2.2 or 1.3.2.4 and it is trivial to see that the lemma holds for all these cases. Indeed, for each task τ_i with $\varepsilon(1+\varepsilon)^\ell \leq v_i < \varepsilon(1+\varepsilon)^{\ell+1}$ that is assigned to processors of type-2 through one of these cases, there is a task, say τ_k, also with $\varepsilon(1+\varepsilon)^\ell \leq v_k < \varepsilon(1+\varepsilon)^{\ell+1}$ which is also assigned to processors of type-2 in $\mathscr{H}_{\text{feas}}$ (since we consider the ordered pair P^{feas}).

Since we have shown that the lemma holds as long as A_{hvy} does not declare failure, we now show that A_{hvy} cannot fail while considering the ordered pair P^{feas} of feasible configurations. For a failure to occur, it is necessary for A_{hvy} to go through case 1.3.2.3, i.e., there must be some $\ell \in [0, L-1]$ such that there are strictly more than y_ℓ^{feas} tasks τ_i yet unassigned, that satisfy both $v_i^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ and $u_i^{\text{rd}} > 0$. Let us consider the largest such ℓ and denote by $n_1 > y_\ell^{\text{feas}}$ the number of tasks satisfying both the aforementioned conditions. Recall that in $\mathscr{H}_{\text{feas}}$, y_ℓ^{feas} tasks τ_i with $v_i^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ are assigned to type-2 processors. Therefore, it must be the case that in $\mathscr{H}_{\text{feas}}$, some of the $n_1 - y_\ell^{\text{feas}}$ "additional" tasks were assigned to type-1 processors. Let τ_j denote one of these additional tasks, thus satisfying $v_j^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ and $u_j^{\text{rd}} = \varepsilon(1+\varepsilon)^x > 0$, for some $x \in [0, L-1]$. Since this task τ_j has not been assigned yet by A_{hvy}, we know that at the time A_{hvy} was assigning tasks in Step 1.3.1 with $\ell = x$, it went through case 1.3.1.3 and instead of choosing to assign τ_j, it chose to assign another task $\tau_k \neq \tau_j$, also with $u_k^{\text{rd}} = \varepsilon(1+\varepsilon)^x$, that is assigned to type-2 processors in $\mathscr{H}_{\text{feas}}$. Furthermore, according to case 1.3.1.3, it must hold that $v_k^{\text{rd}} \geq v_j^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$. Now, two cases may arise.

Case 1. If $v_k^{\text{rd}} = v_j^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ then τ_k is one of the y_ℓ^{feas} tasks assigned to type-2 processors in $\mathscr{H}_{\text{feas}}$ and, since A_{hvy} assigned τ_k to type-1 processors, there is a free "slot" on type-2 processors in which τ_j can fit. This contradicts our assumption that τ_j is unassigned at this time instant.

Case 2. If $v_k^{\text{rd}} > v_j^{\text{rd}} = \varepsilon(1+\varepsilon)^\ell$ then τ_k is one of the y_r^{feas} tasks (with $r > \ell$) assigned to type-2 processors in $\mathscr{H}_{\text{feas}}$ and, since A_{hvy} assigned τ_k to type-1 processors, there *was* a free slot on type-2 processors in Step 1.3.2, when ℓ was equal to r. At this moment, when $\ell = r$, A_{hvy} necessarily went through case 1.3.2.1 and since this case allows tasks with smaller utilization on type-2 processors to be accommodated in unused slots that were reserved for tasks with larger utilization, τ_j must have been assigned at that moment. This contradicts our assumption that τ_j is unassigned at this time instant.

Hence, we can conclude that A_{hvy} does not declare failure for the ordered pair P^{feas} of feasible configurations and the lemma holds for every task in $\Gamma^1_{\text{hvy}} \cup \Gamma^2_{\text{hvy}}$. □

Definition 22 (The corresponding sets Φ^1_{hvy} and Φ^2_{hvy}). *We define by Φ^1_{hvy} the set of tasks assigned to type-1 processors in $\mathscr{H}_{\text{feas}}$ such that each task $\tau_k \in \Phi^1_{\text{hvy}}$ can be mapped to exactly one task $\tau_i \in \Gamma^1_{\text{hvy}}$ (bijective relation, implying $\left|\Phi^1_{\text{hvy}}\right| = \left|\Gamma^1_{\text{hvy}}\right|$) and for which $u_i \leq (1+\varepsilon)u_k$. The set Φ^2_{hvy} is defined analogously (for which $v_i \leq (1+\varepsilon)v_k$)*[8].

Lemma 26. *After assigning the tasks in τ_{hvy}, we have*

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i \leq (1+\varepsilon)m_1 \tag{4.118}$$

$$\textit{and} \quad \sum_{\tau_i \in \Gamma^2_{\text{hvy}}} v_i \leq (1+\varepsilon)m_2 \tag{4.119}$$

Proof. We show only the proof of Expression (4.118), as the proof of Expression (4.119) is quite similar. The proof is a direct consequence of Lemma 25. We know from Lemma 25 and Definition 22 that, there exists a $1:1$ mapping between every task $\tau_i \in \Gamma^1_{\text{hvy}}$ and every task $\tau_k \in \Phi^1_{\text{hvy}}$ such that, $u_i \leq (1+\varepsilon)u_k$. Therefore, since $|\Phi^1_{\text{hvy}}| = |\Gamma^1_{\text{hvy}}|$ (from the bijective relation between the two sets), we have:

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i \leq (1+\varepsilon) \sum_{\tau_k \in \Phi^1_{\text{hvy}}} u_k \tag{4.120}$$

Finally, we know from the feasibility of $\mathscr{H}_{\text{feas}}$ that, $\sum_{k \in \Phi^1_{\text{hvy}}} u_k \leq m_1$, and hence it holds that: $\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i \leq (1+\varepsilon)m_1$. □

[8] Note that, Lemma 25 showed that, such task sets Φ^1_{hvy} and Φ^2_{hvy} exist.

4.6.5 Assigning the tasks in τ_{int} (Step 2)

The tasks from τ_{hvy} that were not assigned by algorithm, A_{hvy}, form the set, τ_{int}, i.e., $\tau_{\text{int}} = \tau_{\text{hvy}} \setminus \{\Gamma^1_{\text{hvy}} \cup \Gamma^2_{\text{hvy}}\}$. Let us partition τ_{int} into two subsets τ^1_{int} and τ^2_{int} as follows:

$$\tau^1_{\text{int}} = \{\tau_i \in \tau_{\text{int}} \mid u_i < \varepsilon \text{ and } v_i \geq \varepsilon\} \tag{4.121}$$

$$\tau^2_{\text{int}} = \{\tau_i \in \tau_{\text{int}} \mid u_i \geq \varepsilon \text{ and } v_i < \varepsilon\} \tag{4.122}$$

4.6.6 The description of the algorithm A_{int}

The algorithm, A_{int}, to assign the tasks in τ_{int} is as follows:

1. Assign all the tasks in τ^1_{int} to type-1 processors using the *wrap-around* technique. This technique works as follows. Take the first processor of type-1 and assign as many of the tasks as possible from τ^1_{int} "integrally" onto that processor. When a task fails to be assigned integrally, assign that task "fractionally" such that the current processor is filled completely and the remaining fraction is assigned to the next processor of type-1, continue this procedure until all the tasks from τ^1_{int} are assigned to type-1 processors.
2. Analogously, assign all the tasks in τ^2_{int} to type-2 processors using the *wrap-around* technique.

4.6.6.1 Assignment analysis

We now show that for a task set, τ, that is feasible on a platform, π, A_{int} always succeeds in assigning all the tasks in τ^1_{int} to type-1 processors on a platform $\pi^{(1+\varepsilon)}$. That is, if Γ^1_{int} and Γ^2_{int} denote the set of tasks assigned to type-1 and type-2 processors by A_{int}, we have $\Gamma^1_{\text{int}} = \tau^1_{\text{int}}$ and $\Gamma^2_{\text{int}} = \tau^2_{\text{int}}$.

In the following lemma, we make use of the fact that the two sets of tasks, Γ^1_{hvy} and Γ^2_{hvy}, have been obtained by algorithm A_{hvy}, using the ordered pair P^{feas} of feasible configurations.

Lemma 27. *After assigning all the tasks in τ_{int} using the ordered pair of feasible configuration , we have:*

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \tau^1_{\text{int}}} u_i \leq (1+\varepsilon) m_1 \tag{4.123}$$

$$\textit{and} \quad \sum_{\tau_i \in \Gamma^2_{\text{hvy}}} v_i + \sum_{\tau_i \in \tau^2_{\text{int}}} v_i \leq (1+\varepsilon) m_2 \tag{4.124}$$

Proof. In the feasible assignment, $\mathscr{H}_{\text{feas}}$, $\left|\tau^1_{\text{int}}\right|$ number of tasks with $u_i < \varepsilon \wedge v_i \geq \varepsilon$ must have been assigned to type-1 processors. This is a consequence of the fact that, P^{feas} contains exactly the same number of tasks with utilization $\geq \varepsilon$ on the processor that they are assigned to, as in $\mathscr{H}_{\text{feas}}$. Let Φ^1_{int} denote the set of tasks with $u_i < \varepsilon \wedge v_i \geq \varepsilon$ that are assigned to type-1 processors in $\mathscr{H}_{\text{feas}}$. Since $\mathscr{H}_{\text{feas}}$ is a feasible assignment, it holds that,

$$\sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i \leq m_1 \tag{4.125}$$

Since the number of tasks with $u_i < \varepsilon \wedge v_i \geq \varepsilon$ that have been assigned to type-1 processors is same in both $\mathscr{H}_{\text{feas}}$ and the assignment computed by our algorithm, we have $\left|\tau^1_{\text{int}}\right| = \left|\Phi^1_{\text{int}}\right| = \left|\Gamma^1_{\text{int}}\right|$. Here, it is worth recalling Step 1.3.1.3 and Step 1.3.2.4 of algorithm A_{hvy}. In these steps, while assigning the tasks to processors of type-1 (respectively, type-2), when A_{hvy} has to choose few tasks to assign from the available set of tasks, it always chooses those tasks that have a larger utilization on type-2 (respectively, type-1) processors (leaving "easier" tasks for A_{int} to assign). Now coming back to A_{int} algorithm, although the tasks (with $u_i < \varepsilon \wedge v_i \geq \varepsilon$) assigned by A_{int} to type-1 processors may not be the *same* as those assigned by $\mathscr{H}_{\text{feas}}$, we can infer that:

$$\sum_{\tau_i \in \tau^1_{\text{int}}} u_i \leq \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i \tag{4.126}$$

Applying Inequality (4.120) and (4.126) on Inequality (4.125), we get:

$$\frac{\sum_{i \in \Gamma^1_{\text{hvy}}} u_i}{1+\varepsilon} + \sum_{\tau_i \in \tau^1_{\text{int}}} u_i \leq m_1 \tag{4.127}$$

Multiplying Inequality (4.127) by $1+\varepsilon$ and from trivial arithmetic $\sum_{i \in \tau^1_{\text{int}}} u_i \leq (1+\varepsilon) \times \sum_{i \in \tau^1_{\text{int}}} u_i$, we obtain:

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \tau^1_{\text{int}}} u_i \leq (1+\varepsilon) \times m_1$$

Using similar reasoning as above, we can show that Expression (4.124) holds as well. Hence the proof. □

Corollary 7. *After assigning the tasks in τ_{int}, we have:*

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}} \cup \Gamma^1_{\text{int}}} u_i \leq (1+\varepsilon) \sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}}} u_i \tag{4.128}$$

and

$$\sum_{\tau_i \in \Gamma^2_{\text{hvy}} \cup \Gamma^2_{\text{int}}} v_i \leq (1+\varepsilon) \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}}} v_i \tag{4.129}$$

Proof. Expression (4.128) follows from Expression (4.120) and Expression (4.126) (since $\Gamma^1_{\text{int}} = \tau^1_{\text{int}}$) and Expression (4.129) can be inferred from analogous expressions for type-2 processors. □

4.6.7 Assigning the tasks in τ_{lgt} (Step 3)

Let us partition τ_{lgt} into τ^1_{lgt} and τ^2_{lgt} as follows:

$$\tau^1_{\text{lgt}} = \{\tau_i \in \tau_{\text{lgt}} \mid u_i \leq v_i\} \tag{4.130}$$

$$\tau^2_{\text{lgt}} = \{\tau_i \in \tau_{\text{lgt}} \mid u_i > v_i\} \tag{4.131}$$

Algorithm 10: A_{lgt}: An algorithm to assign τ_{lgt} tasks

1 $\Gamma^1_{lgt^1} :=$ *fract-next-fit*(τ^1_{lgt}, m_1)
2 $\Gamma^2_{lgt^2} :=$ *fract-next-fit*(τ^2_{lgt}, m_2)
3 **if** $(\Gamma^1_{lgt^1} = \tau^1_{lgt} \wedge \Gamma^2_{lgt^2} = \tau^2_{lgt})$ **then** declare SUCCESS;
4 **if** $(\Gamma^1_{lgt^1} \neq \tau^1_{lgt} \wedge \Gamma^2_{lgt^2} \neq \tau^2_{lgt})$ **then** declare FAILURE;
5 **if** $(\Gamma^1_{lgt^1} \neq \tau^1_{lgt} \wedge \Gamma^2_{lgt^2} = \tau^2_{lgt})$ **then**
6 $\quad \Gamma^2_{lgt^1} := \tau^1_{lgt} \setminus \Gamma^1_{lgt^1}$
7 $\quad$ **if** (*fract-next-fit*$(\Gamma^2_{lgt^1}, m_2) = \Gamma^2_{lgt^1}$) **then**
8 $\quad\quad$ declare SUCCESS
9 $\quad$ **else**
10 $\quad\quad$ declare FAILURE
11 $\quad$ **end**
12 **end**
13 **if** $(\Gamma^1_{lgt^1} = \tau^1_{lgt} \wedge \Gamma^2_{lgt^2} \neq \tau^2_{lgt})$ **then**
14 $\quad \Gamma^1_{lgt^2} := \tau^2_{lgt} \setminus \Gamma^2_{lgt^2}$
15 $\quad$ **if** (*fract-next-fit*$(\Gamma^1_{lgt^2}, m_1) = \Gamma^1_{lgt^2}$) **then**
16 $\quad\quad$ declare SUCCESS
17 $\quad$ **else**
18 $\quad\quad$ declare FAILURE
19 $\quad$ **end**
20 **end**

4.6.7.1 The description of the A_{lgt} algorithm

The pseudo-code for assigning tasks in τ_{lgt} is shown in Algorithm 10 (which in turn uses the `fract-next-fit` subroutine, shown in Algorithm 11). The intuition behind the design of this algorithm is that, assuming a platform, $\pi^{(1+2\varepsilon)}$, first we assign tasks to processors on which they have a smaller utilization (line 1 and line 2 in Algorithm 10). Then, if there are remaining tasks, these are assigned to processors on which they have a larger utilizations (line 7 and line 15 in Algorithm 10).

4.6.7.2 Assignment analysis

First, we present some useful result in Lemma 28, obtained by relating the problem under consideration to the *fractional knapsack problem* (see Chapter 16.2 in [CLRS01]). This result will be used in Lemma 29. The fractional knapsack problem, an algorithm for this problem and the relation between the fractional knapsack problem and the problem under consideration was briefly discussed earlier in Section 4.3.3 (see page 87) in the context of FF-3C algorithm.

Informally, the relation between fractional knapsack problem and the task assignment problem on two-type platform can be described as follows. For a given problem instance in our scheduling problem, we can create an instance of a fractional knapsack problem as follows: (i) for each task

[9] While assigning tasks to type-1 processors, if a task cannot be assigned integrally on m_1'th processor (the last processor of type-1), then assign a fraction of that task such that m_1'th processor is fully utilized and assign the rest of the fraction to m_2'th processor (the last processor of type-2). This task is denoted by τ_f later in the proofs — in Section 4.6.8. This is not shown in the pseudo-code explicitly for ease of representation.

Algorithm 11: fract-next-fit(ts, ps): Next-fit bin-packing with fractional assignment of tasks

Input : *ts*: set of tasks; *ps*: set of processors
Output: set of tasks that were assigned successfully

1 If *ps* consists of type-1 (respectively, type-2) processors, then sort *ts* by decreasing v_i/u_i (respectively, increasing v_i/u_i). Use any order for processors *ps*, but maintain it during the execution of *fract-next-fit*.
2 Assign tasks using *wrap-around* technique[9].
3 Return the set of successfully assigned tasks.

in our scheduling problem, create a corresponding item in the fractional knapsack problem, (ii) the weight of an item in the fractional knapsack problem is the utilization of the corresponding task where the utilization here is taken for the processor on which the task executes fast and (iii) the value of an item in the fractional knapsack problem is how much lower the utilization of its corresponding task is when the task is assigned to the processor on which it executes fast as compared to its utilization if assigned to the processor on which it executes slowly. Informally speaking, we can see that if tasks could be split, then solving the fractional knapsack problem is equivalent to assigning tasks to processors so that the cumulative utilization of tasks is minimized. Again, informally speaking, we can then show that a task assignment minimizes the cumulative utilization of tasks assuming that (i) the cumulative utilization of tasks that are assigned to the processors on which they execute fast is sufficiently high and (ii) the tasks that are assigned to the processors where they execute fast has a higher ratio (v_i/u_i) than the ones that are not. We now express this formally in Lemma 28 and provide the proof (Lemma 12 is an adaptation of Lemma 5 in [ARB10]); the proof relies on the fractional knapsack algorithm whose pseudo-code is listed on page 87 as part of Lemma 11. Lemma 28 is a straight-forward adaptation of Lemma 12 presented in Section 4.3.3 (see page 12); however, for the sake of readability, we present the claim and the proof of the adapted version in detail now.

For the purpose of this lemma, let us define the following notations. Let the task set τ be partitioned into two disjoint subsets, τ^1 and τ^2. The set τ^1 consists of those tasks which run at least as fast on a type-1 processor as on a type-2 processor; τ^2 consists of all other tasks. In notation:

$$\tau = \tau^1 \cup \tau^2 \tag{4.132}$$

$$\forall \tau_i \in \tau^1 : u_i \leq v_i \tag{4.133}$$

$$\forall \tau_i \in \tau^2 : u_i > v_i \tag{4.134}$$

We now state the lemma and prove it.

Lemma 28. *Consider n tasks and a two-type platform conforming to the system model (and notation) of Section 2. Let x denote a number such that* $0 \leq x \leq m_1$.

*Let A*1 *denote a subset of* τ^1 *such that*

$$\sum_{\tau_i \in A1} u_i > m_1 - x \tag{4.135}$$

and for every pair of tasks $\tau_i \in A1$ *and* $\tau_j \in \tau^1 \setminus A1$ *it holds that* $\frac{v_i}{u_i} - 1 \geq \frac{v_j}{u_j} - 1$. *Let* $A2$ *denote* $\tau^1 \setminus A1$.

Let $B1$ *denote a subset of* τ^1 *such that*

$$\sum_{\tau_i \in B1} u_i \leq m_1 - x \tag{4.136}$$

Let $B2$ *denote* $\tau \setminus B1$. *It then holds that:*

$$\sum_{\tau_i \in A1} u_i + \sum_{\tau_i \in A2} v_i + \sum_{\tau_i \in \tau^2} v_i \leq \sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} v_i \tag{4.137}$$

Proof. Let us arbitrarily choose $A1$, $B1$ as defined. We will prove that this implies Inequality (4.137). Using Inequalities (4.135) and (4.136) we clearly get:

$$\sum_{\tau_i \in A1} u_i > \sum_{\tau_i \in B1} u_i \tag{4.138}$$

With this choice of $A1$ and $B1$, let us consider different instances of the fractional knapsack problem:

Instance1:
CAP = left-hand side of Inequality (4.138).
For each $\tau_i \in \tau$, create an item i with
$p_i = v_i - u_i$ and $w_i = u_i$
SUMVALUE_1=value of variable SUMVALUE when the algorithm in Lemma 11 (on page 86) terminates with Instance1 as input.

Instance2:
CAP = left-hand side of Inequality (4.138).
For each $\tau_i \in A1$, create an item i with
$p_i = v_i - u_i$ and $w_i = u_i$
SUMVALUE_2=value of variable SUMVALUE when the algorithm in Lemma 11 (on page 86) terminates with Instance2 as input.

Instance3:
CAP = right-hand side of Inequality (4.138).
For each $\tau_i \in B1$, create an item i with
$p_i = v_i - u_i$ and $w_i = u_i$
SUMVALUE_3=value of variable SUMVALUE when the algorithm in Lemma 11 (on page 86) terminates with Instance3 as input.

Instance4:
CAP = right-hand side of Inequality (4.138).
For each $\tau_i \in \tau$, create an item i with
$p_i = v_i - u_i$ and $w_i = u_i$

SUMVALUE_4=value of variable SUMVALUE when the algorithm in Lemma 11 (on page 86) terminates with Instance4 as input.

Observe that:

O1: In all four instances, it holds for each element that $\frac{p_i}{w_i} = \frac{v_i}{u_i} - 1$.

O2: Instance1 and Instance2 have the same capacity.

O3: Although Instance2 has a subset of the elements of Instance1, this subset is the subset of those elements with the largest p_i/w_i. (Follows from the definition of $A1$.)

O4: CAP in Instance2 is exactly the sum of the weights of the elements in $A1$.

O5: From O1-O4: SUMVALUE_2=SUMVALUE_1.

O6: Instance3 and Instance4 have the same capacity.

O7: Instance3 has a subset of the elements of Instance4.

O8: From O6 and O7: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_4$.

O9: Instance4 has smaller capacity than Instance1.

O10: Instance4 has the same elements as Instance1.

O11: From O9 and O10: $\text{SUMVALUE}_4 \leq \text{SUMVALUE}_1$.

O12: From O8 and O11: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_1$.

O13: From O12 and O5: $\text{SUMVALUE}_3 \leq \text{SUMVALUE}_2$.

Using O13 and the definitions of the instances and of $A1$ and $B1$ and observing that the capacity of Instance2 and Instance3 are set such that all elements in either instance will fit into the respective "knapsack", we obtain:

$$\sum_{\tau_i \in B1} (v_i - u_i) \leq \sum_{\tau_i \in A1} (v_i - u_i) \tag{4.139}$$

Now, observing that $\tau = \tau^1 \cup \tau^2 = B1 \cup B2$ gives us:

$$\sum_{\tau_i \in \tau^1} v_i + \sum_{\tau_i \in \tau^2} v_i = \sum_{\tau_i \in B1} v_i + \sum_{\tau_i \in B2} v_i$$

Substituting the value of $\sum_{i \in B1} v_i$ in Inequality (4.139) yields:

$$\sum_{\tau_i \in \tau^1} v_i + \sum_{\tau_i \in \tau^2} v_i - \sum_{\tau_i \in B2} v_i - \sum_{\tau_i \in B1} u_i \leq \sum_{\tau_i \in A1} v_i - \sum_{\tau_i \in A1} u_i$$

Rearranging terms, we get:

$$\sum_{\tau_i \in A1} u_i + \sum_{\tau_i \in \tau^1} v_i - \sum_{\tau_i \in A1} v_i + \sum_{\tau_i \in \tau^2} v_i \leq \sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} v_i$$

Exploiting $A2 = \tau^1 \backslash A1$ yields:

$$\sum_{\tau_i \in A1} u_i + \sum_{\tau_i \in A2} v_i + \sum_{\tau_i \in \tau^2} v_i \leq \sum_{\tau_i \in B1} u_i + \sum_{\tau_i \in B2} v_i$$

This is the statement of the lemma. Hence the proof. □

We now use the result of the previous lemma in the lemma presented below.

Lemma 29. *Let Γ^1_{lgt} and Γ^2_{lgt} be the subset of tasks from τ_{lgt} that are assigned by A_{lgt} to type-1 and type-2 processors, respectively. After assigning all the tasks from τ_{lgt}, we have:*

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Gamma^1_{\text{int}}} u_i + \sum_{\tau_i \in \Gamma^1_{\text{lgt}}} u_i \leq (1+2\varepsilon)m_1 \tag{4.140}$$

$$\text{and} \quad \sum_{\tau_i \in \Gamma^2_{\text{hvy}}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{int}}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}}} v_i \leq (1+2\varepsilon)m_2 \tag{4.141}$$

where

1. *all the tasks in $\tau_{\text{hvy}} \setminus \tau_{\text{int}}$ are assigned integrally*
2. *some tasks in τ_{int} are assigned fractionally and the rest are assigned integrally*
3. *some tasks in τ_{lgt} are assigned fractionally and the rest are assigned integrally*

Proof. Informally, the claim can be written as follows: if there exists a feasible non-migrative assignment for a task set τ on a two-type platform π then algorithms A_{hvy}, A_{int} and A_{lgt} succeed in assigning the tasks in τ as well but on a platform $\pi^{(1+2\varepsilon)}$ and with some tasks assigned fractionally. We already know from Lemma 27 that, after assigning the tasks in $\tau_{\text{hvy}} \setminus \tau_{\text{int}}$ and τ_{int} using algorithms A_{hvy} and A_{int}, respectively, the sum of the utilizations of the tasks assigned on type-1 (respectively, type-2) processors does not exceed $(1+\varepsilon)m_1$ (respectively, $(1+\varepsilon)m_2$).

Therefore, we need to show that after assigning the tasks in τ_{lgt} using algorithm A_{lgt}, the sum of the utilizations of the tasks assigned on processors of type-1 (respectively, type-2) does not exceed $(1+2\varepsilon)m_1$ (respectively, $(1+2\varepsilon)m_2$). An equivalent claim is that, after assigning tasks in $\tau_{\text{hvy}} \setminus \tau_{\text{int}}$ and τ_{int} using algorithms A_{hvy} and A_{int} respectively, if A_{lgt} fails to assign the tasks of τ_{lgt} (with fractional assignment of tasks allowed) on platform $\pi^{(1+2\varepsilon)}$ then there does not exist a feasible non-migrative assignment of the tasks in τ on platform π. Here, we prove this equivalent claim by contradiction. Assume that, there exists a feasible assignment, $\mathscr{H}_{\text{feas}}$, of τ on π but A_{lgt} fails to assign the tasks in τ_{lgt} on $\pi^{(1+2\varepsilon)}$ (after A_{hvy} and A_{int} successfully assigned the tasks of $\tau_{\text{hvy}} \setminus \tau_{\text{int}}$ and τ_{int}). Since A_{lgt} failed to assign these tasks, it must have declared FAILURE and we explore all possibilities for this to occur:

Failure on line 4 in Algorithm 10: From the case, we have $\Gamma^1_{\text{lgt}^1} \subset \tau^1_{\text{lgt}}$ and $\Gamma^2_{\text{lgt}^2} \subset \tau^2_{\text{lgt}}$. Therefore, when executing line 1 in A_{lgt} there was a task $\tau_{f_1} \in \tau^1_{\text{lgt}} \setminus \Gamma^1_{\text{lgt}^1}$ which could not be assigned to type-1 processors and similarly, when executing line 2 in A_{lgt} there was a task $\tau_{f_2} \in \tau^2_{\text{lgt}} \setminus \Gamma^2_{\text{lgt}^2}$ which could not be assigned to type-2 processors. Hence, we have:

$$\sum_{p \in P^1} U[p] + u_{f_1} > m_1(1+2\varepsilon) = m_1 + 2m_1\varepsilon \tag{4.142}$$

$$\text{and} \quad \sum_{p \in P^2} U[p] + v_{f_2} > m_2(1+2\varepsilon) = m_2 + 2m_2\varepsilon \tag{4.143}$$

where P^1 and P^2 denote the set of type-1 and type-2 processors respectively and $U[p]$ denotes the sum of the utilization of the tasks assigned on processor p.

Since $\tau_{f_1} \in \tau^1_{\text{lgt}} \overset{(4.130)}{\Rightarrow} \tau_{f_1} \in \tau_{\text{lgt}} \overset{(4.116)}{\Rightarrow} u_{f_1} < \varepsilon \leq m_1\varepsilon$ and analogously since $\tau_{f_2} \in \tau^2_{\text{lgt}}$, we know that $v_{f_2} < \varepsilon \leq m_2\varepsilon$. Using these on Expressions (4.142) and (4.143), we get

$$\sum_{p\in P^1} U[p] > m_1(1+\varepsilon) \tag{4.144}$$

$$\text{and} \quad \sum_{p\in P^2} U[p] > m_2(1+\varepsilon) \tag{4.145}$$

Observe that (i) the set of tasks that has been assigned on type-1 processors so far is $\Gamma^1_{\text{hvy}} \cup \Gamma^1_{\text{int}}$ and a strict subset of τ^1_{lgt}, and (ii) the set of tasks assigned on type-2 processors is $\Gamma^2_{\text{hvy}} \cup \Gamma^2_{\text{int}}$ and a strict subset of τ^2_{lgt}. Therefore, it holds from Expression (4.144) and (4.145) that:

$$\sum_{\tau_i\in\Gamma^1_{\text{hvy}}\cup\Gamma^1_{\text{int}}} u_i + \sum_{\tau_i\in\tau^1_{\text{lgt}}} u_i > m_1(1+\varepsilon) \tag{4.146}$$

$$\sum_{\tau_i\in\Gamma^2_{\text{hvy}}\cup\Gamma^2_{\text{int}}} v_i + \sum_{\tau_i\in\tau^2_{\text{lgt}}} v_i > m_2(1+\varepsilon) \tag{4.147}$$

Applying Expression (4.128) and (4.129) on Expression (4.146) and (4.147) respectively, yields:

$$(1+\varepsilon) \times \sum_{\tau_i\in\Phi^1_{\text{hvy}}\cup\Phi^1_{\text{int}}} u_i + \sum_{\tau_i\in\tau^1_{\text{lgt}}} u_i > m_1(1+\varepsilon) \tag{4.148}$$

$$(1+\varepsilon) \times \sum_{\tau_i\in\Phi^2_{\text{hvy}}\cup\Phi^2_{\text{int}}} v_i + \sum_{\tau_i\in\tau^2_{\text{lgt}}} v_i > m_2(1+\varepsilon) \tag{4.149}$$

Dividing Expression (4.148) by $1+\varepsilon$ and from trivial arithmetic $\sum_{\tau_i\in\tau^1_{\text{lgt}}} u_i > \frac{1}{1+\varepsilon} \times \sum_{\tau_i\in\tau^1_{\text{lgt}}} u_i$, we obtain:

$$\sum_{\tau_i\in\Phi^1_{\text{hvy}}\cup\Phi^1_{\text{int}}} u_i + \sum_{\tau_i\in\tau^1_{\text{lgt}}} u_i > m_1 \tag{4.150}$$

Analogously, Expression (4.149) yields:

$$\sum_{\tau_i\in\Phi^2_{\text{hvy}}\cup\Phi^2_{\text{int}}} v_i + \sum_{\tau_i\in\tau^2_{\text{lgt}}} v_i > m_2 \tag{4.151}$$

Summing Expressions (4.150) and (4.151) yields:

$$\sum_{\tau_i\in\Phi^1_{\text{hvy}}\cup\Phi^1_{\text{int}}\cup\tau^1_{\text{lgt}}} u_i + \sum_{\tau_i\in\Phi^2_{\text{hvy}}\cup\Phi^2_{\text{int}}\cup\tau^2_{\text{lgt}}} v_i > m_1 + m_2 \tag{4.152}$$

It is trivial to see that assigning all the tasks of τ^1_{lgt} and τ^2_{lgt} to type-1 and type-2 processors, respectively (as in the above expression), requires the minimum processing capacity. Hence, Expression (4.152) continues to hold for any other assignment of these tasks, implying that $\mathscr{H}_{\text{feas}}$ cannot be a feasible assignment, which leads to a contradiction.

Failure on line 10 in Algorithm 10: From the case, we have $\Gamma^1_{\text{lgt}^1} \subset \tau^1_{\text{lgt}}$ and $\Gamma^2_{\text{lgt}^2} = \tau^2_{\text{lgt}}$. Therefore, when executing line 7 in A_{lgt} there was a task $\tau_f \in \tau^1_{\text{lgt}} \setminus \Gamma^1_{\text{lgt}^1}$ which was attempted on type-2

processors but failed. Hence, we have:

$$\sum_{p \in P^2} U[p] + v_f > m_2(1+2\varepsilon) \tag{4.153}$$

We know that the tasks assigned to type-2 processors at this stage are $\Gamma^2_{\text{hvy}} \cup \Gamma^2_{\text{int}} \cup \Gamma^2_{\text{lgt}^2}$ and a strict subset of tasks from $\Gamma^2_{\text{lgt}^1}$ (line 7). Therefore, we can rewrite Expression (4.153) as:

$$\sum_{\tau_i \in \Gamma^2_{\text{hvy}} \cup \Gamma^2_{\text{int}} \cup \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i > m_2(1+2\varepsilon) - v_f \tag{4.154}$$

Since $\tau_f \in \tau^1_{\text{lgt}} \setminus \Gamma^1_{\text{lgt}^1}$, we know that $v_f < \varepsilon \leq m_2\varepsilon$. Using this on Expression (4.154) gives us:

$$\sum_{\tau_i \in \Gamma^2_{\text{hvy}} \cup \Gamma^2_{\text{int}}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i > m_2(1+\varepsilon) \tag{4.155}$$

Applying Expression (4.129) on Expression (4.155), we get:

$$(1+\varepsilon) \times \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i > m_2(1+\varepsilon) \tag{4.156}$$

Dividing Expression (4.156) by $1+\varepsilon$ and since from trivial arithmetic we know that $\sum_{\tau_i \in \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i > \frac{1}{1+\varepsilon} \times \sum_{\tau_i \in \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i$, we obtain:

$$\sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}^2} \cup \Gamma^2_{\text{lgt}^1}} v_i > m_2 \tag{4.157}$$

We also know that, when A_{lgt} executed line 1 (where it performed *fract-next-fit*), there must have been a task $\tau_{f_1} \in \tau^1_{\text{lgt}} \setminus \Gamma^1_{\text{lgt}^1}$ which was attempted on type-1 processors but failed to be assigned. Note that this task τ_{f_1} may be the same as τ_f mentioned above or it may be different. Because it was not possible to assign τ_{f_1} on type-1 processors, we know that:

$$\sum_{p \in P^1} U[p] + u_{f_1} > m_1(1+2\varepsilon) \tag{4.158}$$

We know that the tasks assigned to type-1 processors are $\Gamma^1_{\text{hvy}} \cup \Gamma^1_{\text{int}} \cup \Gamma^1_{\text{lgt}^1}$ and thus, we rewrite Expression (4.158) as:

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}} \cup \Gamma^1_{\text{int}} \cup \Gamma^1_{\text{lgt}^1}} u_i > m_1(1+2\varepsilon) - u_{f_1} \tag{4.159}$$

Since $\tau_{f_1} \in \tau^1_{\text{lgt}} \setminus \Gamma^1_{\text{lgt}^1}$, we have $u_{f_1} < \varepsilon \leq 2\varepsilon$. Hence, we can rewrite Expression (4.159) as:

$$\sum_{\tau_i \in \Gamma^1_{\text{hvy}} \cup \Gamma^1_{\text{int}}} u_i + \sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i > m_1(1+\varepsilon) \tag{4.160}$$

Applying Expression (4.128) on Expression (4.160), we get:

$$(1+\varepsilon) \times \sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}}} u_i + \sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i > m_1(1+\varepsilon) \tag{4.161}$$

Dividing Expression (4.161) by $1+\varepsilon$ and since (from trivial arithmetic we know that) $\sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i > \frac{1}{1+\varepsilon} \sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i$, we get:

$$\sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}}} u_i + \sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i > m_1 \tag{4.162}$$

Finally, Expression (4.162) can be rewritten as:

$$\sum_{\tau_i \in \Gamma^1_{\text{lgt}^1}} u_i > m_1 - \left(\sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i \right) \tag{4.163}$$

Let us now discuss the feasible assignment, $\mathscr{H}_{\text{feas}}$. Let Φ^1_{lgt} denote the set of tasks assigned to type-1 processors in $\mathscr{H}_{\text{feas}}$, excluding those in $\Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}}$. Similarly, let Φ^2_{lgt} denote the set of tasks assigned to type-2 processors in $\mathscr{H}_{\text{feas}}$, excluding those in $\Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}}$. Since, by assumption, $\mathscr{H}_{\text{feas}}$ succeeds in assigning all the tasks in τ to the processors, it holds that:

$$\sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{lgt}}} u_i \leq m_1 \tag{4.164}$$

$$\text{and} \sum_{\tau_i \in \Phi^2_{\text{hvy}}} v_i + \sum_{\tau_i \in \Phi^2_{\text{int}}} v_i + \sum_{\tau_i \in \Phi^2_{\text{lgt}}} v_i \leq m_2 \tag{4.165}$$

Expression (4.164) can be rewritten as:

$$\sum_{\tau_i \in \Phi^1_{\text{lgt}}} u_i \leq m_1 - \left(\sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i \right) \tag{4.166}$$

We can now reason about the inequalities we obtained about the assignment, $\mathscr{H}_{\text{feas}}$, and the one constructed by A_{lgt}. We can see that Expressions (4.163) and (4.166), with $x = \sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i$, ensure that the assumptions of Lemma 28 are true, given the ordering of tasks in τ^1_{lgt} during assignment over type-1 processors (line 1 in Algorithm 11), which ensures that $\forall \tau_i \in \Gamma^1_{\text{lgt}^1}, \forall \tau_j \in \Gamma^2_{\text{lgt}^1} : \frac{v_i}{u_i} \geq \frac{v_j}{u_j}$. By applying Lemma 28 with the following input:

- $T = \tau \setminus (\Phi^1_{\text{hvy}} \cup \Phi^2_{\text{hvy}} \cup \Phi^1_{\text{int}} \cup \Phi^2_{\text{int}})$,
- $T^1 = \tau^1_{\text{lgt}}$, $T^2 = \tau^2_{\text{lgt}} = \Gamma^2_{\text{lgt}^2}$,
- $x = \sum_{\tau_i \in \Phi^1_{\text{hvy}}} u_i + \sum_{\tau_i \in \Phi^1_{\text{int}}} u_i$,
- $A1$ is $\Gamma^1_{\text{lgt}^1}$; $\overset{(4.163)}{\Rightarrow} \sum_{\tau_i \in A1} u_i > m_1 - x$,

- $A2$ is $\Gamma^2_{\text{lgt}^1}$; Note that, for every pair of tasks, $\tau_i \in A1$ and $\tau_j \in A2$, it holds that, $\frac{v_i}{u_i} - 1 \geq \frac{v_j}{u_j} - 1$,
- $B1$ is Φ^1_{lgt}; $\overset{(4.166)}{\Rightarrow} \sum_{\tau_i \in B1} u_i \leq m_1 - x$,
- $B2$ is Φ^2_{lgt}.

we get:

$$\sum_{\tau_i \in \Gamma^1_{\text{lgt}1}} u_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}1}} v_i + \sum_{\tau_i \in \Gamma^2_{\text{lgt}2}} v_i \leq \sum_{\tau_i \in \Phi^1_{\text{lgt}}} u_i + \sum_{\tau_i \in \Phi^2_{\text{lgt}}} v_i$$

Adding $\sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}}} u_i + \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}}} v_i$ to both the sides in the above inequality yields

$$\sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}} \cup \Gamma^1_{\text{lgt}1}} u_i + \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}} \cup \Gamma^2_{\text{lgt}1} \cup \Gamma^2_{\text{lgt}2}} v_i \leq \sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}} \cup \Phi^1_{\text{lgt}}} u_i + \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}} \cup \Phi^2_{\text{lgt}}} v_i$$

Applying Expressions (4.164) and (4.165), we get:

$$\sum_{\tau_i \in \Phi^1_{\text{hvy}} \cup \Phi^1_{\text{int}} \cup \Gamma^1_{\text{lgt}1}} u_i + \sum_{\tau_i \in \Phi^2_{\text{hvy}} \cup \Phi^2_{\text{int}} \cup \Gamma^2_{\text{lgt}1} \cup \Gamma^2_{\text{lgt}2}} v_i \leq m_1 + m_2 \tag{4.167}$$

Applying Expressions (4.157) and (4.162) to Expression (4.167) yields:

$$m_1 + m_2 < m_1 + m_2$$

This is a contradiction.

Failure on line 18 in Algorithm 10: A contradiction results — proof analogous to the previous case.

We showed that all the cases where A_{lgt} declares FAILURE lead to a contradiction. Hence, the lemma holds. □

4.6.8 Integral assignment of τ_{int} and τ_{lgt} (Step 4)

We now discuss how to integrally assign the tasks from τ_{int} and τ_{lgt} that were fractionally assigned by algorithms A_{int} and A_{lgt}, respectively. We also show that, if there is a feasible non-migrative assignment of the given task set on a given two-type platform then our PTAS succeeds in finding such a feasible non-migrative assignment of τ as well but on a platform in which every processor is $1 + 3\varepsilon$ times faster.

4.6.8.1 The description of A_{fract} algorithm

The algorithm, A_{fract}, works as follows:

1. Copy the assignment (made by A_{hvy}, A_{int} and A_{lgt} on $\pi^{(1+2\varepsilon)}$) onto a faster platform, $\pi^{(1+3\varepsilon)}$.

2. On this platform, $\pi^{(1+3\varepsilon)}$, assign the task split between any two processors p_1 and p_1+1 of type-1 entirely on to processor p_1, where $1 \leq p_1 < m_1$; similarly, assign the task split between any two processors p_2 and p_2+1 of type-2 entirely on to processor p_2, where $1 \leq p_2 < m_2$.

3. Assign the task split between m_1'th processor of type-1 and m_2'th processor of type-2 (i.e, task τ_f) to any of these processors.

4.6.8.2 Assignment analysis

Theorem 18. *If there exists a feasible non-migrative assignment of a task set τ on a two-type platform π then our PTAS algorithm,* PTAS_{NF}*, (which uses* A_{hvy}*,* A_{int}*,* A_{lgt} *and* A_{fract}*, in sequence) succeeds as well in finding a feasible non-migrative assignment of τ but on a platform $\pi^{(1+3\varepsilon)}$.*

Proof. We know from Lemma 29 that, if there exists a feasible non-migrative assignment of τ on π then the three algorithms A_{hvy}, A_{int} and A_{lgt} described in Section 4.6.4 to Section 4.6.7 succeed in assigning tasks in τ (with a subset of tasks from τ_{int} and τ_{lgt} fractionally assigned) on $\pi^{(1+2\varepsilon)}$. As a consequence, we have:

$$\forall p \in \pi^{(1+2\varepsilon)} : U[p] \leq 1+2\varepsilon \tag{4.168}$$

We also know that, in such an assignment, as a consequence of using the wrap-around technique in A_{int} and A_{lgt}, it holds that:

- at most $m_1 - 1$ tasks are *split* between processors of type-1 with one task split between each pair of consecutive processors; let the set Γ^1_{split} denote these fractional tasks.

- at most $m_2 - 1$ tasks are *split* between processors of type-2 with one task split between each pair of consecutive processors; let the set Γ^2_{split} denote these fractional tasks.

- at most one task (from τ_{lgt}) is *split* between processors of type-1 and type-2; let $\tau_f \in \tau_{\text{lgt}}$ denote this task that must be split between the m_1'th processor of type-1 and the m_2'th processor of type-2.

- the rest of the tasks are integrally assigned to either type-1 or type-2 processors.

Let $\tau^1_{p_1,p_1+1} \in \Gamma^1_{\text{split}}$ denote the task split between the p_1'th and the (p_1+1)'th processors of type-1 where $1 \leq p_1 < m_1$. Analogously, let $\tau^2_{p_2,p_2+1} \in \Gamma^2_{\text{split}}$ denote the task split between the p_2'th and the (p_2+1)'th processors of type-2 where $1 \leq p_2 < m_2$.

To prove the theorem, we need to show that A_{fract} succeeds in *integrally* assigning all the fractional tasks on $\pi^{(1+3\varepsilon)}$.

On Step 1, A_{fract} copies the assignment from $\pi^{(1+2\varepsilon)}$ onto a faster platform $\pi^{(1+3\varepsilon)}$. After this step,

$$\forall p \in \pi^{(1+3\varepsilon)} : U[p] \leq 1+2\varepsilon \tag{4.169}$$

Since $\Gamma^1_{\text{split}} \subseteq \{\tau^1_{\text{int}} \cup \tau_{\text{lgt}}\}$, $\Gamma^2_{\text{split}} \subseteq \{\tau^2_{\text{int}} \cup \tau_{\text{lgt}}\}$, we have:

$$(4.116), (4.121) \Rightarrow \forall \tau_i \in \Gamma^1_{\text{split}} : u_i < \varepsilon \tag{4.170}$$

$$(4.116), (4.122) \Rightarrow \forall \tau_i \in \Gamma^2_{\text{split}} : v_i < \varepsilon \tag{4.171}$$

On Step 2, A_{fract} assigns the split tasks integrally. So, $\forall p_1 \in$ type-1 of $\pi^{(1+3\varepsilon)}$, it moves the fraction of the task, $\tau^1_{p_1,p_1+1}$, that is assigned to (p_1+1)'th processor of type-1 to p_1'th processor of type-1. After this re-assignment, it follows from Expression (4.169) and Expression (4.170) that:

$$\forall p_1 \in \text{type-1 of } \pi^{(1+3\varepsilon)} \ \wedge \ p_1 \neq m_1 : \quad U[p_1] \leq 1+3\varepsilon \tag{4.172}$$

$$\text{if } p_1 \in \text{type-1 of } \pi^{(1+3\varepsilon)} \ \wedge \ p_1 = m_1 : \quad U[p_1] \leq 1+2\varepsilon \tag{4.173}$$

Analogously, $\forall p_2 \in$ type-2 of $\pi^{(1+3\varepsilon)}$, it moves the fraction of the task, $\tau^2_{p_2,p_2+1}$, that is assigned to (p_2+1)'th processor of type-2 to p_2'th processor of type-2. After this re-assignment, it follows from Expression (4.169) and Expression (4.171) that:

$$\forall p_2 \in \text{type-2 of } \pi^{(1+3\varepsilon)} \ \wedge \ p_2 \neq m_2 : \quad U[p_2] \leq 1+3\varepsilon \tag{4.174}$$

$$\text{if } p_2 \in \text{type-2 of } \pi^{(1+3\varepsilon)} \ \wedge \ p_2 = m_2 : \quad U[p_2] \leq 1+2\varepsilon \tag{4.175}$$

Finally, the task, τ_f, that is split between the m_1'th processor of type-1 and the m_2'th processor of type-2 remains to be integrally assigned. Since $\tau_f \in \tau_{\text{lgt}}$, it holds that, $u_f < \varepsilon$ and $v_f < \varepsilon$. From Expression (4.173) and (4.175), it follows that, task τ_f can be *integrally* assigned to either m_1'th or m_2'th processor. Hence, after integrally assigning this task, we obtain:

$$\forall p \in \pi^{(1+3\varepsilon)} : U[p] \leq 1+3\varepsilon \tag{4.176}$$

Since Expression (4.176) is a necessary and sufficient schedulability condition for EDF on a uniprocessor of capacity $1+3\varepsilon$, the assignment of τ on $\pi^{(1+3\varepsilon)}$ returned by our algorithm, PTAS_{NF}, is a feasible assignment. Hence, the proof. □

4.6.9 Average-case performance evaluations

After studying the theoretical bound, i.e., the speed competitive ratio of our algorithm, PTAS_{NF}, we evaluate its average-case performance and compare it with prior state-of-the-art algorithm, PTAS_{LP}. For this purpose, we look at the following aspects: (i) how much faster processors our algorithm needs *in practice* in order to obtain a feasible non-migrative assignment of a task set compared to PTAS_{LP}? (i.e., comparison of the necessary multiplication factors) and (ii) how fast our algorithm runs compared to PTAS_{LP}? Also, we look at (iii) how much pessimism is there in the theoretically derived performance bound of our algorithm, PTAS_{NF}?

In order to answer these questions, we performed two sets of experiments. In the first set of experiments, we compared the average-case performance of our algorithm, PTAS_{NF}, with PTAS_{LP}.

Recall that the speed competitive ratio of both these algorithms depend on the value of the input parameter, ε. Hence, we evaluated the average-case performance of both the algorithms for different values of ε. We observed that, in our evaluations with randomly generated task sets, our algorithm requires significantly smaller necessary multiplication factor than PTAS_{LP}. We also observed that our algorithm runs faster by orders of magnitude compared to PTAS_{LP}. Overall, PTAS_{NF} exhibits a better average-case performance by outperforming the prior state-of-the-art algorithm, PTAS_{LP}. In the second set of evaluations, in order to see how much pessimism our theoretical analysis has, we evaluated only PTAS_{NF} for different values of ε. We observed that, it performs significantly better in simulations by requiring much smaller processor speedup than indicated by its theoretical bound of $1+3\varepsilon$. We now discuss both the cases in detail.

4.6.9.1 First set of evaluations: Comparison with the state-of-the-art

We implemented both the algorithms, PTAS_{NF} and PTAS_{LP}, using C on Windows XP on an Intel Core2 (2.80 GHz) machine. For PTAS_{LP}, which relies on solving linear programming formulations, we used one of the state-of-the-art LP/ILP solvers, IBM ILOG CPLEX [IBM12].

The algorithm, PTAS_{LP}, proposed in [WBB13], for partitioning the task set on heterogeneous multiprocessors, can be summarized as follows:

- The given task set is transformed into another task set by "rounding up" the utilizations to some specific values that are determined based on the value of ε.
- The tasks in the transformed task set are grouped into *big* and *small* tasks based on their utilizations. For big tasks, different *feasible patterns* are generated using dynamic programming.
- For a feasible pattern, the task assignment problem (for both big and small tasks) is formulated as an ILP and then relaxed to LP. The LP formulation is solved using an LP solver. If a feasible solution is returned by the LP solver then go to next step else consider the next feasible pattern and repeat this step.
- Using the values of the indicator variables from the solution returned by the LP solver, construct a bipartite graph and define a *fractional matching*. In the bipartite graph, one set of nodes represent the tasks and another set of nodes represent the processors. The fractional matching represents how much fraction of a task (indicated by the value of the indicator variable) is assigned to the processor to which it is connected in the graph.
- Using any maximum cardinality bipartite matching algorithm (e.g., Ford-Fulkerson algorithm — see pp. 714 in [CLRS01]), find an *integral matching* from the fractional matching. This integer matching gives the non-migrative assignment of the tasks to the processors.

We denote the necessary multiplication factor of PTAS_{NF} and PTAS_{LP} by NMF_{NF} and NMF_{LP}, respectively. For different values of ε, for many task sets, we assess the average-case performance

of both the algorithms by measuring their (i) necessary multiplication factors and (ii) average running times.

The problem instances (number of tasks, their utilizations and the number of processors of each type) are generated randomly. Each problem instance had at most 10 tasks and at most 2 processors of each type. We generated 5000 task sets[10], denoted as $\{\tau^{(1)}, \tau^{(2)}, \ldots, \tau^{(5000)}\}$, which we transformed into "critically feasible non-migrative task sets". Recall that, a *critically feasible task set* is a task set which is non-migrative feasible on a given two-type platform but rendered non-migrative infeasible if all the task utilizations (i.e., both u_i and v_i of each task) are increased by an arbitrarily small factor.

To obtain a critically feasible non-migrative task set, $\tau^{(k)}_{\text{crit}}$, from a randomly generated task set, $\tau^{(k)}$, $k \in [1, 5000]$, we perform the assignment of tasks in $\tau^{(k)}$ by formulating the problem as an MILP as shown in Figure 4.17 and feeding it to a solver (such as IBM ILOG CPLEX) which outputs Z, the utilization of the most utilized processor. Then, we multiply all the task utilizations by $1/Z$ (which is equivalent to multiplying the speeds of every processor by Z) and repeatedly feed it back to the solver until $0.99 < Z \leq 1$, which gives us $\tau^{(k)}_{\text{crit}}$.

Minimize Z subject to the following constraints:

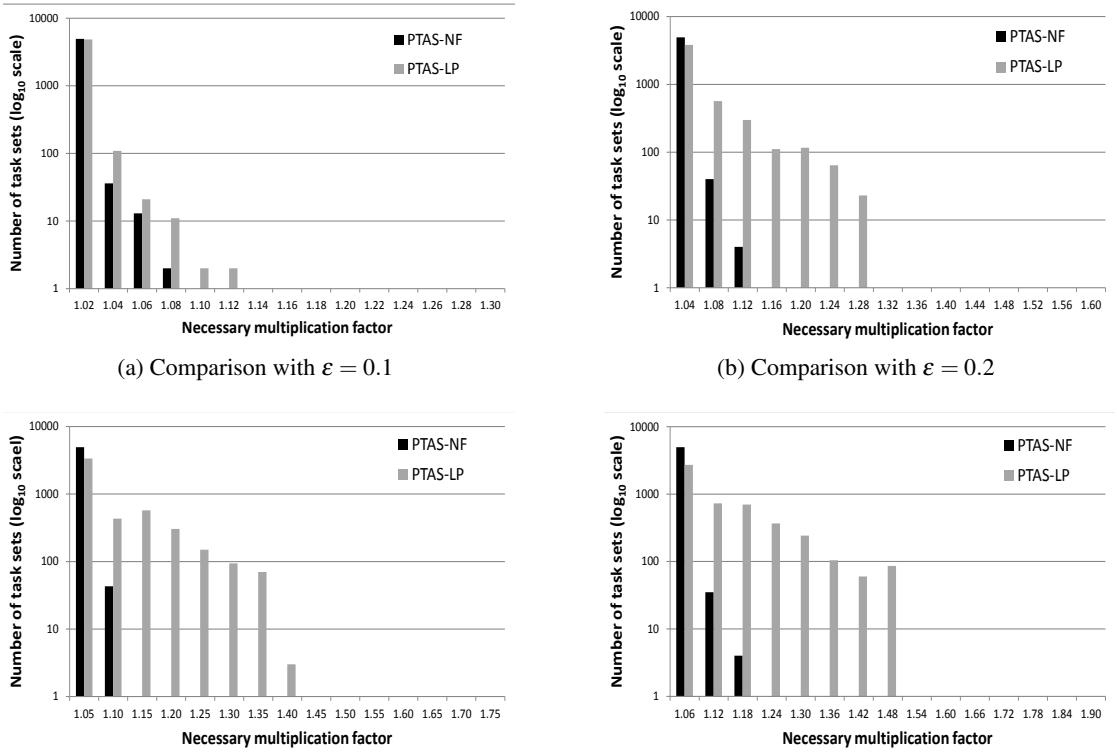
I1.	$\sum_{j=1}^{m} x_{ij} = 1$	$(i = 1, 2, \cdots, n)$
I2.	$\sum_{i=1}^{n} (x_{ij} \cdot u_{ij}) \leq Z$	$(j = 1, 2, \cdots, m)$
I3.	x_{ij} are non-negative integers	$(i = 1, 2, \cdots, n)$
		$(j = 1, 2, \cdots, m)$

Figure 4.17: Mixed Integer Linear Programming formulation to find a feasible partitioning of $\tau^{(k)}$ on π — x_{ij} are indicator variables and u_{ij} are utilizations.

For a given ε, for each critically feasible non-migrative task set, $\tau^{(k)}_{\text{crit}}$, and algorithm, $\mathscr{A}$ (where $\mathscr{A}$ is either PTAS_{NF} or PTAS_{LP}), we measure the necessary multiplication factor, denoted by $\text{NMF}^{(k)}_{\mathscr{A}}(\varepsilon)$. The procedure to obtain the necessary multiplication factor of an algorithm for a given set of critically feasible non-migrative task sets was discussed (along with the pseudo-code — see Algorithm 1 on page 70 for the pseudo-code) earlier in Section 3.8 (see page 68) in Chapter 3. This procedure is repeated for 5000 critically feasible task sets. Algorithm 1 is repeatedly called with different values of ε, specifically, we used $\varepsilon = 0.1, 0.2, 0.25$ and 0.3.

With this procedure, we obtain the histograms of NMFs for both the algorithms for different values of ε. Figure 4.18 shows the histograms. As can be seen from Figure 4.18b, in the evaluations with $\varepsilon = 0.2$, the NMF_{NF} never exceeded 1.12 which is only 20% away from the optimal value of 1.0 compared to its upper bound of $1 + 3\varepsilon = 1.60$, i.e., $\frac{1.12-1.0}{1.6-1.0} \times 100 = 20\%$, whereas NMF_{LP} is as high as 1.30 which is 60% away from the optimal value of 1.0 compared to its upper bound of $\frac{1+\varepsilon}{1+\varepsilon} = 1.50$, i.e., $\frac{1.3-1.0}{1.5-1.0} \times 100 = 60\%$. Overall, in simulations, PTAS_{NF} requires

[10]Since PTAS_{LP} has a huge run-time complexity as it heavily relies on solving LP formulation (i.e., it solves LP formulation for every feasible pattern generated by the dynamic programming till it succeeds), the number of problem instances and the size of each problem instance were set to relatively smaller values. For example, in the simulations with $\varepsilon = 0.3$, PTAS_{LP} took $48h$ to determine the NMF of 5000 critically feasible task sets.



(a) Comparison with $\varepsilon = 0.1$

(b) Comparison with $\varepsilon = 0.2$

(c) Comparison with $\varepsilon = 0.25$

(d) Comparison with $\varepsilon = 0.3$

Figure 4.18: Comparison of *necessary multiplication factors* of PTAS_{NF} and PTAS_{LP} for different values of ε (if an algorithm has low NMF for many task sets then the algorithm is said to perform well).

much smaller necessary multiplication factor compared to PTAS_{LP} in order to find a feasible non-migrative assignment. As can be seen from Figure 4.18, the observations for other values of ε follow the same trend.

We also measure the average running times of both the algorithms for different values of ε. In these evaluations, the necessary multiplication factor is set to $1+3\varepsilon$ for PTAS_{NF} and to $\frac{1+\varepsilon}{1+\varepsilon}$ for PTAS_{LP}. This ensures that both the algorithms always succeed in finding a feasible non-migrative assignment for a given task set in a *single run* and hence the evaluations are not biased to give advantage to any of the algorithms. In our evaluations with 5000 task sets, as can be seen in Table 4.13, for $\varepsilon = 0.1$, for each task set, PTAS_{NF}, has an average running time of 128 μs whereas the PTAS_{LP} has an average running time of 6583384 $\mu s \approx 6.6$ s. Hence, for $\varepsilon = 0.1$, for each task set, PTAS_{NF} runs approximately 50000 times faster compared to PTAS_{LP}. This factor is even higher for other values of ε as illustrated in Table 4.13.

To summarize, in our evaluations, PTAS_{NF} exhibits a better average-case performance by requiring significantly smaller processor speedup for finding a feasible non-migrative assignment and by running orders of magnitude faster compared to PTAS_{LP}. Overall, PTAS_{NF} outperforms prior state-of-the-art algorithm, PTAS_{LP}.

	Measured average running times		Ratio of average running times
Value of ε	PTAS_{NF}	PTAS_{LP} **[WBB13]**	
0.10	128.57	6583384.71	51204
0.15	45.43	6914127.72	152192
0.20	18.40	4449061.29	241796
0.25	10.48	1564060.39	149242
0.30	7.17	465894.09	64978

Table 4.13: Comparison of average running times of PTAS_{NF} and PTAS_{LP} (in μs) for different values of ε.

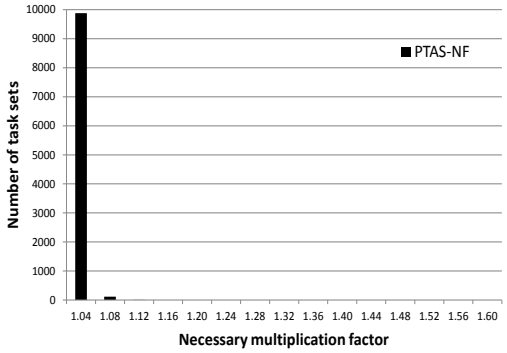
4.6.9.2 Second set of evaluations: Performance of PTAS_{NF} for different values of ε

In order to understand how much pessimism is there in the analysis of PTAS_{NF}, we evaluated its average-case performance for different values of ε. In this set of evaluations, we chose larger number of problem instances with each problem instance being more complex. We generated 10000 critically feasible non-migrative task sets where each task set had at most 25 tasks and at most 3 processors of each type. Since we do not run PTAS_{LP} (which takes much longer to output a solution as it relies on solving several linear programming formulations) in this batch of evaluations, we could increase the problem instances and size of each problem compared to the previous set of evaluations. Then, for different values of ε, we ran PTAS_{NF} on these 10000 critically feasible non-migrative task sets and obtained histograms of NMF_{NF}. Figure 4.19 shows these histograms. As can be seen from Figure 4.19c, for example, in the experiments with $\varepsilon = 0.3$, for almost 98% of the task sets, the NMF_{NF} did not exceed 1.06 which is approximately 7% of its theoretical bound (i.e., $1+3\varepsilon = 1.90$), for the remaining 2% of the task sets, the factor did not exceed 1.12 which is approximately 13% of its theoretical bound. Thus, in the evaluations, for the vast majority of task sets, our algorithm requires much smaller processor speedup than indicated by its speed competitive ratio. As can be seen from Figure 4.19, the observations for other values of ε follow the same trend.

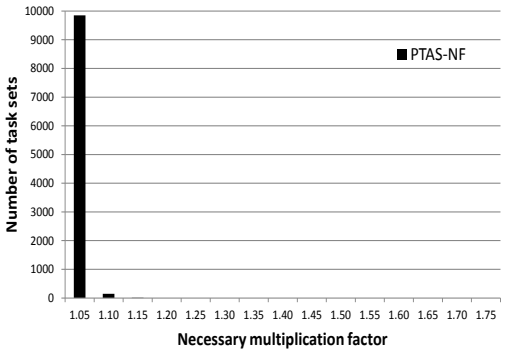
Hence, PTAS_{NF} performs significantly better in simulations than indicated by its theoretical bound.

4.6.10 Summary

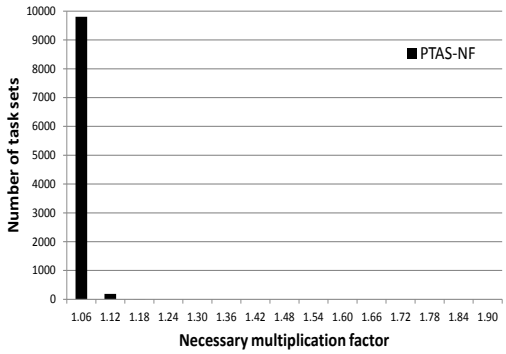
In this section, for the problem of non-migrative task assignment on two-type heterogeneous multiprocessors, we presented a polynomial-time approximation scheme, PTAS_{NF}. This algorithm uses a combination of dynamic programming technique and bin-packing heuristic to output the task assignment. We showed that the speed competitive ratio of PTAS_{NF} is $1+3\varepsilon$ against an equally powerful non-migrative adversary. The PTAS_{NF} algorithm is shown to outperform the prior state-of-the-art PTAS (referred to as PTAS_{LP}) in terms of the time-complexity. We also evaluated and compared the average-case performance of our PTAS_{NF} algorithm with PTAS_{LP}. This is done by generating random task sets and converting them into critically feasible non-migrative



(a) Performance with $\varepsilon = 0.2$



(b) Performance with $\varepsilon = 0.25$



(c) Performance with $\varepsilon = 0.3$

Figure 4.19: Performance evaluation of PTAS_{NF} for different values of ε in terms of the necessary multiplication factor.

task sets and then measuring the necessary multiplication factor of the algorithm for each of those critically feasible task sets. In our evaluations, we observed that, for the vast majority of task sets, (i) our PTAS_{NF} algorithm outperforms PTAS_{LP} by requiring significantly smaller processor speedup for finding a feasible non-migrative assignment and by running orders of magnitude faster and (ii) PTAS_{NF} algorithm performed significantly better by succeeding in finding a feasible non-migrative assignment with necessary multiplication factor much smaller than the speed competitive ratio.

For the problem of non-migrative task assignment on heterogeneous multiprocessors, although many PTAS existed before this work, all of them had prohibitively large constants in their respective run-time expressions which limited their practical significance severely. This work designed a polynomial time approximation scheme which is efficient to be usable in practice.

4.7 Conclusions and Discussions

We first summarize the four non-migrative algorithms presented in this chapter and then discuss the advantages and disadvantages of each of these algorithms.

4.7.1 Summary

In this chapter, we considered the problem of non-migrative scheduling of implicit-deadline sporadic tasks on two-type heterogeneous multiprocessors. This problem consists of two sub-problems: (i) assigning tasks to processors and (ii) scheduling the tasks on each processor. The second sub-problem is well-understood and even optimal scheduling algorithms (for example, EDF) exist to schedule the tasks assigned to a processor. Hence, assuming that such an optimal scheduling algorithm is used to schedule the tasks on each processor, the challenge is to find a feasible assignment of tasks to processors. Thus, the problem of *non-migrative task scheduling* under consideration translates to the problem of *non-migrative task assignment*. This problem is shown to be NP-Complete in the strong sense. For this problem, we proposed four polynomial time-complexity algorithms with different speed competitive ratio and time-complexity.

The first algorithm, FF-3C, has a low-degree polynomial time-complexity and has provably good performance. Specifically, FF-3C (i) has a time-complexity of $O(n \cdot \max(m, \log n))$, where n denotes the number of tasks and m denotes the number of processors, (ii) has a speed competitive ratio of $1+\alpha \leq 2$ against equally powerful non-migrative adversary, where the parameter $0 < \alpha \leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than 1. Note that, the speed competitive ratio of FF-3C reaches 2 when $\alpha = 1$. FF-3C has a superior performance to state-of-the-art. This is because (i) FF-3C has the same speed competitive ratio as algorithms in [Bar04b, Bar04c, LST90] (whose performances have been proven against a non-migrative adversary) but with a better time-complexity and (ii) among the algorithms with speed competitive ratio proven against a more powerful fully-migrative adversary [CSV12], FF-3C offers the best speed competitive ratio and (iii) compared to PTAS algorithms [HS76, JP99, WBB13] that offer better speed competitive ratios (for lower values of ε) but whose practical significance is severely limited as they incur a very high time-complexity (i.e., exponential in number processors or exponential in $1/\varepsilon$), FF-3C offers a significantly lower (i.e., low-degree polynomial) time-complexity.

Several extensions to FF-3C algorithms were also presented; these offer the same speed competitive ratio and time-complexity but in addition, they offer improved average-case performance. Via experiments with randomly generated task sets, we compare the average-case performance of FF-3C algorithm (and its variants) and two established state-of-the-art algorithms (and variations of the latter) [Bar04b, Bar04c]. We evaluate algorithms based on (i) the average running time and (ii) the necessary multiplication factor. Overall FF-3C and its variants compare favorably to the state-of-the-art. In particular, in our evaluations, one of the variants of FF-3C, namely FF-4C-COMB, runs 12000 to 160000 times faster and has significantly smaller necessary multiplication factor than the state-of-the-art [Bar04b, Bar04c] for the vast majority of the task sets.

Further, FF-3C and its variants rely on bin-packing heuristics (such as *first-fit*) to output the task assignment. To the best of our knowledge, no previous algorithm exists to assign real-time tasks on heterogeneous multiprocessors that makes use of bin-packing heuristics and that has a provably good performance. Therefore, this is the first work to show how bin-packing heuristics can be used to design a task assignment algorithm for two-type heterogeneous multiprocessors with a finite speed competitive ratio.

The second algorithm, SA-P, is an extension of (the intra-migrative) SA algorithm. SA-P also has a low-degree polynomial time-complexity and has a provably good performance. Specifically, SA-P (i) has a time-complexity of $O(n \log n)$, where n denotes the number of tasks and (ii) has a speed competitive ratio of $1 + \alpha \leq 2$ against a more powerful intra-migrative adversary, where the parameter $0 < \alpha \leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than 1. Note that, in the worst-case (i.e., when $\alpha = 1$), the speed competitive ratio of SA-P is 2. SA-P has superior performance compared to state-of-the-art since SA-P has (i) the same speed competitive ratio as FF-3C [ARB10, RAB13] and other algorithms in [Bar04b, Bar04c, LST90] but with a stronger adversary and also a better time-complexity, (ii) compared to the algorithms whose speed competitive ratio have been proven against an adversary with a migration model of intra-migrative or greater power [CSV12], SA-P offers the best speed competitive ratio and (iii) compared to PTAS algorithms [HS76, JP99, WBB13] that offer better speed competitive ratios (for lower values of ε) but whose practical significance is severely limited as they incur a very high time-complexity (i.e., exponential in number processors or exponential in $1/\varepsilon$), SA-P offers a significantly lower (i.e., low-degree polynomial) time-complexity.

Further, in the average-case performance evaluations, SA-P performed well. The average-case performance evaluations were done by generating random task sets, converting them to critically feasible intra-migrative task sets and then running SA-P for each of them in order to compute the necessary multiplication factor of SA-P. In these evaluations, we observed that, for the vast majority of task sets, SA-P algorithm performed significantly better by succeeding in finding a feasible non-migrative assignment with a necessary multiplication factor much smaller than its speed competitive ratio.

The third algorithm, LPC, has a polynomial time-complexity and offers the following guarantee. If there exists a feasible non-migrative assignment of a task set on a two-type platform then, LPC succeeds as well in finding such a feasible non-migrative assignment for the same task set but on a platform in which each processor is 1.5 times faster and there are 3 additional processors, compared to the platform used by the adversary. LPC has superior performance compared to state-of-the-art for systems with large number of processors, since LPC offers a better (i.e., smaller) speed competitive ratio than all the previous algorithms. This is because (i) for systems with large number of processors, the additional 3 processors that LPC requires become negligible and hence its speed competitive ratio tends to 1.5 which is better than the algorithms in [Bar04b, Bar04c, LST90, CSV12, RAB13, RABN12] and (ii) compared to PTAS algorithms [HS76, JP99, WBB13] which incur a very high time-complexity, LPC offers a lower (i.e., polynomial) time-complexity.

Further, LPC relies on solving linear program formulation with *cutting planes*. Although task assignment schemes with provably good performance have previously been developed by relaxing an Mixed Integer-Linear Program (MILP) to a Linear Program (LP) (e.g., [Bar04c, Bar04b, LST90]) and cutting planes have been used to solve (M)ILP in different efforts, no work in the past has shown how cutting planes can be used to improve the speed competitive ratio of provably good algorithms for assigning real-time tasks to processors. Hence, to the best of our knowledge, this work is the first one to show how cutting planes can be used to improve the speed competitive ratio of algorithms for assigning real-time tasks to two-type heterogeneous multiprocessors.

The fourth and final non-migrative algorithm that we presented, namely PTAS_{NF}, has a polynomial time-complexity and has a speed competitive ratio of $1+3\varepsilon$ against equally powerful non-migrative adversary. This algorithm relies on dynamic programming techniques and bin-packing heuristics to output the feasible non-migrative task assignment. PTAS_{NF} has superior performance compared to prior state-of-the-art since (i) compared to algorithms proposed in [Bar04c, Bar04b, LST90, RAB13, RABN12, RA13], it has a better speed competitive ratio and (ii) compared to previous PTASs [HS76, JP99, WBB13], it has a better time-complexity. Specifically, compared to PTAS_{LP} of [WBB13], our PTAS has a much better run-time complexity, in the sense that, it is efficient enough to be usable in practice, whereas the practical significance of PTAS_{LP} is severely limited as it has a very high run-time complexity since the constants in the run-time expression of PTAS_{LP} are prohibitively large.

Further, in the average-case performance evaluations, PTAS_{NF} performed better as well. The average-case performance evaluations were done by generated random task sets, converting them to critically feasible non-migrative task sets and then running PTAS_{NF} and PTAS_{LP} for each of them in order to compute their respective necessary multiplication factors. In our evaluations, PTAS_{NF} exhibits a better average-case performance than PTAS_{LP} by succeeding to find a feasible non-migrative assignment with a significantly smaller necessary multiplication factor for a large number of task sets and by running orders of magnitude faster compared to PTAS_{LP}. Overall, PTAS_{NF} outperforms PTAS_{LP}. Also, in evaluations with different values of ε, for the vast majority of task sets, PTAS_{NF} succeeds to obtain a feasible non-migrative assignment with a much smaller necessary multiplication factor compared to its speed competitive ratio.

4.7.2 Discussion

We now briefly discuss the advantages and disadvantages of each of these algorithms when compared against each other. The four algorithms can be summarized as shown in Figure 4.20.

As can be seen from Figure 4.20, SA-P algorithm completely dominates FF-3C algorithm. This is because although SA-P has approximately the same time-complexity as FF-3C, it has a speed competitive ratio which is better than that of FF-3C. This is due to the fact that although the speed competitive ratio of both FF-3C and SA-P is given by $1+\alpha \leq 2$, the speed competitive ratio of FF-3C is quantified against an *equally powerful* non-migrative adversary whereas the speed competitive ratio of SA-P is quantified against a *more powerful* intra-migrative adversary. As illustrated in Table 2.2 on page 32 in Chapter 2, it can be concluded that, the speed competitive

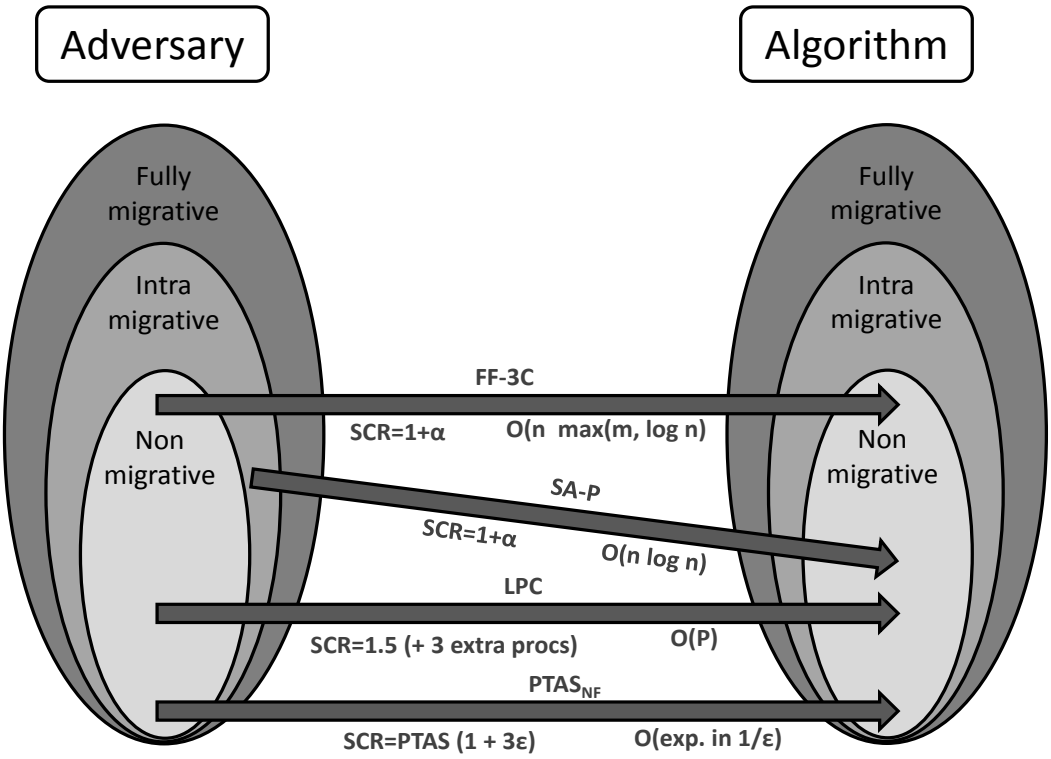


Figure 4.20: Summary of the four non-migrative algorithms presented in this chapter. Here, SCR denotes the "speed competitive ratio", n denotes the number of tasks, m denotes the number of processors, $\varepsilon > 0$ is an input parameter to the algorithm and α is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

ratio of SA-P is better than that of FF-3C. Hence, SA-P dominates FF-3C and therefore we ignore FF-3C algorithm in the rest of the discussion.

For the other three algorithms, i.e., SA-P, LPC and PTAS_{NF}, it is difficult to establish such a clear dominance relationship (mainly because their respective speed competitive ratios are quantified in different ways). Hence, we only highlight some of the advantages and disadvantages of these algorithms when compared to each other.

SA-P**.** This algorithm has the following advantages compared to LPC and PTAS_{NF} algorithms. It is easy to implement and it has a low-degree polynomial time-complexity. However, its speed competitive ratio is generally higher than that of the LPC and the PTAS_{NF} algorithms. Specifically, its speed competitive ratio (i) is higher than that of the LPC for systems with large processors and (ii) is higher than that of the PTAS_{NF} for small values of ε.

LPC**.** Its advantage is in the speed competitive ratio that it offers when compared to SA-P and PTAS_{NF} algorithms. For systems with a large number of processors, the speed competitive ratio of LPC tends to 1.5. Hence, (i) compared to algorithms whose speed competitive ratios have been proven against an adversary with a migration model of intra-migrative or greater power (i.e., SA-P), LPC offers the best speed competitive ratio and (ii) compared to PTAS_{NF}, it has a better speed competitive ratio for any value of ε greater than about 0.17. Its disadvantage is that it has a

higher time-complexity than SA-P since it relies on solving an LP formulation.

PTAS_{NF}**.** It has the advantage of being able to partition the task set in polynomial time, to any desired degree of accuracy (which can be tuned using the input parameter ε) which the other two algorithms are not capable of. However, its disadvantage is that, for smaller values of ε, it may take significantly longer time to finish execution when compared to the other two algorithms (as its time-complexity depends exponentially on $1/\varepsilon$).

Chapter 5

Shared Resource Scheduling on Two-type Heterogeneous Multiprocessors

5.1 Introduction

In this chapter, we consider the problem of scheduling a set of tasks to meet all deadlines on a two-type heterogeneous multiprocessor platform where tasks may access shared resources — we refer to this problem as *shared resource scheduling problem*. Tasks typically share a processor but in many computer systems, tasks also share other resources such as data structures, sensors, etc. Tasks must operate on such shared resources in a *mutually exclusive* manner while accessing the resource, that is, at all times, when a job of a task holds a resource, no other job of any task can hold that resource. Even on a single processor, the sharing of such resources can have a profound effect on timing behavior as witnessed by the near failure of the NASA mission, Mars Pathfinder, because the resource-sharing protocol in the operating system was not enabled [Jon97]. Scheduling real-time tasks that share resources on a multiprocessor platform is even more complex. Our goal in this work is to design an algorithm for scheduling real-time tasks that share resources (apart from processors) on two-type heterogeneous multiprocessors so as to meet all the deadlines.

Despite the trend of increase in the usage of chips having two different types of processors (for example, see [AMD13a, Int13c, Nvi12]), the state-of-art in real-time scheduling theory for heterogeneous multiprocessors is under-developed. The reasons include (i) processors typically sharing low-level hardware resources (e.g., caches, interconnects), which makes task execution times interdependent and (ii) dispatching limitations, for example, some processors depend on another processor for dispatching [GHF$^+$06]. Such idiosyncratic challenges must be addressed on a case-by-case basis, accounting for the particularities of the architecture. The state-of-the-art does offer some general ideas on analyzing shared low-level hardware resources [DAN$^+$11, LSL$^+$09, LYGY10, PSC$^+$10, RAEP07, SNE10] and scheduling co-processors [Ble07, GAB02, LR10]. Ultimately though, the dependency of the task execution time on the processor-type is

what inherently complicates the design of scheduling algorithms for heterogeneous platforms. Therefore, designers using heterogeneous multiprocessors today and in the future can benefit from scheduling theories that consider this inherent property. And for this reason, in this work, we design an algorithm (considering this property) to schedule tasks that share resources (in addition to processors) on two-type heterogeneous multiprocessors and prove its speed competitive ratio.

Problem Statement. We consider the problem of scheduling a task set of implicit-deadline sporadic tasks to meet all deadlines on a two-type heterogeneous multiprocessor platform where each task may access *at most* one resource from the set of given shared resources. The multiprocessor platform has m_1 processors of type-1 and m_2 processors of type-2. The set of shared resources is denoted by R. For each task τ_i, there is a resource in R such that for each job of task τ_i, during one phase of its execution, the job requests to hold this resource *exclusively*, with the interpretation that, (i) the job makes a single request to hold this resource and (ii) at all times, when a job of task τ_i holds the resource, no other job holds this resource. Each job of task τ_i may request this resource at most once during its execution. A job is allowed to migrate when it requests a resource and when it releases the resource but a job is not allowed to migrate at other times. One can show that the problem under consideration is NP-Complete in the strong sense (by mapping an instance of the 3-PARTITION problem to an instance of the problem under consideration). Hence, in this work, we aim to design a scheduling algorithm for this problem with a finite speed competitive ratio.

Related Work. The problem of scheduling independent sporadic tasks on heterogeneous multiprocessors has been studied in the past [HS76, JP99, LST90, Bar04c, Bar04b, WBB13, RAB13, RABN12, RN12a, CSV12] but without considering the case in which tasks share resources. One might assign tasks to processors and apply a resource-sharing protocol conceived for identical multiprocessors (e.g. D-PCP [RSL88]). However, protocols such as D-PCP are less effective in minimizing priority inversion when used in heterogeneous multiprocessors as they are in minimizing priority inversion when used in identical multiprocessors. For example, a task holding a shared resource may be executing on a processor where it runs slowly — causing large *priority inversion* to other tasks and poor schedulability. Therefore, a resource-sharing protocol for heterogeneous platforms ought to be cognizant of the execution speed of each task on each processor. It should also provide a finite bound on how much worse it performs, compared to an optimal scheme.

Contributions and Significance of the work discussed in this chapter. This paper presents an algorithm, FF-3C-vpr, for scheduling tasks that share resources on a two-type heterogeneous multiprocessor. It has a low-degree time complexity and offers the following guarantee. If a task set can be scheduled on a two-type platform to meet all deadlines by an optimal scheme that allows a task to migrate only when requesting or releasing a resource then FF-3C-vpr succeeds to meet all deadlines as well with the same restriction on the migration but given a platform in which every processors is $4+6\cdot\left\lceil\frac{|R|}{\min(m_1,m_2)}\right\rceil$ times faster.

We believe the significance of this work is as follows. To the best of our knowledge, for the problem of scheduling task sets that share resources on two-type heterogeneous multiprocessors, no previous algorithm is known to exist with a provably good performance and hence this is the

first result in this direction.

Organization of the chapter. The rest of the chapter is organized as follows. Section 5.2 describes the system model and states the assumptions. Section 5.3 discusses the hardness of the shared resource scheduling problem. Section 5.4 gives the main idea of our new algorithm, FF-3C-vpr. Section 5.5 lists notations and a few results used later to prove the speed competitive ratio of FF-3C-vpr and also discusses virtual processors which are integral in designing FF-3C-vpr. The algorithm, FF-3C-vpr, is presented in Section 5.6 along with the proof of its speed competitive ratio. Finally, Section 5.7 presents some concluding remarks.

5.2 System model and assumptions

We consider the problem of scheduling implicit-deadline sporadic tasks that share resources on a two-type heterogeneous multiprocessor platform with *restricted migration* (defined later). The system is specified as follows:

- **Platform (Π):** The two-type platform consists of m processors of which $m_1 \geq 1$ processors are of type-1 and $m_2 \geq 1$ processors are of type-2.
- **Shared Resources (R):** A set R of $|R|$ resources that tasks share in addition to processors.
- **Task set (τ):** The task set consists of *n implicit-deadline sporadic tasks*.
- **Minimum inter-arrival time, WCET and Utilization:** The minimum inter-arrival time of a task τ_i is denoted by T_i. Its worst-case execution time on a processor of type-t (where $t \in \{1,2\}$) is denoted by C_i^t, and its utilization by U_i^t.

We make the following assumptions:

- **Sharing the resources:** Each task may request *at most one* resource from R (known at design time) and *at most once* by each job of that task.
- **Virtual processors:** *Virtual processors* are logical constructs, used as task assignment targets by our algorithm. A virtual processor vp_i acts equivalent to a (physical) processor of the same type with (scaled) speed $\frac{1}{f}$ — and we assume that it can be "emulated" on a physical processor of the same type (of speed 1), using no more than $\frac{1}{f}$ of its processing capacity[1].
- **Restricted migration:** A job of a task may migrate to another processor during execution only when it requests a resource; it must then migrate back to the original processor upon

[1] One intuitive way of achieving this is by dividing time to short slots of length S and using $\frac{1}{f} \cdot S$ time units in each slot to serve the workload of vp_i. By selecting S, we can then make the speed of the emulated processor arbitrarily close to $\frac{1}{f}$ (and in practice, S need rarely be impractically short) [BA09]. In strict terms, a sufficient condition for emulating m_1 type-1 virtual processors from VP_{AC} onto m_1 type-1 physical processors is: $\sum_{\substack{vp_i \in VP_{AC} \\ vp_i \text{ is type}-1}} V_i < m_1$, where V_i is the speed of virtual processor vp_i (and similarly for type-2 processors in VP_{AC} and for VP_{B} processors). For more details (including how to tradeoff spare processing capacity for longer S), see [BA09].

releasing the resource. A task cannot migrate at any other instant. We call this model *restricted migration*. Migration between processors of any type is allowed.

Because of the *restricted migration* model, we can categorize the execution of a task into different phases described next. For a *job of a task τ_i that accesses a resource*, we categorize the execution into three phases as follows. Let phase-A execution of a job of task τ_i denote the execution the job performs from when it arrives until it requests the resource. Let phase-B execution of a job of task τ_i denote the execution the job performs from when it requests the resource until it releases that resource. Let phase-C execution of a job of task τ_i denote the execution the job performs from when it releases the resource until it finishes execution. For a *job of a task that does not accesses a resource*, we categorize its execution into a single phase, phase-A, which denotes the entire execution of the job, i.e., the execution the job performs from when it arrives until it finishes execution.

5.3 The hardness of the shared resource scheduling problem

In this section, we show that the problem under consideration, i.e., the problem of shared resource scheduling with "restricted migration" (i.e., a job can only migrate when it requests or releases a resource) on a two-type heterogeneous multiprocessor is NP-Complete in the strong sense. We denote this problem as HET2-RES-MIG-REQ-REL and is stated in Figure 5.1.

In order to show this, we will first consider a special case of this problem which is denoted as HET2-RES-MIG-REQ-REL-PHASE-A-EXEC — see Figure 5.2. We will show that this problem is NP-complete in the strong sense. It then follows that HET2-RES-MIG-REQ-REL is NP-complete in the strong sense as well.

For showing that the HET2-RES-MIG-REQ-REL-PHASE-A-EXEC problem is NP-Complete in the strong sense, we make use of the 3-PARTITION problem. The 3-PARTITION problem is shown in Figure 5.3 and it is well-known that this problem is NP-Complete in the strong sense [GJ78].

Lemma 30. *The HET2-RES-MIG-REQ-REL-PHASE-A-EXEC problem described in Figure 5.2 is NP-Complete in the strong sense.*

Proof. Note that, in this problem, since (i) there is only execution in phase-A (that is, phase-B and phase-C do not exist) and (ii) the execution time of phase-A is independent of the type of the processor to which it is assigned, the problem is equivalent to finding a mapping of tasks to processors so that on each processor, the cumulative utilization of all tasks assigned to the processor is at most 100%, where it is assumed that the utilization of a task is given by $u_i^t \stackrel{\text{def}}{=} CA_i^t/T_i$ (where $t \in \{1,2\}$).

In order to show that a problem is NP-Complete in the strong sense, we need to: (1) show that the problem is in NP, (2) transform a problem which is NP-Complete in the strong sense to the problem under consideration and (3) show that the transformation (of Step (2)) can be

HET2-RES-MIG-REQ-REL	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type heterogeneous multiprocessor platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. There is a set R of resources. Each job of task τ_i requests a resource in set R during one phase of its execution and each job may request this resource at most once during its execution. The resource accessed by the jobs of a task is determined by the task. The execution of a job of task τ_i has three phases: phase-A, phase-B and phase-C. Phase-A is the execution of a job from when it arrives until it requests the resource. Phase-B is the execution of a job from when it requests the resource until it has released the resource. Phase-C is the execution of a job from when it has released the resource until it has finished execution. Let CA_i^t denote an upper bound on the execution time of phase-A of a job of task τ_i if this phase-A execution is assigned to a processor of type-t (where $t \in \{1,2\}$). Analogously, let CB_i^t (resp., CC_i^t) denote an upper bound on the execution time of phase-B (resp., phase-C) of a job of task τ_i if this phase-B (resp., phase-C) execution is assigned to a processor of type-t (where $t \in \{1,2\}$).
Problem	Find an assignment of phase-A: $f : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ phase-B: $g : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ phase-C: $h : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ such that with this assignment the task set is schedulable.

Figure 5.1: The shared resource scheduling problem considered in this work.

done in polynomial time. We now show these for HET2-RES-MIG-REQ-REL-PHASE-A-EXEC problem.

1. It is straightforward to see that the problem belongs to NP. As a *certificate*, we take the task assignment on each processor. To check whether the given assignment in fact satisfies $\sum_{i:f(i)=j} u_i^t \leq 1$ for every processor $j \in$ type-t of π (where $t \in \{1,2\}$) is obviously possible in polynomial time.

2. We now transform the 3-PARTITON problem (which is NP-Complete in the strong sense) to the above decision problem. Given an instance $c_1, c_2, \ldots, c_{3m}$ of the 3-PARTITION problem, we transform it into an instance of HET2-RES-MIG-REQ-REL-PHASE-A-EXEC problem by computing utilizations of tasks as follows:

$$\forall \tau_i \in \tau, \forall t \in \{1,2\} : u_i^t = \frac{c_i}{B} \tag{5.1}$$

   It can be seen that the assignment of these $3m$ tasks on m processors is possible if and only if I can be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j : \sum_{i \in I_j} c_i = B$.

3. Finally, it can be easily seen that the transformation from 3-PARTITION to HET2-RES-MIG-REQ-REL-PHASE-A-EXEC using Expression 5.1 is possible in polynomial time.

HET2-RES-MIG-REQ-REL-PHASE-A-EXEC	
Instance	A task set τ of n implicit-deadline sporadic tasks and a two-type heterogeneous multiprocessor platform π of m processors of which m_1 processors are of type-1 and m_2 processors are of type-2. There is a set R of resources. Each job of task τ_i requests a resource in set R during one phase of its execution and each job may request this resource at most once during its execution. The resource accessed by the jobs of a task is determined by the task. The execution of a job of task τ_i has three phases: phase-A, phase-B and phase-C. Phase-A is the execution of a job from when it arrives until it requests the resource. Phase-B is the execution of a job from when it requests the resource until it has released the resource. Phase-C is the execution of a job from when it has released the resource until it has finished execution. Let CA_i^t denote an upper bound on the execution time of phase-A of a job of task τ_i if this phase-A execution is assigned to a processor of type-t (where $t \in \{1,2\}$). Analogously, let CB_i^t (resp., CC_i^t) denote an upper bound on the execution time of phase-B (resp., phase-C) of a job of task τ_i if this phase-B (resp., phase-C) execution is assigned to a processor of type-t (where $t \in \{1,2\}$). Assume that: (i) no task accesses any resource in R, i.e., $\forall \tau_i \in \tau, \forall t \in \{1,2\}$: $CB_i^t = CC_i^t = 0$ and (ii) $\forall \tau_i \in \tau : CA_i^1 = CA_i^2$
Problem	Find an assignment of phase-A: $f : \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,m\}$ such that with this assignment the task set is schedulable.

Figure 5.2: A restricted version of the shared resource scheduling problem considered in this work (which is shown in Figure 5.1).

Hence the proof. □

Theorem 19. *The HET2-RES-MIG-REQ-REL problem described in Figure 5.1 is NP-Complete in the strong sense.*

Proof. Follows from Lemma 30 and the fact that the HET2-RES-MIG-REQ-REL-PHASE-A-EXEC problem is a restricted version of the HET2-RES-MIG-REQ-REL problem. □

5.4 Overview of our approach

The key to our approach is to distinguish between three *phases* in the execution of a task and make different scheduling provisions for each of them (Figure 5.4):

- **Phase-A** of a task spans from its arrival until it requests a shared resource.
- In its **Phase-B**, the task is holding (or waiting for) the shared resource.
- In its **Phase-C**, the task has released the resource and executes till its completion.

The main structure of our approach is as follows:

3-PARTITION PROBLEM	
Instance	A list of $3m$ integers $I = \{c1, c2, \ldots, c_{3m}\}$ where $\forall i : c_i \geq 2$ and a bound B such that $\sum_{i=1}^{3m} c_i = mB$ and $\forall i : B/4 < c_i < B/2$.
Question	Can I be partitioned into m subsets $I_1, I_2, \ldots, I_m$ such that $\forall j : \sum_{i \in I_j} c_i = B$.

Figure 5.3: The 3-partitioning problem which is known to be NP-Complete in the strong sense [GJ78].

1. Split the task execution into phases A, B and C — in essence create three subtasks out of it. The phase-B and phase-C subtasks of a task "arrive" (i.e. first become ready to execute) at a (respective) fixed time offset to the arrival of the respective phase-A subtask. This ensures that subtasks "inherit" the inter-arrival time of the original task and exhibit no arrival jitter.

2. Use m physical processors to create a set VP of virtual processors, formed by disjoint sets, VP_{AC} and VP_{B} (i.e., $\text{VP}_{\text{AC}} \bigcup \text{VP}_{\text{B}} = \text{VP}$ and $\text{VP}_{\text{AC}} \bigcap \text{VP}_{\text{B}} = \emptyset$).

3. Both Phase-A and phase-C of a task are assigned to a single virtual processor, $vp_j \in \text{VP}_{\text{AC}}$. Phase-B of the same task is assigned to a virtual processor, $vp_k \in \text{VP}_{\text{B}}$.

4. The phase-A and phase-C subtasks of a task are scheduled using preemptive EDF on their assigned virtual processor in VP_{AC} along with the other subtasks that are assigned on that virtual processor; the phase-B subtask is scheduled using non-preemptive EDF on its assigned virtual processor in VP_{B} along with the other phase-B tasks that are assigned on that virtual processor — as a way of serializing accesses to shared resources[2].

Steps 1-3 are performed at *design time*; step 4 is carried out at *run time*. Despite using virtual processors, our algorithm by-construction ensures that the "restricted migration" assumption is not violated — discussed in Section 5.5.2 and Section 5.6. Subtasks corresponding to task phases are assigned *constrained* deadlines, i.e., not exceeding their inter-arrival time (inherited from the original task).

We use some notations and well-known results while proving the speed competitive ratio for our algorithm. We present those notations and results in the next section.

5.5 Few notations and useful results

5.5.1 Notations

Let $\Pi(m_1, m_2)$ denote a two-type heterogeneous multiprocessor platform having m_1 processors of type-1 and m_2 processors of type-2. Let $\Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle$ denote a platform in which the speed

[2]Observe that implementing multiple virtual processors on the same physical processor might in practice involve frequent "context-switching" between those. Yet, whenever a physical processor "context-switches" between a phase-B virtual processor and some other virtual processor mapped to it, this does not violate the semantics of non-preemptive scheduling on the phase-B virtual processor because we are only interested (for the purposes of resource access serialization) in ensuring that phase-B subtasks never preempt each other – and this property is not violated.

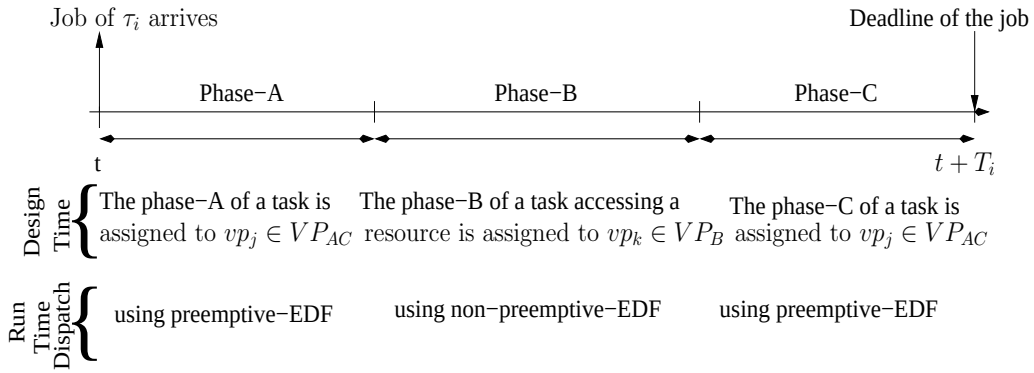


Figure 5.4: Three execution phases of a job along with the design time (task assignment) and run time (task dispatching) decisions of FF-3C-vpr.

of a type-1 and type-2 processor is respectively, s_1 and s_2 times the speed of a type-1 and type-2 processor in platform $\Pi(m_1, m_2)$, where s_1 and s_2 are positive real-numbers.

Let the predicate $sched(A, \tau, \Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle)$ signify that a task set τ that do not share resources *meets all its deadlines* if scheduled by an algorithm A on a platform $\Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle$. The term *meets all its deadlines* in this and other predicates means 'meets deadlines for every possible valid arrival of jobs of tasks in τ'.

We use $sched(\text{nmo}, \tau, \Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle)$ to signify that there exists a *non-migrative-offline* preemptive schedule which meets all deadlines for the specified system. Here, *non-migrative* schedule refers to a schedule in which all the jobs of a task execute on the same processor to which the task is assigned. In this predicate (and others), the term *offline* means that the schedule (i) can contain inserted idle times and (ii) can be generated using knowledge of future task arrival times (irrespective of whether such knowledge is available in practice).

We use $sched(\text{rmo}, \tau, R, \Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle)$ to denote a predicate to signify that there exists a *restricted-migration-offline* preemptive schedule which meets all deadlines for the specified system when tasks in τ share resources in R. As mentioned in Section 5.2, each task requests at most one resource from R and each job of that task may request that resource at most once during its execution. The term "restricted migration" has the same meaning as discussed in Section 5.2.

Similarly, $sched(A, \tau, R, \Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle)$ signifies that the task set τ in which tasks are "sharing the resources" (see Section 5.2) in R *meets all its deadlines* when scheduled by an algorithm A with "restricted migration" (see Section 5.2) on platform $\Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle$.

Finally, in the above predicates, the suffix -δ (where applicable) to a non-migrative scheduling algorithm (or algorithm class) implies that the schedulability of τ (other than just being established via some exact test) must additionally be ascertainable via a (potentially pessimistic) *density-based* uniprocessor schedulability test. This means that for the subset τ' of subtasks assigned on every type-t processor of speed V, it has to hold that $\sum_{i \in \tau'} \delta_i^t \leq V$, where $\delta_i^t = \frac{C_i^t}{D_i^t}$ is the *density*, C_i^t is the execution time (w.r.t. a processor of speed 1) and D_i^t is the deadline of a task τ_i on a type-t processor.

On a type-t processor: Let $C^t_{i,1}$ denote the execution time of a task τ_i before requesting a resource, i.e. in its phase-A. Let $C^t_{i,2(k)}$ denote the execution time of τ_i while holding resource R^k (where k is the index of the resource used by τ_i), i.e. in its phase-B. Let $C^t_{i,3}$ denote the execution time of a task τ_i after releasing the resource, i.e. in its phase-C. Note that $\forall \tau_i \in \tau$: $C^t_{i,1}+C^t_{i,2(k)}+C^t_{i,3}=C^t_i$.

We derive three new *constrained-deadline* (denoted by D^t_i) *sporadic task* sets (i.e., for each task, its deadline is less than or equal to its minimum inter-arrival time) namely, $TD_A(\tau)$, $TD_{B,R^k}(\tau)$ and $TD_C(\tau)$ from the given implicit-deadline sporadic task set τ by modifying the parameters of the tasks in τ. Intuitively, (i) a task $\tau_{i(A)} \in TD_A(\tau)$ represents phase-A execution of $\tau_i \in \tau$, (ii) a task $\tau_{i(B)} \in TD_{B,R^k}(\tau)$ represents phase-B execution of $\tau_i \in \tau$, accessing the resource R^k and (iii) a task $\tau_{i(C)} \in TD_C(\tau)$ represents phase-C execution of $\tau_i \in \tau$.

$TD_A(\tau)$, $TD_{B,R^k}(\tau)$ and $TD_C(\tau)$ are defined as follows – for each task $\tau_i \in \tau$:

$$\tau_{i(A)} = \{T_{i(A)} = T_i, \quad D^t_{i(A)} = \frac{C^t_{i,1}}{C^t_i} \cdot \frac{T_i}{2}, \quad C^t_{i(A)} = C^t_{i,1}\}$$

$$\tau_{i(B)} = \{T_{i(B)} = T_i, \quad D^t_{i(B)} = \frac{T_i}{2}, \quad C^t_{i(B)} = C^t_{i,2(k)}\}$$

$$\tau_{i(C)} = \{T_{i(C)} = T_i, \quad D^t_{i(C)} = \frac{C^t_{i,3}}{C^t_i} \cdot \frac{T_i}{2}, \quad C^t_{i(C)} = C^t_{i,3}\}$$

Note that $D^A_i + D^B_i + D^C_i \leq T_i$. This is essential as it ensures that if the subtasks $\tau_{i(A)}, \tau_{i(B)}$ and $\tau_{i(C)}$ derived from the task τ_i meet their deadlines then τ_i meets its deadline as well. Also, observe that $TD_A(\tau)$ and $TD_C(\tau)$ are derived such that the densities of $\tau_{i(A)}$ and $\tau_{i(C)}$ are twice the utilization of $\tau_i \in \tau$. For example,

$$\forall \tau_{i(A)} \in TD_A(\tau): \quad \delta^t_{i(A)} = \frac{C^t_{i(A)}}{D^t_{i(A)}} = \frac{C^t_{i,1}}{\frac{C^t_{i,1}}{C^t_i} \cdot \frac{T_i}{2}} = \frac{2C^t_i}{T_i} = 2U^t_i \text{ of } \tau_i \in \tau \tag{5.2}$$

Analogous expression holds for tasks in $TD_C(\tau)$.

5.5.2 Useful results

Lemma 31 and Lemma 32 (re-)state the speed competitive ratios of FF-3C (which is 2 – see Th. 1 in [ARB10]) and of uniprocessor non-preemptive EDF. Recall from the discussion of FF-3C algorithm in Section 4.3 of Chapter 4 that, the speed competitive ratio of the FF-3C algorithm is 2 (see Theorem 9 on page 100). It is shown in [AE10] that the speed competitive ratio of uniprocessor non-preemptive EDF algorithm is 3 (see Lemma 1 in [AE10]). Recall that FF-3C is a *non-migrative* algorithm for assigning implicit-deadline sporadic tasks (that do not share resources) on two-type heterogeneous multiprocessors.

Lemma 31. *(From Theorem 9 in Section 4.3 of Chapter 4)*
$sched(nmo, \tau, \Pi(m_1, m_2)) \Rightarrow sched(FF\text{-}3C, \tau, \Pi(m_1, m_2) \cdot \langle 2, 2 \rangle)$

Lemma 32. *(From Lemma 1 in [AE10])*
$sched(nmo\text{-}np, \tau, \Pi(1,0)) \Rightarrow sched(nm\text{-}np\text{-}EDF, \tau, \Pi(1,0) \cdot \langle 3,3 \rangle)$

Note that (i) the heterogeneous multiprocessor platform $\Pi(1,0)$ with one processor of type-1 in Lemma 32 is trivially a uniprocessor platform and (ii) Lemma 32 also holds for platform $\Pi(0,1)$ with one processor of type-2.

Lemma 33 states that if a task is non-preemptive EDF-schedulable on a uniprocessor, it is also non-preemptive non-migrative EDF-schedulable on a platform that has an additional processor.

Lemma 33. $sched(nm\text{-}np\text{-}EDF, \tau, \Pi(1,0) \cdot \langle 3,3 \rangle) \Rightarrow sched(nm\text{-}np\text{-}EDF, \tau, \Pi(1,1) \cdot \langle 3,3 \rangle)$

The intuition behind Lemma 33 is that if the additional (type-2) processor is kept idle during the scheduling then τ is schedulable on the original (type-1) processor. It is trivial to see that the lemma also holds if the computing platform in the left-hand side predicate is replaced with $\Pi(0,1) \cdot \langle 3,3 \rangle$.

Lemma 34. *(Combining Lemma 32 and Lemma 33)*
$sched(nmo\text{-}np, \tau, \Pi(1,0)) \Rightarrow sched(nm\text{-}np\text{-}EDF, \tau, \Pi(1,1) \cdot \langle 3,3 \rangle)$

The following lemma states that if implicit-deadline task set τ is non-migrative offline schedulable on platform $\Pi(m_1, m_2)$ then constrained-deadline sporadic task set $TD_A(\tau)$ derived from τ (as described in Section 5.5.1) is also non-migrative schedulable (e.g., using preemptive-EDF) on platform $\Pi(m_1, m_2) \cdot \langle 2,2 \rangle$ *and additionally* this can be established via use of a (potentially pessimistic) *density-based* schedulability test. It is easy to see that the claim holds since the density of a task $\tau_{i(A)}$ in $TD_A(\tau)$ is always twice the utilization of the corresponding task τ_i in τ.

Lemma 35. $sched(nmo, \tau, \Pi(m_1, m_2)) \Rightarrow sched(nmo\text{-}\delta, TD_A(\tau), \Pi(m_1, m_2) \cdot \langle 2,2 \rangle)$

Proof. Let us assume that a non-migrative-offline feasible schedule exists for task set τ on platform $\Pi(m_1, m_2)$. So, there must exist a schedule in which the following holds $\forall t \in 1,2$:

$$\forall p \text{ of type-t} \in \Pi(m_1, m_2) : \sum_{\tau_i \in \tau[p]} U_i^t \leq 1 \tag{5.3}$$

where $\tau[p]$ denotes the set of tasks assigned to processor p of type-t. Now, we show that there also exists a non-migrative-offline feasible schedule for task set $TD_A(\tau)$ on platform $\Pi(m_1, m_2) \cdot \langle 2,2 \rangle$.

We know that, for every task $\tau_i \in \tau$, there exists a task, $\tau_{i(A)} \in TD_A(\tau)$. We also know from Expression (5.2) that, $\forall t \in 1,2$, it holds that: $\forall \tau_{i(A)} \in TD_A(\tau) : \delta_{i(A)}^t = 2U_i^t$ of $\tau_i \in \tau$. Let us assign the tasks in $TD_A(\tau)$ to processors in $\Pi(m_1, m_2) \cdot \langle 2,2 \rangle$ as follows: if the task, $\tau_i \in \tau$, is assigned to a processor, $p \in \Pi(m_1, m_2)$, then assign the corresponding task, $\tau_{i(A)} \in TD_A(\tau)$, to the corresponding processor, $p \in \Pi(m_1, m_2) \cdot \langle 2,2 \rangle$. From the fact that this assignment of $TD_A(\tau)$ (which is identical to the assignment of τ) is made on a platform twice faster (on which the densities of tasks will be halved) and from Expressions (5.2) and (5.3), we obtain: $\forall t \in 1,2$:

$$\forall p \text{ of type-t} \in \Pi(m_1, m_2) \cdot \langle 2,2 \rangle : \sum_{\tau_{i(A)} \in TD_A(\tau)[p]} \delta_{i(A)}^t \leq 1 \tag{5.4}$$

The above inequality corresponds to the density-based schedulability test, on every processor p of type-t $\in \Pi(m_1,m_2)\cdot\langle 2,2\rangle$, for non-migrative preemptive EDF scheduling policy. Thus, the task set $TD_A(\tau)$ is also non-migrative-offline schedulable on platform $\Pi(m_1,m_2)\cdot\langle 2,2\rangle$. □

The following lemma largely follows from Lemma 31 — obtained by applying density-based schedulability test, using faster platforms and using the reasoning provided in Lemma 35.

Lemma 36. *(From Lemma 31 and Lemma 35)*
$sched(\text{nmo-}\delta, TD_A(\tau), \Pi(m_1,m_2)\cdot\langle 2,2\rangle) \Rightarrow sched(\text{FF-3C-}\delta, TD_A(\tau), \Pi(m_1,m_2)\cdot\langle 4,4\rangle)$

Proof. Assume that the left-hand side predicate $sched(\text{nmo-}\delta, TD_A(\tau), \Pi(m_1,m_2)\cdot\langle 2,2\rangle)$ of the claim holds true. Then, since the density of every subtask in $TD_A(\tau)$ is twice the utilization of the corresponding task in τ, for reason similar to the one provided in the previous lemma, the predicate $sched(\text{nmo}, \tau, \Pi(m_1,m_2))$ holds true as well. In that case, we know from Lemma 31 that the predicate $sched(\text{FF-3C}, \tau, \Pi(m_1,m_2)\cdot\langle 2,2\rangle)$ holds true also. But then, since the density of every subtask in $TD_A(\tau)$ is twice the utilization of the corresponding task in τ, it follows from similar reasoning provided in previous lemma that, the predicate $sched(\text{FF-3C-}\delta, TD_A(\tau), \Pi(m_1,m_2)\cdot\langle 4,4\rangle)$ holds true as well. Hence the proof. □

Finally, a lemma that will be relied upon for assigning phase-C subtasks:

Lemma 37. *If, for a set $TD_A(\tau)[p]$ of phase-A subtasks,*

$$\delta_{TD_A(\tau)[p]} \stackrel{def}{=} \sum_{\tau_{i(A)}\in TD_A(\tau)[p]} \frac{C^t_{i(A)}}{D^t_{i(A)}} \leq V$$

then $TD_A(\tau)[p] \cup TD_C(\tau)[p]$ (where $TD_C(\tau)[p]$ is the set of the respective phase-C subtasks) is preemptive-EDF schedulable on a type-t (virtual) processor vp_p of speed V.

Proof. That $\delta_{TD_A(\tau)[p]} \leq V$ means that the task set $TD_A(\tau)[p]$ is schedulable under preemptive EDF on processor vp_p. We now show that the *demand-bound function*[3], $\text{dbf}(\tau', t)$, of a task set $\tau' = TD_A(\tau)[p] \cup TD_C(\tau)[p]$ is upper bounded at every instant t by $\delta_{TD_A(\tau)[p]}\cdot t$ and hence is *also* schedulable on processor vp_p under preemptive EDF. Note that, for every phase-A subtask $\tau_{i(A)} \in TD_A(\tau)$ (and respective phase-C subtask $\tau_{i(C)} \in TD_C(\tau)$), it holds $\forall t \in 1,2$ that:

$$\text{dbf}(\{\tau_{i(A)}, \tau_{i(C)}\}, t) \leq \delta^t_{i(A)}\cdot t = \frac{C^t_{i(A)}\cdot t}{D^t_{i(A)}} \tag{5.5}$$

This is easy to verify because, the maximum "slope" to any point in the graph of Figure 5.5 of $\text{dbf}\left(\left\{\tau_{i(A)}, \tau_{i(C)}\right\}, t\right)$ from the origin is $\delta^t_{i(A)} = \frac{C^t_{i(A)}}{D^t_{i(A)}}$ (which is equal to $2U^t_i$ of task $\tau_i \in \tau$, as per our

[3]The *demand bound function* of a task τ_i, $dbf(\tau_i,t)$, is the maximum possible computation demand by jobs of τ_i, that have both release and deadline within any interval of length t. The demand bound function of a task set τ is defined as: $dbf(\tau,t) = \sum_{\tau_i\in\tau} dbf(\tau_i,t)$ [BMR90].

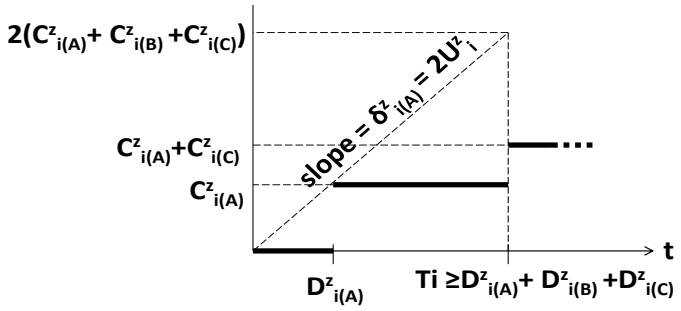


Figure 5.5: Assigning phase-C subtasks to the same virtual processor as the respective phase-A subtasks (earlier assigned using a density-based test) preserves schedulability.

choice of $D^t_{i(A)}$), at abscissa $t = D^t_{i(A)}$. Summation of Equation (5.5) over all $\tau_{i(A)} \in TD_A(\tau)[p]$ (and respective $\tau_{i(C)} \in TD_C(\tau)[p]$) yields:

$$\mathrm{dbf}(TD_A(\tau)[p] \cup TD_C(\tau)[p], t) \leq t \cdot \sum_{\tau_{i(A)} \in TD_A(\tau)[p]} \delta^t_{i(A)} = t \cdot \delta_{TD_A(\tau)[p]}$$

Hence the proof. □

5.5.3 Creating virtual processors

We create $m + 2|R|$ virtual processors from m physical processors of a two-type heterogeneous multiprocessor platform as shown in Figure 5.6. The main idea is as follows. We treat physical processors of each type as an identical multiprocessor platform and create a certain number of virtual processors of the corresponding type from this platform. To be precise, m_1 physical processors of type-1 are treated as an identical multiprocessor platform and $m_1 + |R|$ virtual processors (of type-1) are created from them and ordered as shown in the left half of Figure 5.6 (i.e., left side of the vertical *solid line*). Analogously, m_2 physical processors of type-2 are treated as an identical multiprocessor platform and $m_2 + |R|$ virtual processors (of type-2) are created from them and ordered as shown in the right half of Figure 5.6 (i.e., right side of the vertical *solid line*). Now, if we look at each row in Figure 5.6 (separated by *horizontal lines*), it represents a two-type heterogeneous multiprocessor platform (for example, the second row represents a two-type heterogeneous multiprocessor platform with m_1 virtual processors of type-1 and m_2 virtual processors of type-2). Thus, $m + 2|R|$ virtual processors are formed from m physical processors on a two-type heterogeneous platform. Precisely, we create the virtual processors with the following specifications:

- m **virtual processors (denoted as** $\mathrm{VP_{AC}}$**):** m_1 virtual processors of type-1 each of speed $\frac{2}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}$ times the speed of a physical processor of type-1 and m_2 virtual processors of type-2 each of speed $\frac{2}{2+3\left\lceil \frac{|R|}{m_2} \right\rceil}$ times the speed of a physical processor of type-2. They are used to schedule phase-A and phase-C of a task execution and are referred to as '*virtual processors in* $\mathrm{VP_{AC}}$'.

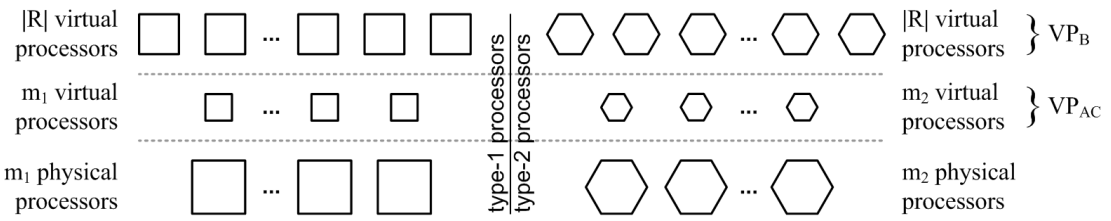


Figure 5.6: $m+2|R|$ virtual processors created from m physical processors on a two-type heterogeneous multiprocessor platform ($m = m_1 + m_2$).

- $2|R|$ **virtual processors (denoted as** VP_B**):** $|R|$ virtual processors of type-1 each of speed $\frac{3}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}$ times the speed of a physical processor of type-1 and $|R|$ virtual processors of type-2 each of speed $\frac{3}{2+3\left\lceil \frac{|R|}{m_2} \right\rceil}$ times the speed of a physical processor of type-2. They are used to schedule phase-B of task execution and are referred to as '*virtual processors in* VP_B'.

We ensure that no virtual processor is created using two or more physical processors, i.e., the capacity of a virtual processor comes from a single physical processor alone. The process of creating virtual processors along with the pseudo-code is discussed in the next subsection.

5.5.4 Algorithm for creating virtual processors

In our notation, *PP* denotes the set of physical processors, pp_i denotes the i^{th} physical processor and vp_i denotes the i^{th} virtual processor. The VP_Create pseudo-code for creating the previously specified virtual processors is listed in Algorithm 12. It, in turn, uses the subroutine VP_ABC_Create, which in turn is listed in Algorithm 13.

The VP_Create algorithm on line 2 calls the subroutine, VP_ABC_Create, to create $m_1 + |R|$ virtual processors of type-1 from m_1 physical processors of type-1. The subroutine first creates m_1 virtual processors (see lines 1-5 in Algorithm 13) from m_1 physical processors and then creates $|R|$ virtual processors (see lines 6-20 in Algorithm 13) from the remaining capacity of type-1

Algorithm 12: VP_*Create*(PP, $|R|$): for creating virtual processors from a two-type heterogeneous computing platform

Input : PP, $|R|$
Output: VP_AC, VP_B
// *PP* denotes the set of physical processors
// $|R|$ denotes the number of shared resources
1 $\text{VP}_\text{AC}[1,\cdots,m] := \{0,\cdots,0\}$ $\text{VP}_\text{B}[1,\cdots,2|R|] := \{0,\cdots,0\}$
2 VP_ABC_*Create*$(PP, \text{VP}_\text{AC}, \text{VP}_\text{B}, 0, 0, 1)$
3 VP_ABC_*Create*$(PP, \text{VP}_\text{AC}, \text{VP}_\text{B}, m_1, |R|, 2)$
4 **return**$\text{VP}_\text{AC}, \text{VP}_\text{B}$

processors. Observe that no virtual processor is created using two physical processors, i.e., the capacity of a virtual processor comes from a single physical processor alone. Similarly, VP_Create() on line 3 creates $m_2 + |R|$ virtual processors of type-2 from m_2 physical processors of type-2.

Algorithm 13: $\text{VP}_{\text{ABC}}_Create(\text{PP}, \text{VP}_{\text{AC}}, \text{VP}_{\text{B}}, lb, si, t)$: for creating phase-AC and phase-B virtual processors

Input : $\text{PP}, \text{VP}_{\text{AC}}, \text{VP}_{\text{B}}, lb, si, t$
Output: VP_{AC}, VP_{B}
// lb denotes the starting index for array VP_{AC}
// si denotes the starting index for array VP_B
// t denotes the processor type
1 $\text{VP}_{\text{AC}}[lb+1, \cdots, lb+m_t] := \{0, \cdots, 0\}$ // initialize the relevant elements in VP_{AC} to zero
2 **for** $i = 1$ **to** m_t **do**
3 Create a virtual processor, vp_i^{ACt}, from pp_i of speed $\frac{2}{2+3\left\lceil \frac{|R|}{m_t} \right\rceil}$ times the speed of pp_i
4 $\text{VP}_{\text{AC}}[lb+i] := vp_i^{ACt}$
5 **end**
6 $cnt := 1, flag := 0$
7 **for** $i = 1$ **to** m_t **do**
8 **for** $j = 1$ *to* $\left\lceil \frac{|R|}{m_t} \right\rceil$ **do**
9 Create a virtual processor, vp_{cnt}^{Bt}, from pp_i of speed $\frac{3}{2+3\left\lceil \frac{|R|}{m_t} \right\rceil}$ times the speed of pp_i
10 $\text{VP}_{\text{B}}[si+cnt] := vp_{cnt}^{Bt}$
11 **if** $(cnt = |R|)$ **then**
12 $flag := 1$
13 break
14 **end**
15 $cnt := cnt + 1$
16 **end**
17 **if** $(flag = 1)$ **then**
18 break
19 **end**
20 **end**

Since VP_{ABC}_Create creates a virtual processor out of the processing capacity of a single respective physical processor, within each of its phases, every job executes on only one physical processor (i.e., a job does not migrate between different physical processors). However, a job can migrate to a different physical processor at the boundaries separating (i) its phase-A and phase-B and (ii) its phase-B and phase-C. FF-3C-vpr adheres to the "restricted migration" model by assigning phase-A and phase-C of a task to the same physical processor.

The following observations can be made regarding our specification and creation of virtual processors. After creating one VP_{AC} virtual processor (for phase-A and phase-C) from every physical processor (lines 1-5 in the subroutine shown in Algorithm 13), let us see (i) how much capacity remains in each of the physical processors and (ii) how many phase-B virtual processors

(i.e. virtual processors in $\mathrm{VP_B}$) can be created from that capacity. For ease of explanation, consider the case of type-1 processors. After creating one $\mathrm{VP_{AC}}$ virtual processor (for phase-A and phase-C) of speed $\frac{2}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}$ (times the speed of a physical processor of type-1) from each physical processor, every physical processor is left with a capacity: $1 - \frac{2}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil} = \frac{3\left\lceil \frac{|R|}{m_1} \right\rceil}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}$. As per our specification (in Section 5.5.2), the phase-B virtual processor must have $\frac{3}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}$ times the speed of a physical processor of type-1. Hence, it is possible to create:

$$\left\lfloor \frac{\frac{3\left\lceil \frac{|R|}{m_1} \right\rceil}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}}{\frac{3}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}} \right\rfloor = \left\lfloor \left\lceil \frac{|R|}{m_1} \right\rceil \right\rfloor = \left\lceil \frac{|R|}{m_1} \right\rceil \geq 1$$

phase-B virtual processors from the remaining capacity of every physical processor of type-1. This allows us to successfully create $|R|$ phase-B virtual processors from the remaining capacity of m_1 processors of type-1. Analogous reasoning holds for type-2 processors as well.

5.6 FF-3C-vpr algorithm and its speed competitive ratio

5.6.1 The FF-3C-vpr algorithm

The pseudo-code of our new algorithm, FF-3C-vpr, is listed in Algorithm 14 and it works as follows. On line 1, it creates $TD_A(\tau)$, $TD_{B,R^k}(\tau)$ and $TD_C(\tau)$ subsets of tasks from the given task set τ. On line 2, it creates $m + 2|R|$ virtual processors specified in Section 5.5.2 from the given m physical processors. On lines 3-5, it groups $2|R|$ phase-B virtual processors into $|R|$ *pairs of virtual processors*, each pair containing one virtual processor of each type, i.e., one virtual processor of type-1 and one virtual processor of type-2. Each pair $Pair_B[k]$ of virtual processors, where $k = \{1, \cdots, |R|\}$, is used for scheduling phase-B of tasks that access the resource, R^k. At any time instant, only one virtual processor from each pair is used for executing the tasks: this is, in each case, the virtual processor of the type on which the given task executes fastest (termed the *favorite* processor type for that task); the other virtual processor is kept idle during the execution of the task. This technique ensures *mutual exclusion* for accessing each resource. Moreover, it effectively creates, out of each pair, the equivalent of a hypothetical single virtual processor whereupon every task would execute as fast as on its (respective) favorite processor type. This design choice aims at minimizing *blocking times*[4] related to resource sharing. On line 6, the algorithm assigns phase-A of the tasks (i.e., subtasks in $TD_A(\tau)$) to virtual processors in $\mathrm{VP_{AC}}$ using FF-3C [ARB10]. On lines 7-16, it assigns phase-B of the tasks (i.e., subtasks in $TD_{B,R^k}(\tau)$) accessing resource R^k to that virtual processor in $Pair_B[k]$ which is of its *favorite* processor type in phase-B. On line 17, it assigns phase-C of the tasks (i.e., subtasks in $TD_C(\tau)$) to a virtual processor

[4]The *blocking time* of a task that requests to access a resource is defined as the time duration during which it is blocked by a lower priority task holding that resource.

Algorithm 14: FF-3C-vpr($\tau, \Pi^2(m_1,m_2), R$): for scheduling tasks that share resources on a two-type heterogeneous multiprocessor platform

```
   // Lines 1-17 execute offline; line 18 executes at run-time.
 1 Create TD_A(τ), TD_{B,R^k}(τ) and TD_C(τ) from τ as described in Section 5.5.1
 2 {VP_AC, VP_B} := VP_Create(Π^2(m_1,m_2), R) // Create VP_AC and VP_B virtual
       processors and store them in arrays of structures
 3 for i = 1 to |R| do //Form |R| pairs from 2|R| virtual processors in VP_B
 4 |   Pair_B[i] := ⟨VP_B[i], VP_B[|R| + i]⟩
 5 end
 6 Assign TD_A(τ) to virtual processors in VP_AC using FF-3C
 7 for i = 1 to n do
 8 |   if τ_i requests a resource then
 9 |   |   let k denote the resource that task τ_i requests
10 |   |   if (C^1_{i(B)} ≤ C^2_{i(B)}) then
11 |   |   |   assign τ_{i(B)} to VP_B[k]
12 |   |   else
13 |   |   |   assign τ_{i(B)} to VP_B[|R| + k]
14 |   |   end
15 |   end
16 end
17 Assign TD_C(τ) to virtual processors in VP_AC using the assignment made by FF-3C for
       phase-A of tasks on line 6, i.e. if τ_{i(A)} of TD_A(τ) was assigned to VP_AC[j] processor then
       assign τ_{i(C)} of TD_C(τ) to VP_AC[j] processor
18 Dispatch tasks in (i) TD_A(τ) with preemptive EDF on VP_AC, (ii) TD_B(τ) with
       non-preemptive EDF on VP_B and (iii) TD_C(τ) with preemptive EDF on VP_AC
```

in VP_{AC} in the same manner as that of assignment of a task in $TD_A(\tau)$ to a virtual processor in VP_{AC} by FF-3C (on line 6). Instead of running FF-3C again on $TD_C(\tau)$ task set, the algorithm makes use of the output of FF-3C (that was run on line 6 to assign tasks in $TD_A(\tau)$ on VP_{AC}) to assign tasks in $TD_C(\tau)$. Thus, line 17 ensures that phase-C of a task is assigned to that virtual processor in VP_{AC} to which phase-A of the same task has been assigned. Assigning phase-C subtasks on the same virtual processor as its corresponding phase-A subtask (i) does not endanger the schedulability of a previously schedulable virtual processor; intuitively, this is because these two subtasks have precedence constraints – Lemma 37 provides formal proof and (ii) ensures that the "restricted migration" assumption is not violated. On line 18, FF-3C-vpr schedules the tasks executing in their phase-A onto VP_{AC} virtual processors using preemptive EDF, the tasks in their phase-B onto VP_{B} virtual processors using non-preemptive EDF and the tasks in their phase-C onto VP_{AC} virtual processors using preemptive EDF. Lines 1-17 can be performed before run-time and only line 18 has to be performed at run-time.

5.6.2 Time complexity of FF-3C-vpr algorithm

We now show that the time-complexity of FF-3C-vpr is a polynomial function of the number of tasks (n) and the number of processors (m). From the pseudo-code of FF-3C-vpr, listed in Algorithm 14, we can observe that the time-complexity for:

- creating the subsets of tasks $TD_A(\tau)$, $TD_{B,R^k}(\tau)$ and $TD_C(\tau)$, on line 1 is: $O(n)$.
- creating the virtual processor subsets, VP_{AC} and VP_{B}, on line 2 is: $O(m)$.
- forming the virtual processor pairs, on lines 3-5 is: $O(|R|)$.
- assigning the tasks in $TD_A(\tau)$ to VP_{AC} virtual processors using FF-3C (on line 6) is: $O(n \cdot \max(m, \log n))$ — see Section 4.3.7 on page 100 in Chapter 4.
- assigning the tasks in $TD_{B,R^k}(\tau)$ to VP_{B} virtual processors, on lines 7-16 is: $O(n)$.
- assigning the tasks in $TD_C(\tau)$ to VP_{AC} virtual processors, on line 17 is: $O(n)$.

Thus, the time-complexity of FF-3C-vpr algorithm is at most:

$$\Big(\underbrace{O(n)}_{\substack{\text{create}\\ \text{subtasks}}} + \underbrace{O(m)}_{\substack{\text{create virtual}\\ \text{processors}}} + \underbrace{O(|R|)}_{\substack{\text{form virtual}\\ \text{processor pairs}}} + \underbrace{O(n \cdot \max(m, \log n)}_{\substack{\text{assign tasks}\\ \text{in } TD_A(\tau)}} + \underbrace{O(n)}_{\substack{\text{assign tasks}\\ \text{in } TD_{B,R^k}(\tau)}} + \underbrace{O(n)}_{\substack{\text{assign tasks}\\ \text{in } TD_C(\tau)}} \Big)$$
$$= O(\max(n \cdot \max(m, \log n)), |R|)$$
$$= O(n \cdot \max(m, \log n))$$

5.6.3 Speed competitive ratio of FF-3C-vpr algorithm

We now prove the speed competitive ratio of FF-3C-vpr algorithm.

Theorem 20. *The speed competitive ratio of FF-3C-vpr is* $4 + 6 \cdot \left\lceil \frac{|R|}{\min(m_1, m_2)} \right\rceil$.

Proof. The proof considers separately the scheduling of each of the three phases and then combines the results. Let us look at phase-A first. Combining Lemma 35 and Lemma 36 and applying the result to virtual processors in VP_{AC} yields:

$$sched(\text{nmo}, \tau, \Pi(m_1, m_2)) \Rightarrow sched(\text{FF-3C-}\delta, TD_A(\tau), \Pi(m_1, m_2) \cdot \langle 4, 4 \rangle) \tag{5.6}$$

Now consider phase-C. Since a task in its phase-A cannot be in its phase-C simultaneously (and vice versa), the respective subtasks are not independent. Treating them as such would be potentially pessimistic; conversely, accounting for these precedence constraints during subtask assignment could improve performance. Indeed, our algorithm assigns each phase-C subtask to the same virtual processor as its respective phase-A subtask (see line 17 in Algorithm 14).

For convenience, let us introduce a notation, FF-3C-δ+cp, for this subtask assignment strategy (using FF-3C-δ to assign phase-A subtasks and "copying" the assignment for respective phase-C

subtasks, as done by FF-3C-vpr algorithm on line 6). Then, applying Lemma 37 to Equation (5.6) yields:

$$\begin{aligned} &sched(\text{nmo}, \tau, \Pi(m_1, m_2)) \Rightarrow \\ &sched(\text{FF-3C-}\delta\text{+cp}, TD_A(\tau) \cup TD_C(\tau), \Pi(m_1, m_2) \cdot \langle 4, 4 \rangle) \end{aligned} \tag{5.7}$$

Now, let us consider phase-B. If tasks in τ that share a resource, R^k, are non-migrative-offline, non-preemptive schedulable on platform $\Pi(m_1, m_2)$ then the task set $TD_{B,R^k}(\tau)$ is also non-migrative-offline, non-preemptive schedulable on platform $\Pi(m_1, m_2) \cdot \langle 2, 2 \rangle$. This speedup factor of 2 comes from the fact that we have halved the deadlines of tasks in $TD_{B,R^k}(\tau)$ compared to the deadlines of corresponding tasks in τ. Hence, we can write: $\forall R^k \in R$:

$$sched(\text{nmo-np}, \tau, \Pi(m_1, m_2)) \Rightarrow sched(\text{nmo-np}, TD_{B,R^k}(\tau), \Pi(m_1, m_2) \cdot \langle 2, 2 \rangle) \tag{5.8}$$

For each resource R^k, since R^k is accessed in a mutually exclusive way, all the tasks that access R^k must execute sequentially. So, if the task set $TD_{B,R^k}(\tau)$ in which tasks share a single resource, R^k, is non-migrative-offline non-preemptive schedulable on platform $\Pi(m_1, m_2) \cdot \langle 2, 2 \rangle$ then the same task set is also non-migrative-offline non-preemptive schedulable on platform $\Pi(1, 1) \cdot \langle 2, 2 \rangle$. The intuition is that the tasks are always executed on their 'favorite' processor type and in a sequential manner as they are accessing a mutually exclusive resource. $\forall R^k \in R$:

$$\begin{aligned} &sched(\text{nmo-np}, TD_{B,R^k}(\tau), \Pi(m_1, m_2) \cdot \langle 2, 2 \rangle) \Rightarrow \\ &\qquad sched(\text{nmo-np}, TD_{B,R^k}(\tau), \Pi(1, 1) \cdot \langle 2, 2 \rangle) \end{aligned} \tag{5.9}$$

Hence, combining Equations (5.8) and (5.9) gives: $\forall R^k \in R$:

$$sched(\text{nmo-np}, \tau, \Pi(m_1, m_2)) \Rightarrow sched(\text{nmo-np}, TD_{B,R^k}(\tau), \Pi(1, 1) \cdot \langle 2, 2 \rangle) \tag{5.10}$$

Without loss of generality, Lemma 34 can be rewritten as:

$$sched(\text{nmo-np}, \tau, \Pi(1, 1)) \Rightarrow sched(\text{nm-np-EDF}, \tau, \Pi(1, 1) \cdot \langle 3, 3 \rangle) \tag{5.11}$$

The intuition behind this generalization of Lemma 34 to Expression (5.11) is that the extra processor added to the left-hand side predicate (of Lemma 34 to obtain the Expression (5.11)) is ignored while scheduling.

Applying Equation (5.11) to a task set $TD_{B,R^k}(\tau)$ and multiplying the processor speeds by 2 on both left-hand and right-hand side platforms gives: $\forall R^k \in R$:

$$\begin{aligned} &sched(\text{nmo-np}, TD_{B,R^k}(\tau), \Pi(1, 1) \cdot \langle 2, 2 \rangle) \Rightarrow \\ &\qquad sched(\text{nm-np-EDF}, TD_{B,R^k}(\tau), \Pi(1, 1) \cdot \langle 6, 6 \rangle) \end{aligned} \tag{5.12}$$

Combining Equation (5.10) and (5.12) and applying the result to $\mathrm{VP_B}$ virtual processors, we obtain: $\forall R^k \in R$,

$$sched(\text{nmo-np}, \tau, \Pi(m_1, m_2)) \Rightarrow sched(\text{nm-np-EDF}, TD_{B,R^k}(\tau), \Pi(1,1) \cdot \langle 6, 6 \rangle) \tag{5.13}$$

Combining the above intermediate results: dividing type-1 and type-2 processor speeds by, respectively, $4+6\left\lceil \frac{|R|}{m_1} \right\rceil$ and $4+6\left\lceil \frac{|R|}{m_2} \right\rceil$ in Equations (5.7) and (5.13) gives us:

$$\begin{aligned} &sched\left(\text{nmo}, \tau, \Pi(m_1, m_2) \cdot \left\langle \frac{1}{4+6\left\lceil \frac{|R|}{m_1} \right\rceil}, \frac{1}{4+6\left\lceil \frac{|R|}{m_2} \right\rceil} \right\rangle\right) \Rightarrow \\ &sched\left(\text{FF-3C-}\delta\text{+cp}, TD_A(\tau) \cup TD_C(\tau), \Pi(m_1, m_2) \cdot \left\langle \frac{2}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}, \frac{2}{2+3\left\lceil \frac{|R|}{m_2} \right\rceil} \right\rangle\right) \end{aligned} \tag{5.14}$$

and

$$\begin{aligned} \forall R^k \in R : &sched\left(\text{nmo-np}, \tau, \Pi(m_1, m_2) \cdot \left\langle \frac{1}{4+6\left\lceil \frac{|R|}{m_1} \right\rceil}, \frac{1}{4+6\left\lceil \frac{|R|}{m_2} \right\rceil} \right\rangle\right) \Rightarrow \\ &sched\left(\text{nm-np-EDF}, TD_{B,R^k}(\tau), \Pi(1,1) \cdot \left\langle \frac{3}{2+3\left\lceil \frac{|R|}{m_1} \right\rceil}, \frac{3}{2+3\left\lceil \frac{|R|}{m_2} \right\rceil} \right\rangle\right) \end{aligned} \tag{5.15}$$

In the right-hand sides of Equation (5.14) and Equation (5.15), the processor specifications match those created by FF-3C-vpr algorithm. Note also that, under FF-3C-vpr algorithm (which only allows "restricted migration"), phase-A and phase-C subtasks are assigned to virtual processors in VP_{AC} and phase-B subtasks are assigned to virtual processors in VP_B (and $VP_{AC} \cap VP_B = \emptyset$). Hence by combining Equation (5.14) and Equation (5.15), we get:

$$\begin{aligned} &sched\left(\text{rmo}, \tau, R, \Pi(m_1, m_2) \cdot \left\langle \frac{1}{4+6\left\lceil \frac{|R|}{m_1} \right\rceil}, \frac{1}{4+6\left\lceil \frac{|R|}{m_2} \right\rceil} \right\rangle\right) \Rightarrow \\ &\qquad sched\Big(\text{FF-3C-vpr}, \tau, R, \Pi(m_1, m_2)\Big) \end{aligned} \tag{5.16}$$

We know that higher speed processors do not jeopardize the schedulability of a task set. Hence, we can write:

$$sched(\text{rmo}, \tau, R, \Pi(m_1, m_2) \cdot \langle \min(s_1, s_2), \min(s_1, s_2) \rangle) \;\Rightarrow\; sched(\text{rmo}, \tau, R, \Pi(m_1, m_2) \cdot \langle s_1, s_2 \rangle)$$

Substituting $s_1 = \frac{1}{4+6\left\lceil \frac{|R|}{m_1} \right\rceil}$ and $s_2 = \frac{1}{4+6\left\lceil \frac{|R|}{m_2} \right\rceil}$ in the previous equation, combining it with Equation (5.16) and rewriting gives:

$$sched\left(\text{rmo}, \tau, R, \Pi(m_1, m_2) \cdot \left\langle \frac{1}{4+6\cdot\max\left(\left\lceil \frac{|R|}{m_1} \right\rceil, \left\lceil \frac{|R|}{m_2} \right\rceil\right)}, \frac{1}{4+6\cdot\max\left(\left\lceil \frac{|R|}{m_1} \right\rceil, \left\lceil \frac{|R|}{m_2} \right\rceil\right)} \right\rangle\right) \Rightarrow sched\Big(\text{FF-3C-vpr}, \tau, R, \Pi(m_1, m_2)\Big) \tag{5.17}$$

Multiplying processor speeds in Equation (5.17) by $4+6\cdot\max\left(\left\lceil \frac{|R|}{m_1} \right\rceil, \left\lceil \frac{|R|}{m_2} \right\rceil\right)$, yields:

$$\begin{aligned} &sched\Big(\text{rmo}, \tau, R, \Pi(m_1, m_2)\Big) \Rightarrow \\ &sched\left(\text{FF-3C-vpr}, \tau, R, \Pi(m_1, m_2) \cdot \left\langle 4+6\cdot\max\left(\left\lceil \frac{|R|}{m_1} \right\rceil, \left\lceil \frac{|R|}{m_2} \right\rceil\right), \right.\right. \\ &\qquad \left.\left. 4+6\cdot\max\left(\left\lceil \frac{|R|}{m_1} \right\rceil, \left\lceil \frac{|R|}{m_2} \right\rceil\right) \right\rangle\right) \end{aligned} \tag{5.18}$$

By rewriting the right-hand side of Equation (5.18), we get:

$$\begin{aligned} &sched\Big(\text{rmo}, \tau, R, \Pi(m_1, m_2)\Big) \Rightarrow \\ &sched\left(\text{FF-3C-vpr}, \tau, R, \Pi(m_1, m_2) \cdot \left\langle 4+6\cdot\left\lceil \frac{|R|}{\min(m_1, m_2)} \right\rceil, 4+6\cdot\left\lceil \frac{|R|}{\min(m_1, m_2)} \right\rceil \right\rangle\right) \end{aligned}$$

Hence the proof. □

5.7 Conclusions

In many computer systems, apart from processors, tasks also share resources such as data structures, sensors, etc in a *mutually exclusive* manner. Scheduling such tasks to meet all deadlines on two-type heterogeneous multiprocessors is a complex problem. In this chapter, we took the first step to solve the issue by studying a restricted version of this problem and proposing an algorithm with a finite speed competitive ratio. Specifically, we considered the problem of scheduling a set of implicit-deadline sporadic tasks to meet all deadlines on two-type heterogeneous multiprocessors where tasks may share resources. The tasks must operate on such resources in a mutually exclusive manner while accessing the resources, that is, at all times, when a job of a task holds a resource, no other job of any task can hold that resource. Each task may request at most one resource and each job of this task can request that resource at most once during its execution. A job is allowed to migrate when it requests/releases the resource but a job is not allowed to migrate at other times.

For this problem, we proposed a new algorithm, FF-3C-vpr, with a low-degree polynomial time-complexity. We also showed that FF-3C-vpr has a speed competitive ratio $4+6\cdot\left\lceil\frac{|R|}{\min(m_1,m_2)}\right\rceil$ against equally powerful adversary (which also allows a job to migrate only when it requests or releases a resource). To the best of our knowledge, for the problem of shared resource scheduling on two-type heterogeneous multiprocessors, no previous algorithm is known to exist and hence this is the first result with provably good performance.

Part III

T-type Heterogeneous Multiprocessors

Chapter 6

Intra-migrative Scheduling on T-type Heterogeneous Multiprocessors

6.1 Introduction

In this chapter, we consider the problem of intra-migrative scheduling of tasks on t-type (where $t \geq 2$) heterogeneous multiprocessors. Recall that we discussed the intra-migrative task assignment problem earlier in Chapter 3 but for two-type heterogeneous multiprocessors. Hence, the algorithm presented in that chapter is only applicable to two-type platforms and unfortunately, cannot be generalized to t-type ($t \geq 2$) heterogeneous multiprocessors. Therefore, in this chapter, we aim to design an intra-migrative task assignment algorithm for t-type heterogeneous multiprocessors.

Recall that, in the *intra-migrative* model, every task is statically assigned to a *processor type* before run-time; the jobs of each task can migrate at run-time from one processor to another as long as these processors are of the same type. Once tasks are assigned to processor types, scheduling them to meet all deadlines under the intra-migrative model is well-understood, e.g., one may use an optimal identical multiprocessor scheduling algorithm, such as, ERfair [AS00], DP-Fair [LFS+10] or U-EDF [NBN+12]. So, assuming that an optimal algorithm is used for scheduling tasks on processors of each type, the challenging part is to find a *task-to-processor-type* assignment such that, *there exists* a schedule that meets all deadlines — such an assignment is referred to as a *feasible* assignment hereafter. It can be shown that the problem of intra-migrative task assignment on t-type heterogeneous multiprocessors is NP-Complete in the strong sense (by reducing an instance of the 3-PARTITION problem, which is known to be NP-Complete in the strong sense [Joh73], to an instance of our problem). Therefore, for this problem, we propose a *polynomial* time-complexity algorithm, LPG_{IM}, with a *finite* speed competitive ratio.

Problem Statement. In this chapter, we consider the problem of intra-migrative scheduling of implicit-deadline sporadic tasks on t-type heterogeneous multiprocessors. That is, assuming that an optimal identical multiprocessor scheduling algorithm is used on processors of each type to schedule the tasks, we design an algorithm for determining a feasible task-to-processor-type assignment.

Hardness of the Problem. It is trivial to see that the problem of intra-migrative task assignment on t-type heterogeneous multiprocessors is NP-Complete in the strong sense. This is because, even in the simpler case, in which each processor type has only one processor, the problem of intra-migrative task assignment is NP-Complete in the strong sense (since in this case, the problem is equivalent to finding a non-migrative task assignment on heterogeneous multiprocessors with one processor of each type; this problem is shown to be NP-Complete in the strong sense even in the simpler case of two-type heterogeneous multiprocessors — see Section 4.2 in Chapter 4 on page 76). Hence, this result continues to hold for t-type ($t \geq 2$) heterogeneous multiprocessors as well.

Related Work. The scheduling problem on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, RAB13, RN12b, WBB13, HS76, LST90, JP99, CSV12, Bar04a]. However, all of them consider the problem of non-migrative scheduling (except the work by Baruah [Bar04a], which studied fully-migrative scheduling) and none of them consider the problem of intra-migrative scheduling in which tasks need to be assigned to processor types and not to individual processors. Although some of these non-migrative algorithms can be "adapted" to the intra-migrative model, these "adapted" algorithms will be inefficient either in terms of the speed competitive ratio or in terms of the time-complexity. An intra-migrative algorithm, namely SA, that was presented earlier in Chapter 3 exists but it is only applicable to two-type heterogeneous multiprocessors and unfortunately cannot be extended to t-type ($t \geq 2$) heterogeneous multiprocessors. The state-of-the-art (along with the contributions of this chapter) is summarized in Table 6.1.

Contributions and Significance of the work discussed in this chapter. Consider a t-type platform π and an implicit-deadline sporadic task set τ, in which it holds that: $\forall k \in \{1,2,\ldots,t\}$, for every task in τ, utilization of each task on a type-k processor is either no greater than α or is equal to ∞, where $0 < \alpha \leq 1$. For this setting, we present an intra-migrative task assignment algorithm, called LPG_{IM}, which has a polynomial time-complexity and offers the following guarantee. If there exists a feasible intra-migrative assignment of the task set τ on the t-type platform π then LPG_{IM} succeeds as well in finding such a feasible intra-migrative assignment of τ but on a platform, π', in which *only one processor of each type* is $1 + \alpha \times \frac{t-1}{t}$ times faster than the corresponding processor in π. For defining its speed competitive ratio, we say that, LPG_{IM} needs a platform, $\pi^{(1+\alpha\times\frac{t-1}{t})}$, in which every processor is $1 + \alpha \times \frac{t-1}{t}$ times faster). For the special case in which $t = 2$, i.e., for two-type heterogeneous multiprocessors, the speed competitive ratio of LPG_{IM} becomes $1 + \frac{\alpha}{2} \leq 1.5$. Hence, this result can be seen as a generalization of the result obtained for SA algorithm in Chapter 3; however, LPG_{IM} algorithm itself is not a generalization of SA algorithm as it is designed using entirely different concepts.

We believe that the significance of this work is as follows. For the problem of intra-migrative task assignment on t-type heterogeneous multiprocessors, no previous algorithm exists[1] and hence

[1]Some of the non-migrative algorithms from state-of-the-art (for example, the algorithms presented in [HS76, LST90]) can be "adapted" to intra-migrative scenario, however, these "adapted" algorithms will be inefficient compared to the LPG_{IM} algorithm, either in terms of the speed competitive ratio or in terms of the time-complexity. For example, the adapted version of the algorithm in [LST90] will have inferior speed competitive ratio and the adapted

Computing Platform	Adversary	Task Assignment Algorithms			
	Task migration	Algorithm	Task migration	Speed competitive ratio	Complexity
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	non-migrative	FF-3C (Chap. 4, Sec. 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	intra-migrative	SA-P (Chap. 4, Sec. 4.4)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	non-migrative	LPC (Chap. 4, Sec. 4.5)	non-migrative	1.5 (and 3 extra processors)	$O(P)$
2-type	non-migrative	PTAS_{NF} (Chap. 4, Sec. 4.6)	non-migrative	PTAS	exponential in $1/\varepsilon$
t-type	intra-migrative	LPG_{IM}	intra-migrative	$1+\alpha \times \frac{t-1}{t}$	$O(P)$

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 6.1: Summary of state-of-the-art task assignment algorithms along with the LPG_{IM} algorithm proposed in this chapter.

our algorithm, LPG_{IM}, is the first for this problem.

A global view. The context of the new algorithm LPG_{IM} can be visualized as shown in Figure 6.1.

Organization of the chapter. The rest of the chapter is organized as follows. Section 6.2 briefs the system model. Section 6.3 presents an optimal *intra-migrative* task assignment algorithm, MILP-Algo, that uses Mixed Integer Linear Programming (MILP) formulation. Since solving the MILP formulation for this problem is NP-Complete in the strong sense, a polynomial time-complexity algorithm, LPG_{IM}, is presented by relaxing the MILP formulation to Linear Programming (LP) formulation and using graph theory techniques. Section 6.4 gives a four step overview of LPG_{IM}. Section 6.5–Section 6.8 discuss each of the four steps of LPG_{IM} algorithm in detail and also prove its speed competitive ratio. Finally, Section 6.9 concludes.

version of the algorithm in [HS76] will continue to have a significantly higher time-complexity (which will severely limit the practicality of this algorithm).

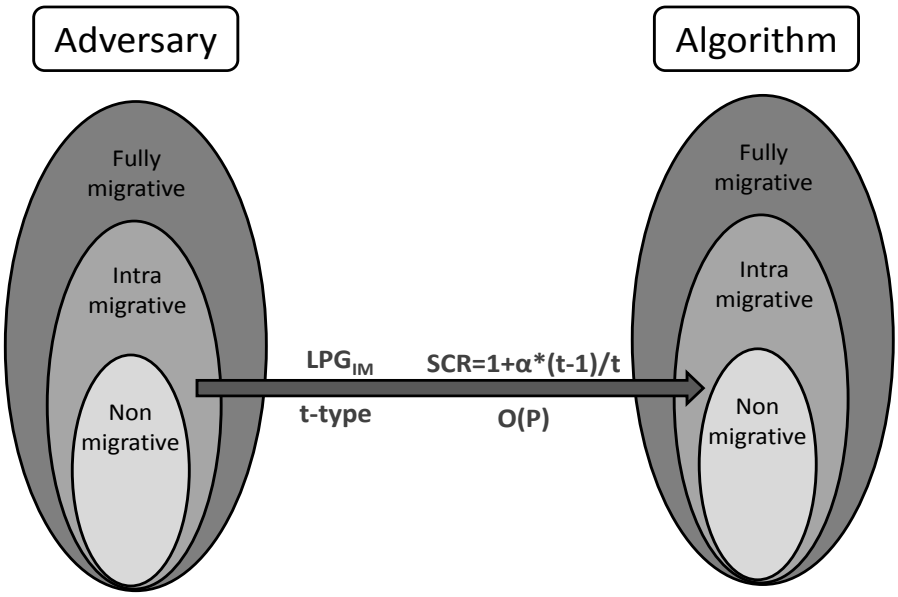


Figure 6.1: A global view of the new algorithm, LPG_{IM}, proposed in this chapter. Here, SCR denotes the "speed competitive ratio", α is a property of the task set — it is the maximum of all the task utilizations that are no greater than one (and hence can take a value in the range $(0,1]$), t denotes the number of processor types and $O(P)$ indicates that the algorithm relies on solving a Linear Program formulation.

6.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks on a t-type heterogeneous multiprocessor platform π comprising m processors of which m_k processors are of type-k, where $k \in \{1,2,\ldots,t\}$. In platform π, the set of m_k processors of type-k is denoted by $\pi^k = \{p_1, p_2 \ldots, p_{m_k}\}$, where p_j denotes a processor of type-k, where $1 \leq j \leq m_k$. It then holds that: $\bigcup_{k=1}^{t} \pi^k = \pi$ and $\bigcap_{k=1}^{t} \pi^k = \emptyset$ and finally $\sum_{k=1}^{t} m_k = m$.

The minimum inter-arrival time of task a τ_i is denoted by T_i. On a t-type platform, the WCET of every task depends on the type of the processor on which the task executes. We denote by C_i^k the WCET of task τ_i when executed on a type-k processor, where $k \in \{1,2,\ldots,t\}$. We denote by $u_i^k \stackrel{\text{def}}{=} C_i^k/T_i$ the utilization of task τ_i on a type-k processor and u_i^k is a real number in $[0,1] \cup \{\infty\}$ — if τ_i cannot be executed on a type-k processor then u_i^k is set to ∞. Let α be a real number defined as follows:

$$\alpha \stackrel{\text{def}}{=} \max_{\tau_i \in \tau, k \in \{1,2,\ldots,t\}} \left\{ u_i^k : u_i^k \leq 1 \right\} \tag{6.1}$$

Then it holds that, the utilization of any task on any processor type is either no greater than α or is equal to ∞, formally,

$$\forall k \in \{1,2,\ldots,t\}, \forall \tau_i \in \tau: \ \ (u_i^k \leq \alpha) \ \vee \ (u_i^k = \infty) \tag{6.2}$$

Minimize Z subject to the following constraints:

I1. $\forall \tau_i \in \tau : \sum_{k \in \{1,2,\ldots,t\}} x_i^k = 1$
I2. $\forall k \in \{1,2,\ldots,t\} : \sum_{\tau_i \in \tau} x_i^k \times u_i^k \leq Z \times m_k$
I3. $\forall \tau_i \in \tau, \forall k \in \{1,2,\ldots,t\} : x_i^k \in \{0,1\}$ are integers

Figure 6.2: MILP-Feas(τ, π) — MILP formulation for assigning tasks in implicit-deadline sporadic task set τ to processor types in t-type heterogeneous platform π.

6.3 MILP-Algo: An optimal intra-migrative algorithm

In this section, an *optimal intra-migrative algorithm* is presented for assigning tasks in a task set τ to processor types in a t-type platform π. The algorithm is optimal in the sense that, if there exists a feasible intra-migrative assignment of τ on π then this algorithm succeeds as well in finding such a feasible intra-migrative assignment. The proposed algorithm is based on solving a Mixed Integer Linear Programming (MILP) formulation. As described in Section 6.1, once the tasks are assigned to processor types, we assume that, an optimal identical multiprocessor scheduling policy (such as, ERfair [AS00], DP-Fair [LFS+10], U-EDF [NBN+12]) is used to schedule the tasks on processors of each type. From the feasibility tests of identical multiprocessor scheduling, the following necessary and sufficient set of conditions must hold for intra-migrative assignment to be feasible:

$$\forall k \in \{1,2,\ldots,t\} : \quad \forall \tau_i \in \tau^k : u_i^k \leq 1 \tag{6.3}$$

$$\forall k \in \{1,2,\ldots,t\} : \quad \sum_{\tau_i \in \tau^k} u_i^k \leq m_k \tag{6.4}$$

where τ^k denotes the set of tasks that are assigned to processors of type-k. The first condition is essential since the system model does not allow a job to execute simultaneously on more than one processor. The second condition is essential as it ensures that the computing *workload* does not exceed the processing *capacity* [Hor74].

Given these necessary and sufficient feasibility conditions, we now propose an optimal intra-migrative task assignment algorithm, MILP-Algo, which works as follows.

First, solve the MILP formulation, MILP-Feas(τ, π), shown in Figure 6.2. In this formulation, variable Z is the objective function to be minimized and it denotes the maximum capacity that is used on any processor type. Each variable x_i^k indicates whether a task τ_i is assigned to processors of type-k or not (i.e., to a processor type and not to an individual processor). The first set of constraints specifies that every task must be assigned. The second set of constraints asserts that at most $Z \times m_k$ capacity of type-k processors can be used. The third set of constraints asserts that each task must be *integrally* assigned to one of the t processor types.

Second, using the solution provided by the MILP solver to our MILP formulation, assign the tasks to processor types as follows. If $Z > 1$ then declare failure as this indicates that the feasibility condition shown in Equation (6.4) is violated (implying that the task set is not intra-migrative feasible). Otherwise, for each task $\tau_i \in \tau$, assign τ_i to type-k processors only if $x_i^k = 1$.

We now show that the MILP-Algo is an optimal intra-migrative task assignment algorithm.

Lemma 38 (**MILP-Algo is optimal**). *If there exists a feasible intra-migrative assignment of implicit-deadline sporadic task τ on t-type heterogeneous multiprocessor platform π then MILP-Algo succeeds as well in finding such a feasible intra-migrative assignment of τ on π.*

Proof. Suppose that, the task set τ is intra-migrative feasible on platform π and let $\mathscr{X}$ denote a feasible assignment. It can be seen that, $\forall \tau_i \in \tau$, by assigning values to x_i^k variables of MILP formulation, MILP-Feas(τ, π), of Figure 6.2 as:

$$\text{if } \mathscr{X}(i) = k \text{ then} \qquad x_i^k \leftarrow 1 \text{ and}$$
$$x_i^j \leftarrow 0, \forall j \in \{1,2,\ldots,t\} \wedge j \neq k$$

gives a (feasible) solution to the MILP formulation in which $Z \leq 1$.

Now, suppose that, there is a (feasible) solution with $Z \leq 1$ to the MILP formulation, MILP-Feas(τ, π), of Figure 6.2. Using this solution, define the assignment of tasks to processor types as follows:

$$\forall \tau_i \in \tau: \text{ if } x_i^k = 1 \text{ then set } \mathscr{X}(i) \leftarrow k$$

By constraint I1 of the MILP formulation, each task is entirely assigned in the assignment $\mathscr{X}$ obtained as shown above. By constraint I2 of the MILP formulation, the capacity of type-k processors is not exceeded in the assignment $\mathscr{X}$ (since $Z \leq 1$ in the feasible solution to MILP formulation). By constraint I3, each task is integrally assigned to one of the processor types. Thus, $\mathscr{X}$ is a feasible intra-migrative assignment. Hence the proof. □

In general, solving an MILP formulation has high computational complexity. In particular, the decision problem MILP is NP-complete and even with knowledge of the structure of the constraints in the modeling of heterogeneous multiprocessor scheduling, no polynomial-time algorithm is known (p. 245 in [GJ79]). Hence, we now propose a polynomial time-complexity (but non-optimal) intra-migrative algorithm, LPG_{IM}, by relaxing the MILP formulation to LP (which can be solved in polynomial time [Kar84]) and using graph theory techniques.

6.4 An overview of our intra-migrative task assignment algorithm, LPG_{IM}

We now give an overview of our new intra-migrative task assignment algorithm, LPG_{IM}. It has the following four steps:

Step 1. We first relax the MILP formulation of Figure 6.2 to an LP formulation by allowing all the x_i^k variables to take *real* values in the range $[0,1]$ instead of binary values $\{0,1\}$ and then solve this relaxed LP formulation. In the solution returned by the LP solver, some tasks will be *integrally* assigned to a processor type and the rest will be *fractionally* assigned to more than one

Minimize Z subject to the following constraints:

R1. $\forall \tau_i \in \tau : \sum_{k \in \{1,2,\ldots,t\}} x_i^k = 1$
R2. $\forall k \in \{1,2,\ldots,t\} : \sum_{\tau_i \in \tau} x_i^k \times u_i^k \leq Z \times m_k$
R3. $\forall \tau_i \in \tau, \forall k \in \{1,2,\ldots,t\} : x_i^k \geq 0$ are real numbers

Figure 6.3: LP-Feas(τ, π) — Relaxed LP formulation for assigning tasks in an implicit-deadline sporadic task set τ to processor types in a t-type heterogeneous platform π.

processor type. We show that, for this LP formulation, there exists a *(vertex) solution* in which *at most* $t-1$ tasks are fractionally assigned and such a solution can be obtained and is of interest to us. This step is discussed in Section 6.5.

Step 2. From such a solution, we construct a bi-partite graph with (i) a set of nodes corresponding to fractional tasks, (ii) another set of nodes corresponding to only those processor types to which these fractional tasks are assigned (note that, there is no processor type node for a processor type to which no fractional task is assigned) and (iii) a set of edges which connect these task nodes and processor type nodes depending on the values of the x_i^k variables (which also represent the *weights* of these edges). The solution (returned by the LP solver) might be such that, upon representing it as a bi-partite graph, the graph may contain a few *circuits*. This step is discussed in detail in Section 6.6 along with the relevant graph theory terminology.

Step 3. The circuits in the graph, if any, are detected and broken, one by one. A circuit is broken by re-adjusting the weights of the edges such that the weight of at least one edge in the circuit becomes zero which is then deleted. While re-adjusting the weights, it is ensured that, for each processor type, its used capacity after re-adjusting the weights does not exceed its used capacity before re-adjusting. This step (discussed in Section 6.7) reduces the complexity of the problem when assigning the at most $t-1$ fractional tasks integrally to processor types, in the final step.

Step 4. The at most $t-1$ fractional tasks are assigned integrally to processor types. We show that, in order to do this, the algorithm needs a platform in which *only one processor of each type* is $1 + \alpha \times \frac{t-1}{t}$ times faster. Thus, we conclude that the speed competitive ratio of LPG_{IM} algorithm is $1 + \alpha \times \frac{t-1}{t}$. This step is discussed in Section 6.8 along with the proof of speed competitive ratio of this four step intra-migrative algorithm, LPG_{IM}.

6.5 Step 1 of LPG_{IM}: Solving the LP formulation

First, we relax the MILP formulation, MILP-Feas(τ, π), to an LP formulation, LP-Feas(τ, π), as shown in Figure 6.3. In this LP formulation, all the variables have the same meaning as in the MILP formulation and the first two sets of constraints are the same as well. Only the third set of constraints is different (i.e., *relaxed*) and it now asserts that a task can either be *integrally* assigned or *fractionally* assigned to processor types. We then solve the LP formulation using standard

LP solvers (such as, IBM ILOG CPLEX [IBM12], Gurobi optimizer [Gur13]). Since the LP formulation is less constrained than the MILP, the following lemma trivially holds.

Lemma 39. *Let Z_{MILP} and Z_{LP} be the values of the objective functions that any MILP solver and LP solver would return by solving MILP-Feas(τ,π) and LP-Feas(τ,π), respectively. It then holds that, $Z_{\text{LP}} \leq Z_{\text{MILP}}$.*

Among all the optimal solutions to an LP formulation, at least one solution lies at a *vertex* of the *feasible region*[2] (see, pp. 117 in [LY08]). We are interested in such a solution, as it reflects a task assignment in which at most $t-1$ tasks are fractionally assigned between different processor types (referred to as *fractional tasks*, hereafter) — see Lemma 40 below. We would like to mention that, if the solution returned by the solver is not a vertex solution then it can always be converted into a vertex solution [Bar04c].

Lemma 40. *Consider an optimal solution for LP-Feas(τ,π), that lies at the vertex of the feasible region. For such a solution, it holds that, at most $t-1$ tasks are fractionally assigned.*

Proof. The proof is based on Fact 2 in [Bar04c]: *"consider a linear program on n variables, in which each variable x_i is subject to the non-negativity constraint, i.e., $x_i \geq 0$. Suppose that there are further m linear constraints. If $m < n$, then at each vertex of the feasible region (including the basic solution), at most m of the variables have non-zero values"*. Clearly, the LP formulation of Figure 6.3 is a linear program on $n' = n \times t + 1$ variables (i.e., $n \times t$ x_i^k variables and one Z variable), all subject to non-negativity constraint, and $m' = n + t$ further linear constraints (n constraints due to R1 plus t constraints due to R2). As $m' < n'$ (we assume $n \geq 2 \wedge t \geq 2$; otherwise the problem becomes trivial), we know from the above fact that, in every optimal solution at the vertex of the feasible region, it holds that, at most $m' = n + t$ variables take non-zero values. Since Z is certain to be non-zero, it holds that:

$$\text{the number of non-zero } x_i^k \text{ variables is } \textit{at most } n + t - 1 \tag{6.5}$$

We know that, for each task $\tau_i \in \tau$, there exists at least one $k \in \{1, 2, \ldots, t\}$ such that $x_i^k > 0$. Let num denote the number of tasks for which there exists at least two k such that, $x_i^k > 0$. It follows from the definition of num that, the total number of non-zero variables is *at least* num $\times 2 + (n -$ num$)$ which can be rewritten as *at least* $n +$ num. If num $\geq t$ then:

$$\text{the number of non-zero } x_i^k \text{ variables is } \textit{at least } n + t \tag{6.6}$$

This contradicts Equation (6.5). Hence, it must be that, num $< t$, which implies that the number of tasks fractionally assigned between different processor types is at most $t - 1$. □

The remaining three steps focus on assigning these (at most) $t - 1$ fractional tasks integrally to processor types.

[2]The *feasible region* of an LP in n-dimensional space is the region over which all the constraints are satisfied. Further, in general, LP solvers (such as CPLEX [IBM12]) always return optimal vertex solution.

6.6 Step 2 of LPG_{IM}: Forming the bi-partite graph

In this step, using the vertex solution, in which at most $t-1$ tasks are fractionally assigned, we construct a bi-partite graph[3]. The graph is constructed with only (i) *fractional tasks* and (ii) those processor types to which at least one fractional task is assigned (sometimes referred to as *fractional processor types*). Hence, while forming the graph, we ignore all the tasks that are integrally assigned and all the processor types to which no fractional task is assigned. Let $\text{G} = (\text{A}, \text{B}, \text{E})$ denote such a bi-partite graph and it is formed as follows:

- each *fractional task*, $\tau_i \in \tau$, is represented by a *task node* $\tau_{\text{i}} \in \text{A}$ defined by a one-to-one mapping.
- each *fractional processor type-k*, $k \in \{1, 2, \ldots, t\}$, is represented by a *processor type node* $\pi^{\text{k}} \in \text{B}$ defined by a one-to-one mapping.
- a task node $\tau_{\text{i}} \in \text{A}$ is connected by an edge, $\text{e}_{\text{i}}^{\text{k}} \in \text{E}$, to a processor type node $\pi^{\text{k}} \in \text{B}$ if and only if $0 < x_i^k < 1$. Each edge, $\text{e}_{\text{i}}^{\text{k}} \in \text{E}$, has a weight set to x_i^k.

Observe that, since the bi-partite graph is constructed only with fractional tasks and fractional processor types, the graph may contain a few *circuits* (defined below).

Definition 23 (Circuit). *A circuit* $C = \{n_1 \to n_2 \to \cdots \to n_s \to n_1\}$ *in a graph* $\text{G} = (\text{A}, \text{B}, \text{E})$ *is a path in which each node is visited exactly once except one node which is visited twice, i.e., both at the start and at the end. Each circuit* C *can also be denoted by a corresponding subgraph,* $\text{G}^{\text{C}} = \left(\text{A}^{\text{C}}, \text{B}^{\text{C}}, \text{E}^{\text{C}}\right) \subseteq \text{G}$*, containing only those nodes and edges that are in* C*.*

For convenience, we use C and G^{C} interchangeably, in the rest of the chapter. The following lemma shows that a circuit in a bi-partite graph is always an *even circuit*.

Lemma 41 (From Theorem 1.2.18 in [Wes00]). *Any circuit* $C = \{n_1 \to n_2 \to \cdots \to n_{s=2N_c} \to n_1\}$*, where* $N_c > 0$ *is a positive integer, in a bi-partite graph* $\text{G} = (\text{A}, \text{B}, \text{E})$*, always has an even number of distinct nodes, with half the number of nodes from the set* A *and the other half from the set* B.

Proof. In cycle, $C = \{n_1 \to n_2 \to \ldots \to n_s \to n_1\}$, let the node n_1 be in set A (abbreviated $n_1 \in \text{A}$). If $n_1 \in \text{A}$ then by definition of bi-partite graph, it must be that $n_2 \in \text{B}$, $n_3 \in \text{A}$, $n_4 \in \text{B}$ and so on. In general, it holds that, $n_{2j+1} \in \text{A}$ and $n_{2j} \in \text{B}$. Since C is a cycle, n_s must be in set B such that $s = 2N_c$ for some positive integer N_c. Therefore, cycle C has even number of nodes (and half the nodes in circuit C are from set A and the other half are from set B). Hence the proof. □

Property 3 (Follows from Lemma 41). *In a circuit* $\text{G}^{\text{C}} = \left(\text{A}^{\text{C}}, \text{B}^{\text{C}}, \text{E}^{\text{C}}\right)$*, it holds that,* $\left|\text{A}^{\text{C}}\right| = \left|\text{B}^{\text{C}}\right| = N_c$*, where* $N_c > 0$ *is a positive integer.*

We now illustrate these concepts with an example.

[3] A bi-partite graph is a graph with two *disjoint* sets of vertices such that every edge connects a vertex in one set to a vertex in the other set.

Tasks	Values of indicator variables				
	x_i^1	x_i^2	x_i^3	x_i^4	x_i^5
τ_1	1	0	0	0	0
τ_2	0	0	0	1	0
τ_3	0.7	0	0	0	0.3
τ_4	0.5	0	0.5	0	0
τ_5	0	0	0	1	0
τ_6	0	0.1	0.5	0	0.4
τ_7	0	0	0	0	1

Table 6.2: Values of x_i^k variables output by the LP solver.

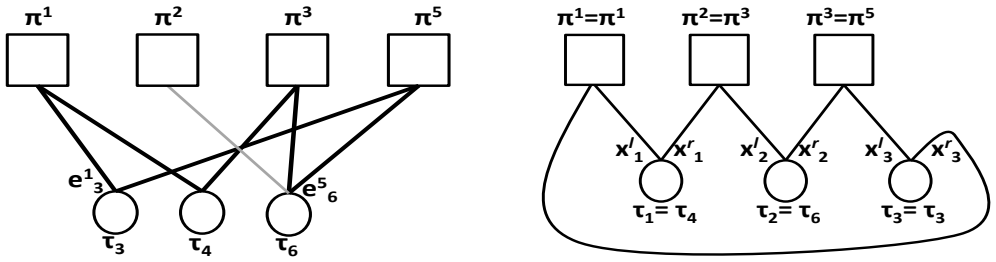
Example 10. *Consider a task set τ of 7 tasks and a t-type platform π with $t = 5$. Let the solution output by the LP solver be as shown in Table 6.2. The bi-partite graph constructed from this solution using the fractional tasks (τ_3, τ_4 and τ_6) and the fractional processor types (type-1, type-2, type-3 and type-5), is shown in Figure 6.4a. As can be seen, there is a circuit $C = \{\tau_3 \to \pi^1 \to \tau_4 \to \pi^3 \to \tau_6 \to \pi^5 \to \tau_3\}$ in the graph, with 6 distinct nodes in which $N_c = 3$ nodes are from the set A and $N_c = 3$ nodes are from the set B. The graph corresponding to this circuit is given by $G^C = (A^C, B^C, E^C)$ where $A^C = \{\tau_3, \tau_4, \tau_6\}$, $B^C = \{\pi^1, \pi^3, \pi^5\}$ and $E^C = \{e_3^1, e_4^1, e_4^3, e_6^3, e_6^5, e_3^5\}$.*

Definition 24 (shared processor type node). *A fractional processor type node $\pi^k \in B$ in a graph $G = (A, B, E)$ is said to be shared only if it is connected to at least two task nodes $\tau_{i1} \in A$ and $\tau_{i2} \in A$. Otherwise, it is said to be non-shared.*

For example, in Figure 6.4a, although all the four nodes, π^1, π^2, π^3 and π^5, are *fractional processor type nodes*, only π^1, π^3 and π^5, are *shared processor type nodes*.

Lemma 42. *If there is no circuit in a graph $G = (A, B, E)$ then there exists at least one task node in A that is connected to at most one shared processor type node in B. Further, since this task is fractional, we know that, it is also connected to at least one non-shared processor type node in B.*

Proof. From Definition 23 and Definition 24, it holds that, in a circuit, each task node is connected to exactly two shared processor type nodes. Thus, it can be easily proven that, if every task node in a graph, $G = (A, B, E)$, is connected to at least two *shared* processor type nodes then there exists at least one circuit in G. Hence, by contraposition, it holds that, if there is no circuit in graph G then it holds that, not every task node is connected to at least two shared processor type nodes. This implies that, if there is no circuit in graph G then there exists at least one task node, $\tau_i \in A$, that is connected to at most one shared processor type node. Since all the task nodes in G are fractional, the task node τ_i must be connected to at least two processor type nodes and hence to at least one *non-shared* processor type node. Hence the proof. □



(a) The bi-partite graph constructed from Table 6.2; circuit is indicated by black edges.

(b) The circuit of Figure 6.4a (with $N_c = 3$ task and $N_c = 3$ processor type nodes) after re-arranging and re-indexing.

Figure 6.4: An example to illustrate the concept of a bi-partite graph (formed from fractional tasks and fractional processor types) and the concept of a circuit.

The circuit shown in Figure 6.4a can be re-arranged as shown in Figure 6.4b. Note in Figure 6.4b that, the nodes are re-indexed. For ease of explanation, we use this notion of re-arranged graph in the next step.

Finally, we define the capacity used on a processor type in a circuit C by the tasks in that circuit as follows.

Definition 25 (Capacity used on a processor type in a circuit). *Consider a circuit,* $\text{G}^{\text{C}} = \left(\text{A}^{\text{C}}, \text{B}^{\text{C}}, \text{E}^{\text{C}}\right)$*, in a graph* G*. The capacity* $\mathscr{C}_{\text{C}}^{j}$ *used on a processor type-j node,* $\pi^{\text{j}} \in \text{B}^{\text{C}}$*, by the task nodes* $\forall\, \tau_{\text{i}} \in \text{A}^{\text{C}}$*, is given by:*

$$\mathscr{C}_{\text{C}}^{j} \stackrel{\text{def}}{=} \sum_{\tau_{\text{i}} \in \text{A}^{\text{C}}:\ \text{x}_{\text{i}}^{\text{j}} > 0} \text{x}_{\text{i}}^{\text{j}} \times \text{u}_{\text{i}}^{\text{j}} \tag{6.7}$$

Remark about notation. In Equation (6.7), index j is used for processor type instead of (the earlier notation) k. This is to avoid any confusion since the processor type nodes are re-indexed in the circuit (as shown in Figure 6.4b).

6.7 Step 3 of LPG_{IM}: Detecting and removing the circuits in the graph

In the graph constructed as described in the previous section, if there are any circuits then we break all such circuits, in this step. Each circuit is broken by re-adjusting the weights of the edges (x_i^j) within the circuit such that the weight of at least one edge becomes zero, which breaks the circuit. The edge whose weight becomes zero is removed from the graph. While manipulating the weights of edges in a circuit $\text{G}^{\text{C}} = \left(\text{A}^{\text{C}}, \text{B}^{\text{C}}, \text{E}^{\text{C}}\right)$, it is ensured that, for each (shared) processor type $\pi^{\text{j}} \in \text{B}^{\text{C}}$, its capacity used by the tasks in the circuit after re-adjusting the weights (denoted by $\mathscr{C}_{\text{C}}^{j'}$) does not exceed its original used capacity, i.e., the used capacity before re-adjusting the weights (denoted by $\mathscr{C}_{\text{C}}^{j}$). Breaking all the circuits reduces the complexity of the problem when

assigning the (at most) $t-1$ fractional tasks integrally to processor types, which is discussed in Section 6.8. We now discuss, in detail, how to detect and remove circuits from the graph.

A circuit in a graph can be detected in polynomial time using *Depth First Search* (DFS) algorithm, generally found in textbooks (e.g., see Chap. 22.3 in [CLRS01]). Hence, we mainly focus on removing the detected circuits in our graph. The following lemma shows how to remove at least one edge in the given circuit without increasing the capacity used on any of the shared processor types that are in the circuit.

Lemma 43. *Consider a circuit* $\mathrm{G^C} = \left(\mathrm{A^C}, \mathrm{B^C}, \mathrm{E^C}\right)$ *(with* N_c *task and* N_c *processor type nodes — see Property 3) arranged as shown in Figure 6.4b. Let* x_i^ℓ *and* x_i^r *denote the fraction of task* τ_i *(*$\forall i \in \{1,2,\ldots,N_c\}$*) that is assigned to the shared processor type which is on* τ_i*'s "left" and* τ_i*'s "right", respectively. From Figure 6.4b and Definition 3,* $\mathscr{C}_\mathrm{C}^j, \forall j \in \{1,2,\ldots,N_c\}$*, can be re-defined as:*

$$\mathscr{C}_\mathrm{C}^j = \begin{cases} (x_{N_c}^\mathrm{r} \times u_{N_c}^\mathrm{r}) + (x_1^\ell \times u_1^\ell) & \text{if } j = 1 \\ (x_{j-1}^\mathrm{r} \times u_{j-1}^\mathrm{r}) + (x_j^\ell \times u_j^\ell) & \text{if } j \in \{2,\ldots,N_c\} \end{cases} \tag{6.8}$$

If it holds that

$$\prod_{g=1}^{N_c} \frac{u_g^\mathrm{r}}{u_g^\ell} \geq 1 \tag{6.9}$$

then after updating the fractional assignments as follows:

$$x_i^{\mathrm{r}'} = x_i^\mathrm{r} - \frac{\varepsilon}{u_i^\ell} \times \prod_{g=1}^{i-1} \frac{u_g^\mathrm{r}}{u_g^\ell} \tag{6.10}$$

$$x_i^{\ell'} = x_i^\ell + \frac{\varepsilon}{u_i^\ell} \times \prod_{g=1}^{i-1} \frac{u_g^\mathrm{r}}{u_g^\ell} \tag{6.11}$$

where $\prod_{g=1}^{i-1} \frac{u_g^\mathrm{r}}{u_g^\ell}$ *is assumed to be* 1 *for* $i = 1$ *and where* $\varepsilon > 0$ *denotes a real number such that*

$$\varepsilon = \min_{z \in [1,2,\ldots,N_c]} \left\{ \frac{x_z^\mathrm{r} \times u_z^\ell}{\prod_{g=1}^{z-1} \frac{u_g^\mathrm{r}}{u_g^\ell}} \right\} \tag{6.12}$$

the following properties are satisfied:

P1. $\forall j \in \{1,2,\ldots,N_c\} : \mathscr{C}_\mathrm{C}^{j'} \leq \mathscr{C}_\mathrm{C}^j$*, where* $\mathscr{C}_\mathrm{C}^{j'}$ *denotes the capacity used on shared processor type* j*, after updating the fractional assignments.*

P2. $\forall i \in \{1,2,\ldots,N_c\} : x_i^{\ell'} + x_i^{\mathrm{r}'} = x_i^\ell + x_i^\mathrm{r}$.

P3. $\forall i \in \{1,2,\ldots,N_c\} : x_i^{\ell'} \geq 0$ *and* $x_i^{\mathrm{r}'} \geq 0$.

P4. $\exists i \in \{1,2,\ldots,N_c\} : x_i^{\mathrm{r}'} = 0$.

Proof. We now prove each of these four properties.

Proof of P1. This will be shown separately for processor type $j = 1$ and $\forall j \in \{2,3,\ldots,N_c\}$.

Case 1: $j = 1$. From Equation (6.8) and Equation (6.10), we have:

$$\text{from Equation (6.8):} \quad \mathscr{C}_{\text{C}}^{1'} = (x_{N_c}^{\text{r}'} \times u_{N_c}^{\text{r}}) + (x_1^{\ell'} \times u_1^{\ell}) \tag{6.13}$$

$$\text{from Equation (6.10):} \quad x_{N_c}^{\text{r}'} = x_{N_c}^{\text{r}} - \frac{\varepsilon}{u_{N_c}^{\ell}} \times \prod_{g=1}^{N_c-1} \frac{u_g^{\text{r}}}{u_g^{\ell}} \tag{6.14}$$

From Equation (6.11) and from the assumption that $\prod_{g=1}^{i-1} \frac{u_g^{\text{r}}}{u_g^{\ell}} = 1$ for $i = 1$, we have:

$$x_1^{\ell'} = x_1^{\ell} + \frac{\varepsilon}{u_1^{\ell}} \tag{6.15}$$

Thus, by substituting Equation (6.14) and Equation (6.15) in Equation (6.13) yields:

$$\begin{aligned}
\mathscr{C}_{\text{C}}^{1'} &= \left(x_{N_c}^{\text{r}} - \frac{\varepsilon}{u_{N_c}^{\ell}} \times \prod_{g=1}^{N_c-1} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \times u_{N_c}^{\text{r}} + \left(x_1^{\ell} + \frac{\varepsilon}{u_1^{\ell}}\right) \times u_1^{\ell} \\
&= (x_{N_c}^{\text{r}} \times u_{N_c}^{\text{r}}) - \varepsilon \times \prod_{g=1}^{N_c} \frac{u_g^{\text{r}}}{u_g^{\ell}} + \left(x_1^{\ell} \times u_1^{\ell}\right) + \varepsilon \\
&\overset{\text{from (6.8)}}{=} \mathscr{C}_{\text{C}}^{1} + \varepsilon \times \left(1 - \prod_{g=1}^{N_c} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \overset{\text{from (6.9)}}{\leq} \mathscr{C}_{\text{C}}^{1}
\end{aligned} \tag{6.16}$$

Case 2: $j \in \{2, \ldots, N_c\}$. From Equation (6.8), Equation (6.10) and Equation (6.11), we have:

$$\mathscr{C}_{\text{C}}^{j'} = (x_{j-1}^{\text{r}'} \times u_{j-1}^{\text{r}}) + (x_j^{\ell'} \times u_j^{\ell}) \tag{6.17}$$

$$x_{j-1}^{\text{r}'} = x_{j-1}^{\text{r}} - \frac{\varepsilon}{u_{j-1}^{\ell}} \times \prod_{g=1}^{j-2} \frac{u_g^{\text{r}}}{u_g^{\ell}} \tag{6.18}$$

$$x_j^{\ell'} = x_j^{\ell} + \frac{\varepsilon}{u_j^{\ell}} \times \prod_{g=1}^{j-1} \frac{u_g^{\text{r}}}{u_g^{\ell}} \tag{6.19}$$

Thus, by substituting Equation (6.18) and Equation (6.19) in Equation (6.17) yields:

$$\begin{aligned}
\mathscr{C}_{\text{C}}^{j'} &= \left(x_{j-1}^{\text{r}} - \frac{\varepsilon}{u_{j-1}^{\ell}} \times \prod_{g=1}^{j-2} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \times u_{j-1}^{\text{r}} + \left(x_j^{\ell} + \frac{\varepsilon}{u_j^{\ell}} \times \prod_{g=1}^{j-1} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \times u_j^{\ell} \\
&= (x_{j-1}^{\text{r}} \times u_{j-1}^{\text{r}}) + (x_j^{\ell} \times u_j^{\ell}) - \left(\varepsilon \times \frac{u_{j-1}^{\text{r}}}{u_{j-1}^{\ell}} \times \prod_{g=1}^{j-2} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) + \left(\varepsilon \times \prod_{g=1}^{j-1} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \\
&= (x_{j-1}^{\text{r}} \times u_{j-1}^{\text{r}}) + (x_j^{\ell} \times u_j^{\ell}) - \left(\varepsilon \times \prod_{g=1}^{j-1} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) + \left(\varepsilon \times \prod_{g=1}^{j-1} \frac{u_g^{\text{r}}}{u_g^{\ell}}\right) \\
&= (x_{j-1}^{\text{r}} \times u_{j-1}^{\text{r}}) + (x_j^{\ell} \times u_j^{\ell}) \\
&\overset{\text{from (6.9)}}{=} \mathscr{C}_{\text{C}}^{j}
\end{aligned} \tag{6.20}$$

From Equation (6.16) and Equation (6.20), it can be seen that, performing operations shown in Equation (6.10) and Equation (6.11) satisfies property **P1**.

Proof of P2. For every $i \in \{1,\ldots,N_c\}$, it can be seen that adding Equation (6.10) and Equation (6.11) results in $x_i^{\ell'} + x_i^{\mathrm{r}'} = x_i^{\ell} + x_i^{\mathrm{r}}$, and hence the property immediately follows.

Proof of P3. Since $\varepsilon > 0$, it is trivial from Equation (6.11) that, $\forall i \in \{1,\ldots,N_c\}$: $x_i^{\ell'} > x_i^{\ell} > 0$. Then, from Equation (6.10), any $x_i^{\mathrm{r}'}$ will be negative if and only if

$$\begin{aligned} x_i^{\mathrm{r}} &< \frac{\varepsilon}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} \\ &\overset{\text{from (6.12)}}{<} \min_{z \in \{1,2,\ldots,N_c\}} \left\{ \frac{x_z^{\mathrm{r}} \times u_z^{\ell}}{\prod_{g=1}^{z-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}} \right\} \times \frac{1}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} \end{aligned}$$

Since the min term evaluates to $\leq \frac{x_i^{\mathrm{r}} \times u_i^{\ell}}{\prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}}$, we have:

$$\begin{aligned} x_i^{\mathrm{r}} &< \frac{x_i^{\mathrm{r}} \times u_i^{\ell}}{\prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}} \times \frac{1}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} < \frac{x_i^{\mathrm{r}} \times u_i^{\ell}}{u_i^{\ell}} \\ &< x_i^{\mathrm{r}} \end{aligned}$$

which is impossible. Hence $x_i^{\mathrm{r}'} \geq 0$.

Proof of P4. From Equation (6.12), it holds that:

$$\exists i \in \{1,2,\ldots,N_c\} : \varepsilon = \frac{x_i^{\mathrm{r}} \times u_i^{\ell}}{\prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}} \tag{6.21}$$

For such i, Equation (6.10) can be re-written as:

$$x_i^{\mathrm{r}'} = x_i^{\mathrm{r}} - \frac{\varepsilon}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} \tag{6.22}$$

Substituting the value of ε, we obtain, $\exists i \in \{1,2,\ldots,N_c\} : x_i^{\mathrm{r}'} = 0$. Hence the property holds.

As a conclusion, we showed that modifying the fractional assignments of the tasks according to Equation (6.10) and Equation (6.11) ensures that all the four properties P1, P2, P3 and P4 are satisfied. Hence the proof. □

Lemma 43 showed that, in a circuit with N_c task nodes, if $\prod_{g=1}^{N_c} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} \geq 1$ then transferring the fractions from "right to left" within the circuit will (i) delete an edge (as its weight becomes zero, by P4) so that the circuit breaks and (ii) ensures that, $\forall j \in \{1,2,\ldots,N_c\} : \mathscr{C}_{\mathrm{C}}^{j'} \leq \mathscr{C}_{\mathrm{C}}^{j}$. Since no fraction was moved to/from those processor types that are in set B but not in circuit C, their capacities remain unaffected. Hence, $\forall \pi^j \in B : \mathscr{C}_{\mathrm{C}}^{j'} \leq \mathscr{C}_{\mathrm{C}}^{j}$. Analogously, it can be shown that if $\prod_{g=1}^{N_c} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} < 1$ then transferring the fractions from "left to right" within the circuit will also yield the same result. The claim is presented formally below in Lemma 44 but the formal proof is omitted since it is very similar to the proof of Lemma 43.

Lemma 44. *Consider a circuit* $\mathrm{G}^{\mathrm{C}} = \left(\mathrm{A}^{\mathrm{C}}, \mathrm{B}^{\mathrm{C}}, \mathrm{E}^{\mathrm{C}}\right)$ *(with* N_c *task and* N_c *processor type nodes — see Property 3) arranged as shown in Figure 6.4b. Let* x_i^{ℓ} *and* x_i^{r} *denote the fraction of task* τ_i *(*$\forall i \in \{1,2,\ldots,N_c\}$*) that is assigned to the shared processor type which is on* τ_i*'s "left" and* τ_i*'s "right", respectively. From Figure 6.4b and Definition 3,* $\mathscr{C}_{\mathrm{C}}^{j}, \forall j \in \{1,2,\ldots,N_c\}$*, can be re-defined as:*

$$\mathscr{C}_{\mathrm{C}}^{j} = \begin{cases} (x_{N_c}^{\mathrm{r}} \times u_{N_c}^{\mathrm{r}}) + (x_1^{\ell} \times u_1^{\ell}) & \text{if } j = 1 \\ (x_{j-1}^{\mathrm{r}} \times u_{j-1}^{\mathrm{r}}) + (x_j^{\ell} \times u_j^{\ell}) & \text{if } j \in \{2,\ldots,N_c\} \end{cases}$$

If it holds that

$$\prod_{g=1}^{N_c} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} < 1$$

then after updating the fractional assignments as follows:

$$x_i^{\mathrm{r}'} = x_i^{\mathrm{r}} + \frac{\varepsilon}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}} \qquad \text{and} \qquad x_i^{\ell'} = x_i^{\ell} - \frac{\varepsilon}{u_i^{\ell}} \times \prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}$$

where $\prod_{g=1}^{i-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}$ *is assumed to be* 1 *for* $i = 1$ *and where* $\varepsilon > 0$ *denotes a real number such that*

$$\varepsilon = \min_{z \in [1,2,\ldots,N_c]} \left\{ \frac{x_z^{\mathrm{r}} \times u_z^{\ell}}{\prod_{g=1}^{z-1} \frac{u_g^{\mathrm{r}}}{u_g^{\ell}}} \right\}$$

the following properties are satisfied:

P1. $\forall j \in \{1,2,\ldots,N_c\} : \mathscr{C}_{\mathrm{C}}^{j'} \leq \mathscr{C}_{\mathrm{C}}^{j}$*, where* $\mathscr{C}_{\mathrm{C}}^{j'}$ *denotes the capacity used on shared processor type* j*, after updating the fractional assignments.*

P2. $\forall i \in \{1,2,\ldots,N_c\} : x_i^{\ell'} + x_i^{\mathrm{r}'} = x_i^{\ell} + x_i^{\mathrm{r}}$.

P3. $\forall i \in \{1,2,\ldots,N_c\} : x_i^{\ell'} \geq 0$ *and* $x_i^{\mathrm{r}'} \geq 0$.

P4. $\exists i \in \{1,2,\ldots,N_c\} : x_i^{\ell'} = 0$.

Proof. The proof is analogous to the proof of Lemma 43. □

Thus, each circuit identified in the graph (for example, using DFS [CLRS01]) can be broken using the procedure described above (i.e., either using Lemma 43 or Lemma 44). Observe that, while removing the circuits, zero or more fractional tasks may get integrally assigned to processor types but for all practical purposes, it is sufficient for us to know that, at the end of this step, (i) there are at most $t-1$ fractional tasks (by Lemma 40) and (ii) there are no circuits in the graph anymore (by repeatedly applying Lemma 43 and/or Lemma 44). In the final step, we integrally assign these (at most) $t-1$ fractional tasks to processor types.

6.8 Step 4 of LPG_{IM}: Integrally assigning the fractional tasks

In this section, we describe how to assign the fractional tasks *integrally* to processor types. This fourth step takes as input the output of the previous step, i.e, *a graph* $\text{G} = (\text{A}, \text{B}, \text{E})$ *with no circuits* and with *at most* $t-1$ fractional tasks, and works iteratively on this input graph. In each iteration y, our algorithm chooses *one* fractional task $\tau_i \in \text{A}$ and assigns it integrally to one of the processor types in B. Then, it removes that fractional task node from *A*, deletes all the edges incident on τ_i and removes from B all the *non-shared* processor type nodes to which τ_i was fractionally assigned. This procedure of integrally assigning a task and then deleting a few nodes and edges is repeated until the graph becomes empty, which implies that all the fractional tasks have been integrally assigned to processor types.

We now introduce two additional sets of notations that we will use extensively in the rest of the section while describing the working of this fourth step and proving its correctness. The first set of notations can be seen as *"global" with respect to the input graph* G while the second set of notations can be seen as *"local" with respect to each task in the graph*.

Global notations w.r.t. the graph. Recall that, in this step, we use the circuit-free graph, $\text{G} = (\text{A}, \text{B}, \text{E})$, output by the previous step. Since this graph contains only fractional tasks and fractional processor types (see Section 6.6), this step deals with only these tasks and processor types. For the purpose of this section, we re-index the fractional tasks in A and the fractional processor types in B as follows. In graph $\text{G} = (\text{A}, \text{B}, \text{E})$, let τ_i denote the i'th task (node) in A and let π^j denote the j'th processor type (node) in B. Since this step works iteratively, let y denote the current iteration. During this step, assigning a fractional task integrally to one of the processor types comes at the cost of additional computing capacity required on the processor type for accommodating this task *entirely*. We denote by $\mathscr{C}_+^j[\text{y}]$ the *cumulative* extra capacity required on processor type $\pi^j \in \text{B}$ from iteration 1 until the beginning of iteration y. Since some of the processor type nodes are deleted from the graph at the end of each iteration, let $\text{P}^{\text{in}}[\text{y}]$ denote the set of processor type nodes that are still in the graph at the beginning of iteration y and let $\text{P}^{\text{out}}[\text{y}]$ denote the set of all the processor types that have been removed from the graph from iteration 1 till the beginning of iteration y. It holds by definition that, $\text{P}^{\text{in}}[1] = \text{B}$ and $\text{P}^{\text{out}}[1] = \phi$.

For example, let the circuit in the graph shown in Figure 6.4a (in the previous section), be broken by removing the edge e_3^5. In that case, the graph output by the previous step (i.e., Step 3 of LPG_{IM}), after re-indexing the task and processor types, is shown in Figure 6.5. In Figure 6.5, the re-indexed task nodes τ_1, τ_2 and τ_3 denote the original task nodes τ_3, τ_4 and τ_6 of Figure 6.4a, respectively. Analogously, the re-indexed processor type nodes π^1, π^2, π^3 and π^4 denote the original processor type nodes π^1, π^2, π^3 and π^5 of Figure 6.4a, respectively.

Local notations w.r.t. a task in the graph. Since this fourth step of LPG_{IM} considers one fractional task, $\tau_i \in \text{A}$, in each iteration and assigns it integrally to one of the processor types to which it is *fractionally* assigned, we also define some notations with respect to task τ_i. Let $\pi(i) = \left\{\pi^1(i), \pi^2(i), \ldots, \pi^{|\pi(i)|}(i)\right\}$ denote the set of fractional processor types to which task, $\tau_i \in \text{A}$, is fractionally assigned in G, where $\forall j \in \{1, 2, \ldots, |\pi(i)|\}$, $\pi^j(i) \in \pi(i)$ denote the j'th processor

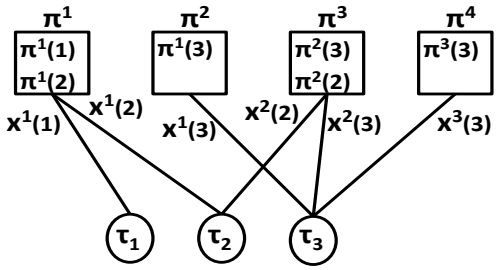


Figure 6.5: The graph of Figure 6.4a after breaking the circuit (as described in Section 6.7) and re-indexing the nodes.

type to which task τ_i is assigned. Let $X(i) = \left\{x^1(i), x^2(i), \ldots, x^{|\pi(i)|}(i)\right\}$ denote the set of fractional assignments of task, $\tau_i \in \text{A}$, where $\forall j \in \{1, 2, \ldots, |\pi(i)|\}$, $x^j(i) \in X(i)$ denotes the fraction of task τ_i that is assigned to *its* j'th processor type, i.e., the fraction that is assigned to $\pi^j(i)$. Let $\mathscr{C}^j_+(i)[\text{y}]$ denote the cumulative extra capacity required on processor type $\pi^j(i)$ from iteration 1 to iteration y.

Note that these two sets of notations, i.e., *global* and *local*, can be used to refer to the same processor type node. For example, in Figure 6.5, we can observe the following: $\pi(1) = \left\{\pi^1(1) = \pi^1\right\}$, $\pi(2) = \left\{\pi^1(2) = \pi^1,\ \pi^2(2) = \pi^3\right\}$ and $\pi(3) = \left\{\pi^1(3) = \pi^2,\ \pi^2(3) = \pi^3,\ \pi^3(3) = \pi^4\right\}$. Hence, for example, processor π^1 is referred to as $\pi^1(1)$ in the context of task τ_1 and is also referred to as $\pi^1(2)$ in the context of task τ_2.

Finally, since G is formed using only fractional tasks and fractional processor types (see Section 6.6), observe that:

$$\forall \tau_i \in \text{A} : \sum_{j=1}^{|\pi(i)|} x^j(i) = 1 \tag{6.23}$$

With these new notations, we now describe the working of this fourth step of LPG_{IM} algorithm.

The pseudo-code of the fourth step is provided in Algorithm 15 and it can be summarized as follows. As long as there are task nodes in the graph, Algorithm 15 chooses a task τ_i from the graph which is connected to *only* non-shared processor type nodes (line 3–4). If there is no such task then it chooses a task which is connected to *exactly* one shared processor type node (line 5–6) — we will prove in Lemma 46 that there always exists such a task. Then, Algorithm 15 *tries* to integrally assign the chosen task τ_i to one of its non-shared processor types. We say that it *fails* to assign τ_i to a processor type $\pi^\ell(i) \in \pi(i)$ if the (cumulative) extra capacity required on $\pi^\ell(i)$, after assigning τ_i to it, exceeds $\alpha \times \frac{t-1}{t}$. If the extra capacity does not exceed that threshold on any one of the non-shared processor types then τ_i is integrally assigned to that processor type (lines 8–15). Otherwise, τ_i is assigned to its (sole) shared processor type (lines 16–19); we show in Lemma 46 that this assignment cannot fail. Finally, the algorithm removes τ_i from the graph, as well as all its non-shared processor type nodes and all the edges connected to τ_i (lines 21–26), and iterates with another task until the graph becomes empty.

Now, we prove the speed competitive ratio of the intra-migrative algorithm, LPG_{IM}, with the help of Property 4 and an intermediate lemma, Lemma 45.

Algorithm 15: Step 4 of LPG_{IM} algorithm for assigning the fractional tasks integrally to processor types.

Input : $\text{G} = (\text{A}, \text{B}, \text{E})$: A graph output by Step 3 of LPG_{IM} representing task assignment with no circuits and at most $t-1$ fractional tasks

1 $\text{y} \leftarrow 1$, $\text{P}^{\text{in}}[\text{y}] \leftarrow \text{B}$, $\text{P}^{\text{out}}[\text{y}] \leftarrow \phi$;
2 **while** A *is not empty* **do**
3 **if** $\exists \tau_\ell \in \text{A}$ *connected to only non-shared processor types* **then**
4 $\tau_i \leftarrow \tau_\ell$;
5 **else**
6 $\tau_i \leftarrow$ a task in A that is connected to exactly one shared processor type ;
7 **end**
8 **foreach** *non-shared processor type* $\pi^\ell(i) \in \pi(i)$ **do**
9 newCap $\leftarrow \mathscr{C}_+^\ell(i)[\text{y}] + \sum_{j=1, j\neq\ell}^{|\pi(i)|} x^j(i) \times u_i^j$;
10 **if** newCap $\leq \alpha \times \frac{t-1}{t}$ **then**
11 assign τ_i to $\pi^\ell(i)$;
12 $\mathscr{C}_+^\ell(i)[\text{y}] \leftarrow$ newCap ;
13 break the `foreach` loop ;
14 **end**
15 **end**
16 **if** τ_i *is not assigned* **then**
17 assign τ_i to the only shared processor type, say $\pi^z(i)$, to which it is connected ;
18 $\mathscr{C}_+^z(i)[\text{y}] \leftarrow \mathscr{C}_+^z(i)[\text{y}] + \sum_{j=1, j\neq z}^{|\pi(i)|} x^j(i) \times u_i^j$;
19 **end**
  `// remove (i) the task` τ_i `from` A `and (ii) all the non-shared processor types that are connected to` τ_i `from` B `(and the edges connecting` τ_i `to these processor types`
20 $\text{y} \leftarrow \text{y}+1$;
21 $\text{A} \leftarrow \text{A} \setminus \{\tau_\text{i}\}$;
22 $\text{delpt} \leftarrow \{\pi^\text{k} \in \text{B} \mid \exists x_i^k > 0$ and π^k is non-shared$\}$;
23 $\text{B} \leftarrow \text{B} \setminus \text{delpt}$;
24 $\text{P}^{\text{in}}[\text{y}] \leftarrow \text{P}^{\text{in}}[\text{y}] \setminus \text{delpt}$;
25 $\text{P}^{\text{out}}[\text{y}] \leftarrow \text{P}^{\text{out}}[\text{y}] \cup \text{delpt}$;
26 $\text{E} \leftarrow \text{E} \setminus \{\text{e}_\text{i}^\text{k} \mid \pi^k \in \pi(i)$ and π^k is non-shared$\}$;
27 **end**

Property 4. *It holds, from lines 21–26 of Algorithm 15, that at each iteration* y*:*

$$\text{P}^{\text{in}}[\text{y}] \cup \text{P}^{\text{out}}[\text{y}] = \text{B} \quad \textit{and} \quad \text{P}^{\text{in}}[\text{y}] \cap \text{P}^{\text{out}}[\text{y}] = \emptyset \tag{6.24}$$

Lemma 45. $\forall \tau_i \in \text{A}$, $\exists \pi^j(i) \in \pi(i)$ *such that* $x^j(i) \geq \frac{1}{|\pi(i)|}$.

Proof. The proof is by contradiction. If $\forall \pi^j(i) \in \pi(i)$, if $x^j(i) < \frac{1}{|\pi(i)|}$ then $\sum_{j=1}^{|\pi(i)|} x^j(i) < |\pi(i)| \times \frac{1}{|\pi(i)|} < 1$, which contradicts Equation (6.23). Hence the proof. □

Lemma 46. *Consider a task set* τ *which is intra-migrative feasible on a platform* π*. After running steps 1 to 3 of* LPG_{IM}*, if the graph* $\text{G} = (\text{A}, \text{B}, \text{E})$ *(with no circuits and at most* $t-1$ *fractional tasks)*

that was output by step 3, is given as input to Algorithm 15 (step 4 of LPG_{IM}*) then Algorithm 15 succeeds to integrally assign the at most* $t-1$ *fractional tasks in* A *to the processor types in* B *and in order to succeed it only requires that each processor type in* B *are provided with an additional capacity of* $\alpha \times \frac{t-1}{t}$.

Proof. The proof is split into three parts where we show:

Part 1. At lines 3–7, there always exists, at the beginning of each iteration y, a task τ_i assigned to *at most* one shared processor type.

Part 2. At the beginning of the first iteration ($\text{y} = 1$), it holds that $\sum_{\pi^j \in \text{P}^{\text{in}}[1]} \mathscr{C}^j_+[1] \leq \alpha \times \frac{|\text{P}^{\text{out}}[1]|}{t}$.

Part 3. At the beginning of each iteration $\text{y} \geq 1$, if it holds that

$$\sum_{\pi^j \in \text{P}^{\text{in}}[\text{y}]} \mathscr{C}^j_+[\text{y}] \leq \frac{|\text{P}^{\text{out}}[\text{y}]|}{t} \times \alpha \tag{6.25}$$

then the task τ_i chosen on line 4 (or line 6) can be assigned *integrally* to one of its non-shared processor types on lines 8–15 (or, to its (sole) shared processor type on lines 16–19). Then, after assigning τ_i integrally, Equation (6.25) remains satisfied at the beginning of the next iteration $\text{y}+1$.

Proof of Part 1. Here we show that, at the beginning of each iteration y, there always exists a task τ_i which is assigned to *at most* one shared processor type. Since the input graph, $\text{G} = (\text{A}, \text{B}, \text{E})$, does not contain any circuit, we know from Lemma 42 that, at the first iteration of Algorithm 15, there is a task, $\tau_i \in \text{A}$, which is assigned to *at most* one shared processor type. Then, at the end of each iteration, $\text{y} \geq 1$, one task is deleted from the graph (line 21) and all the non-shared processor types connected to that task are also deleted (lines 22–25). Since removing nodes and edges from the graph cannot create a new circuit, the graph will always be circuit-free in all the subsequent iterations of Algorithm 15. Hence, from Lemma 42, at every iteration, $\text{y} \geq 1$, there is always a task, $\tau_i \in \text{A}$, assigned to *at most* one shared processor type.

Proof of Part 2. Here we show that at the beginning of the first iteration ($\text{y} = 1$), it holds that, $\sum_{\pi^j \in \text{P}^{\text{in}}[1]} \mathscr{C}^j_+[1] \leq \frac{|\text{P}^{\text{out}}[1]|}{t} \times \alpha$. At the beginning of the first iteration, no fractional task in G has been integrally assigned to a processor type yet. Hence, the extra capacity needed on each processor type to accommodate the tasks in G that have been already integrally assigned is trivially zero, i.e., $\mathscr{C}^j_+[1] = 0$, $\forall \pi^j \in \text{B}$. Besides, we have $\text{P}^{\text{out}}[1] = \phi$ and thus it holds that, $\sum_{\pi^j \in \text{P}^{\text{in}}[1]} \mathscr{C}^j_+[1] = 0 \leq \frac{|\text{P}^{\text{out}}[1]|}{t} \times \alpha = 0$.

Proof of Part 3. Here we show that, at each iteration y, as long as Equation (6.25) holds (and we have shown above that, it holds for $\text{y} = 1$), the fractional task, τ_i, chosen at line 4 (or line 6), can be integrally assigned to one of the processor types connected to it. For this, we need to investigate three cases:

Case 3.1. Task τ_i is *not* assigned to a shared processor type (chosen on line 4). In this case, we need to show that τ_i can be integrally assigned to at least one of its non-shared processor type (on lines 8–15) and Equation (6.25) holds true at the beginning of the next iteration, $\text{y}+1$.

Case 3.2. Task τ_i is assigned to exactly one shared processor type (chosen on line 6) and is integrally assigned to (one of) its non-shared processor types on line 8–15. In this case, we only

have to show that Equation (6.25) holds true at the beginning of the next iteration, y+1.

Case 3.3. Task τ_i is assigned to exactly one shared processor type (chosen on line 6) and fails to be assigned to any of its non-shared processor types. In this case, we need to show that Algorithm 15 succeeds in integrally assigning τ_i to its shared processor type on lines 16–19 and Equation (6.25) holds true at the beginning of the next iteration, y+1.

In the three cases proven below, we assume that Equation (6.25) holds true at the beginning of iteration, y, and then show that it also holds at the beginning of iteration, y+1.

Proof of Case 3.1. We prove this case by contradiction, i.e., we assume that Algorithm 15 tried to integrally assign the task τ_i to every non-shared processor type (to which τ_i is fractionally assigned) but failed to do so and then we show that it is impossible for this to happen. From the case, task τ_i failed to be integrally assigned to its non-shared processor types, which means that for *every* processor type node $\pi^j(i) \in \pi(i)$, migrating all the fractional assignments of task τ_i to processor type $\pi^j(i)$ requires an extra capacity on that processor type, j, which is greater than $\alpha \times \frac{t-1}{t}$, i.e., the following $|\pi(i)|$ inequalities hold:

$$\begin{aligned} & \forall \ell \in [1, |\pi(i)|] : \textstyle\sum_{\substack{j=1 \\ j \neq \ell}}^{|\pi(i)|} \left(x^j(i) \times u_i^\ell \right) + \mathscr{C}_+^\ell(i)[\mathrm{y}] \;\; > \;\; \alpha \times \frac{t-1}{t} \\ \overset{\text{re-writing}}{\Leftrightarrow} \quad & \forall \ell \in [1, |\pi(i)|] : \textstyle\sum_{\substack{j=1 \\ j \neq \ell}}^{|\pi(i)|} \left(x^j(i) \times u_i^\ell \right) \;\; > \;\; \alpha \times \frac{t-1}{t} - \mathscr{C}_+^\ell(i)[\mathrm{y}] \end{aligned}$$

By summing these $|\pi(i)|$ inequalities, we get

$$\sum_{\ell=1}^{|\pi(i)|} \sum_{\substack{j=1 \\ j \neq \ell}}^{|\pi(i)|} \left(x^j(i) \times u_i^\ell \right) \;\; > \;\; \left(|\pi(i)| \times \alpha \times \frac{t-1}{t} \right) - \sum_{\ell=1}^{|\pi(i)|} \mathscr{C}_+^\ell[\mathrm{y}] \tag{6.26}$$

In the left-hand side of Equation (6.26), each $x^j(i)$ appears $(|\pi(i)|-1)$ times and since $\forall \ell$, $u_i^\ell \leq \alpha$ (from Equation (6.2)), for the left-hand side of Equation (6.26), we have:

$$\begin{aligned} \sum_{\ell=1}^{|\pi(i)|} \sum_{\substack{j=1 \\ j \neq \ell}}^{|\pi(i)|} x^j(i) \times u_i^\ell \quad & \leq \quad \alpha \times (|\pi(i)|-1) \times \sum_{j=1}^{|\pi(i)|} x^j(i) \\ & \overset{\text{from (6.23)}}{=} \alpha \times (|\pi(i)|-1) \end{aligned} \tag{6.27}$$

Regarding the right-hand side of Equation (6.26), since we know that $\pi(i) \subseteq \mathrm{P}^{\mathrm{in}}[\mathrm{y}]$, we have

$$\sum_{\ell=1}^{|\pi(i)|} \mathscr{C}_+^\ell(i)[\mathrm{y}] \;\; \leq \sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}]} \mathscr{C}_+^j[\mathrm{y}] \overset{\text{from (6.25)}}{\leq} \alpha \times \frac{|\mathrm{P}^{\mathrm{out}}[\mathrm{y}]|}{t}$$

Therefore, for the right-hand side of Equation (6.26), we have:

$$\left(|\pi(i)| \times \alpha \times \frac{t-1}{t} \right) - \left(\sum_{\ell=1}^{|\pi(i)|} \mathscr{C}_+^\ell(i)[\mathrm{y}] \right) \;\; \geq \;\; \left(|\pi(i)| \times \alpha \times \frac{t-1}{t} \right) - \left(\alpha \times \frac{|\mathrm{P}^{\mathrm{out}}[\mathrm{y}]|}{t} \right) \tag{6.28}$$

By combining Equation (6.26), (6.27) and (6.28), we obtain:

$$\alpha \times (|\pi(i)| - 1) \;>\; \left(|\pi(i)| \times \alpha \times \frac{t-1}{t}\right) - \left(\alpha \times \frac{|\mathrm{P}^{\text{out}}[\mathrm{y}]|}{t}\right) \tag{6.29}$$

Then, since $\pi(i) \subseteq \mathrm{P}^{\text{in}}[\mathrm{y}]$, we have $\left|\mathrm{P}^{\text{in}}[\mathrm{y}]\right| \geq |\pi(i)|$ and thus $|\mathrm{P}^{\text{out}}[\mathrm{y}]| \leq t - |\pi(i)|$. Using this, Equation (6.29) is re-written as:

$$\begin{aligned}
&\alpha \times (|\pi(i)| - 1) \;>\; \left(|\pi(i)| \times \alpha \times \frac{t-1}{t}\right) - \left(\alpha \times \frac{t - |\pi(i)|}{t}\right) \\
\Leftrightarrow\; &|\pi(i)| - \left(|\pi(i)| \times \frac{t-1}{t}\right) \;>\; \frac{|\pi(i)|}{t} \\
\Leftrightarrow\; &\frac{1}{t} \;>\; \frac{1}{t}
\end{aligned}$$

which is impossible. Hence, this contradicts the assumption that the task τ_i could not be integrally assigned to any of its non-shared processor types and hence Algorithm 15 succeeds in doing so. This concludes Case 3.1.

Proof of Case 3.2. Task τ_i is assigned to exactly one shared processor type and is successfully assigned integrally on lines 8–15 to (one of) its non-shared processor types in $\pi(i)$. Here, we only need to show that Equation (6.25) holds true at the beginning of the next iteration, $\mathrm{y}+1$. The proof is somewhat similar to that of Case 3.1. Let us assume, without loss of generality that, $\pi^1(i)$ is the shared processor type in $\pi(i)$. After assigning τ_i to (one of) its *non-shared* processor type, we get:

$$|\,\mathrm{P}^{\text{out}}[\mathrm{y}+1]\,| \;=\; |\,\mathrm{P}^{\text{out}}[\mathrm{y}]\,| + |\pi(i)| - 1 \tag{6.30}$$

$$|\,\mathrm{P}^{\text{in}}[\mathrm{y}+1]\,| \;=\; |\,\mathrm{P}^{\text{in}}[\mathrm{y}]\,| - |\pi(i)| + 1 \tag{6.31}$$

The "-1" and "+1" is the shared processor type node, $\pi^1(i) \in \pi(i)$, which is *not* removed from the graph. Hence, the processor type node, $\pi^1(i)$, remains in $\mathrm{P}^{\text{in}}[\mathrm{y}+1]$ and is not added to $\mathrm{P}^{\text{out}}[\mathrm{y}+1]$. As explained in Case 3.1, since the task τ_i is integrally assigned to (one of) its non-shared processor type, say $\pi^\ell(i)$, and since $\pi^\ell(i) \notin \mathrm{P}^{\text{in}}[\mathrm{y}+1]$ as $\pi^\ell(i)$ is removed from graph, we have

$$\sum_{\pi^j \in \mathrm{P}^{\text{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}+1] = \sum_{\pi^j \in \mathrm{P}^{\text{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}] \tag{6.32}$$

and since $\mathrm{P}^{\text{in}}[\mathrm{y}+1] \subset \mathrm{P}^{\text{in}}[\mathrm{y}]$, we can rewrite Equation (6.32) as:

$$\begin{aligned}
\sum_{\pi^j \in \mathrm{P}^{\text{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}+1] \;\leq\; \sum_{\pi^j \in \mathrm{P}^{\text{in}}[\mathrm{y}]} \mathscr{C}_+^j[\mathrm{y}] \;&\overset{\text{from (6.25)}}{\leq}\; \alpha \times \frac{|\mathrm{P}^{\text{out}}[\mathrm{y}]|}{t} \\
&\overset{\text{from (6.30)}}{=}\; \alpha \times \left(\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}+1]|}{t} - \frac{|\pi(i)| - 1}{t}\right) \\
&<\; \alpha \times \frac{|\mathrm{P}^{\text{out}}[\mathrm{y}+1]|}{t} \text{ (since } |\pi(i)| \geq 2)
\end{aligned}$$

This concludes Case 3.2.

Proof of Case 3.3. Task τ_i is assigned to exactly one shared processor type and fails to be integrally assigned on lines 8–15 to any of its non-shared processor types in $\pi(i)$. In this case, we need to show that, Algorithm 15 succeeds in integrally assigning τ_i to its (sole) shared processor type on lines 16–19 and Equation (6.25) holds true at the beginning of the next iteration, y+1. As in the previous case, let us assume, without loss of generality that, $\pi^1(i) \in \pi(i)$ is the shared processor type connected to τ_i. We prove by contradiction that the integral assignment of task τ_i to processor type $\pi^1(i)$ cannot fail, i.e., by contradiction, we assume that it does fail and then show that it is impossible for this to happen.

From the case, task τ_i also failed to be assigned to *all* its non-shared processor types, $\pi^j(i) \in \pi(i) \ \wedge \ j \neq 1$ (on lines 8–15), which means that, for *every* processor type node, $\pi^j(i) \in \pi(i)$, migrating all the fractional assignments of task τ_i to that node $\pi^j(i)$ requires an extra capacity on that processor type j exceeding $\alpha \times \frac{t-1}{t}$. This scenario is same as Case 3.1 and thus it leads to a contradiction. Hence, the assumption that, Algorithm 15 fails to integrally assign τ_i to its only shared processor type $\pi^1(i)$ is not true and therefore, Algorithm 15 must succeed in doing so.

Now, we assume that τ_i is integrally assigned to $\pi^1(i) \in \pi(i)$ in iteration y and show that, Equation (6.25) still holds at the beginning of the next iteration, y+1. Assigning task τ_i integrally to processor type $\pi^1(i)$ gives us:

$$\mathscr{C}^1_+(i)[\mathrm{y}+1] = \mathscr{C}^1_+(i)[\mathrm{y}] + \sum_{j=2}^{|\pi(i)|} \left(x^j(i) \times u^1_i\right) \tag{6.33}$$

As explained earlier, since the algorithm failed to assign the task, τ_i, integrally to each of the *non-shared* processor types, it holds $\forall \pi^\ell(i) \in \pi(i), \pi^\ell(i) \neq \pi^1(i)$ that:

$$\sum_{\substack{j=1 \\ j\neq\ell}}^{|\pi(i)|} \left(x^j(i) \times u^\ell_i\right) \ > \ \left(\alpha \times \frac{t-1}{t}\right) - \mathscr{C}^\ell_+(i)[\mathrm{y}] \tag{6.34}$$

Since we know from Equation (6.23) that, $\sum_{j=1}^{|\pi(i)|} x^j(i) = 1$, we obtain $\forall \ell \in [1, |\pi(i)|]$: $\sum_{\substack{j=1 \\ j\neq\ell}}^{|\pi(i)|} x^j(i) = 1 - x^\ell(i)$. Using this, Equation (6.34) can be re-written as: $\forall \pi^\ell(i) \in \pi(i), \pi^\ell(i) \neq \pi^1(i)$, it holds that:

$$\begin{aligned}
& (1 - x^\ell(i)) \times u^\ell_i \ > \left(\alpha \times \frac{t-1}{t}\right) - \mathscr{C}^\ell_+(i)[\mathrm{y}] \\
\overset{\text{from (6.2)}}{\Leftrightarrow} \quad & \alpha \times \left(1 - x^\ell(i)\right) \ > \left(\alpha \times \frac{t-1}{t}\right) - \mathscr{C}^\ell_+(i)[\mathrm{y}] \\
\overset{\text{re-writing}}{\Leftrightarrow} \quad & \alpha \times x^\ell(i) \ < \alpha - \left(\alpha \times \frac{t-1}{t}\right) + \mathscr{C}^\ell_+(i)[\mathrm{y}] \\
\overset{\text{re-writing}}{\Leftrightarrow} \quad & x^\ell(i) \ < \frac{\frac{\alpha}{t} + \mathscr{C}^\ell_+(i)[\mathrm{y}]}{\alpha}
\end{aligned} \tag{6.35}$$

By using Equation (6.35) in Equation (6.33), we get

$$\begin{aligned}
\mathscr{C}^1_+(i)[\mathrm{y}+1] &\leq \mathscr{C}^1_+(i)[\mathrm{y}] + \sum_{j=2}^{|\pi(i)|}\left(\frac{\frac{\alpha}{t}+\mathscr{C}^\ell_+(i)[\mathrm{y}]}{\alpha}\right)\times u^1_i \\
&\overset{\text{from (6.2)}}{\leq} \mathscr{C}^1_+(i)[\mathrm{y}] + \sum_{j=2}^{|\pi(i)|}\left(\frac{\alpha}{t}+\mathscr{C}^\ell_+(i)[\mathrm{y}]\right) \\
&\leq \mathscr{C}^1_+(i)[\mathrm{y}] + \left(\alpha\times\frac{|\pi(i)|-1}{t}\right) + \sum_{j=2}^{|\pi(i)|}\mathscr{C}^\ell_+(i)[\mathrm{y}] \qquad (6.36)
\end{aligned}$$

Now, let us focus on the term, $\sum_{j=2}^{|\pi(i)|}\mathscr{C}^j_+(i)[\mathrm{y}]$, from the right-hand side of the above inequality. Since we know that:
$\pi(i)\setminus\{\pi^1(i)\} = \mathrm{P}^{\text{in}}[\mathrm{y}]\setminus(\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i))\setminus\pi^1(i)$, we can write:

$$\begin{aligned}
\sum_{j=2}^{|\pi(i)|}\mathscr{C}^j_+(i)[\mathrm{y}] &= \sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]}\mathscr{C}^j_+[\mathrm{y}] - \left(\sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+[\mathrm{y}]\right) - \mathscr{C}^1_+(i)[\mathrm{y}] \\
&\overset{\text{from (6.25)}}{\leq} \alpha\times\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}]|}{t} - \left(\sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+(i)[\mathrm{y}]\right) - \mathscr{C}^1_+(i)[\mathrm{y}] \qquad (6.37)
\end{aligned}$$

By using Equation (6.36) and Equation (6.37) together, we get

$$\begin{aligned}
\mathscr{C}^1_+(i)[\mathrm{y}+1] &\leq \mathscr{C}^1_+(i)[\mathrm{y}] + \left(\alpha\times\frac{|\pi(i)|-1}{t}\right) + \left(\alpha\times\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}]|}{t}\right) \\
&\qquad - \left(\sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+[\mathrm{y}]\right) - \mathscr{C}^1_+(i)[\mathrm{y}] \\
&\leq \left(\alpha\times\frac{|\pi(i)|-1}{t}\right) + \left(\alpha\times\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}]|}{t}\right) - \sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+[\mathrm{y}]
\end{aligned}$$

Here, we can re-use Equation (6.30) since all the processor type nodes that are connected to task τ_i, except $\pi^1(i)\in\pi(i)$, are deleted from the graph on line 21 (this case is similar to Case 3.2 in that regard). So, the above equation can be re-written as:

$$\begin{aligned}
\mathscr{C}^1_+(i)[\mathrm{y}+1] &\leq \left(\alpha\times\frac{|\pi(i)|-1}{t}\right) + \left(\alpha\times\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}+1]|-(|\pi(i)|-1)}{t}\right) \\
&\qquad - \sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+[\mathrm{y}] \\
&\leq \left(\alpha\times\frac{|\mathrm{P}^{\text{out}}[\mathrm{y}+1]|}{t}\right) - \sum_{\pi^j\in\mathrm{P}^{\text{in}}[\mathrm{y}]\setminus\pi(i)}\mathscr{C}^j_+[\mathrm{y}] \qquad (6.38)
\end{aligned}$$

Now, let us look at the term $\sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}+1]$:

$$\begin{aligned}\sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}+1] &= \left(\sum_{\substack{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1] \\ \pi^j \neq \pi^1(i)}} \mathscr{C}_+^j[\mathrm{y}+1]\right) + \mathscr{C}_+^1(i)[\mathrm{y}+1] \\ &\overset{\text{from (6.38)}}{\leq} \left(\sum_{\substack{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1] \\ \pi^j \neq \pi^1(i)}} \mathscr{C}_+^j[\mathrm{y}+1]\right) + \left(\alpha \times \frac{|\mathrm{P}^{\mathrm{out}}[\mathrm{y}+1]|}{t}\right) \\ &\quad - \sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}] \setminus \pi(i)} \mathscr{C}_+^j[\mathrm{y}] \end{aligned} \tag{6.39}$$

From the case, we have, $\mathrm{P}^{\mathrm{in}}[\mathrm{y}+1] = \mathrm{P}^{\mathrm{in}}[\mathrm{y}] \setminus \pi(i) \cup \{\pi^1(i)\}$, and thus $\mathrm{P}^{\mathrm{in}}[\mathrm{y}+1] \setminus \{\pi^1(i)\} = \mathrm{P}^{\mathrm{in}}[\mathrm{y}] \setminus \pi(i)$. Hence,

$$\sum_{\substack{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1] \\ \pi^j \neq \pi^1(i)}} \mathscr{C}_+^j[\mathrm{y}+1] = \sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}] \setminus \pi(i)} \mathscr{C}_+^j[\mathrm{y}]$$

Using this on Equation (6.39) leads to:

$$\sum_{\pi^j \in \mathrm{P}^{\mathrm{in}}[\mathrm{y}+1]} \mathscr{C}_+^j[\mathrm{y}+1] \leq \alpha \times \frac{|\mathrm{P}^{\mathrm{out}}[\mathrm{y}+1]|}{t}$$

This concludes Case 3.3.

Hence the proof. □

Corollary 8. *If there exists a feasible intra-migrative assignment of a task set τ on a platform π then* $\mathrm{LPG}_{\mathrm{IM}}$ *succeeds as well, in finding such a feasible intra-migrative assignment of τ but on a platform π' in which only one processor of each type is $1 + \alpha \times \frac{t-1}{t}$ times faster.*

Proof. This follows from Lemma 46. From Lemma 46, we have, if there exists a feasible intra-migrative assignment of τ on π then $\mathrm{LPG}_{\mathrm{IM}}$ succeeds as well, in finding such a feasible intra-migrative assignment of τ but on a platform π'' in which each *fractional processor type* (i.e., processor type in the graph to which a fractional task is assigned after step 3 of $\mathrm{LPG}_{\mathrm{IM}}$) has an additional capacity $\alpha \times \frac{t-1}{t}$ than the corresponding processor type in π. Also, for those processor types that are not in the graph, $\mathrm{LPG}_{\mathrm{IM}}$ does not require any additional capacity on those processor types. However, increasing the capacity of those processor types by the same factor does not affect the performance guarantee (shown in Lemma 46) of $\mathrm{LPG}_{\mathrm{IM}}$. Further, since there was no restriction placed by step 4 of $\mathrm{LPG}_{\mathrm{IM}}$ algorithm on how to distribute this additional required capacity among the processors of each type, adding the entire $\alpha \times \frac{t-1}{t}$ capacity to *only one processor of each type* satisfies Lemma 46. Hence the proof. □

Theorem 21 (**Speed competitive ratio of** $\mathrm{LPG}_{\mathrm{IM}}$). *If there exists a feasible intra-migrative assignment of an implicit-deadline sporadic task τ on a t-type heterogeneous multiprocessor platform π then* $\mathrm{LPG}_{\mathrm{IM}}$ *succeeds as well, in finding such a feasible intra-migrative assignment of*

τ but on a platform, $\pi^{\left(1+\alpha\times\frac{t-1}{t}\right)}$, in which every processor is $1+\alpha\times\frac{t-1}{t}$ times faster than the corresponding processor in π.

Proof. This trivially follows from Corollary 8. □

6.9 Conclusions

In this chapter, we considered the problem of intra-migrative scheduling of implicit-deadline sporadic tasks on t-type heterogeneous multiprocessors. Recall that, this problem can be solved in two steps: first, assign tasks to processor types and then globally schedule the tasks that are assigned to each processor type (since all the processors of a type can be seen an identical multiprocessor platform) using a global scheduling algorithm, such as ERFair [AS00], DP-Fair [LFS$^+$10], U-EDF [NBN$^+$12], that is designed for identical multiprocessors. So, assuming that such an optimal scheduling algorithm is used to schedule the tasks on each processor type, the challenge is to assign all the tasks to the processor types.

We showed that, this problem of intra-migrative task assignment on t-type heterogeneous multiprocessors is NP-Complete in the strong sense. We then proposed an algorithm, LPG_{IM} for this problem that relies on solving a linear programming formulation and that uses graph theory techniques to output the feasible intra-migrative task assignment if there exists one. LPG_{IM} algorithm has a polynomial time-complexity and has a speed competitive ratio of $1+\alpha\times\frac{t-1}{t}$ against an equally powerful intra-migrative adversary. The parameter $0<\alpha\leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than one and the parameter $t\geq 2$ denotes the number of distinct processor types in the platform. For the special case in which $t=2$, i.e., for two-type heterogeneous multiprocessors, the speed competitive ratio becomes $1+\frac{\alpha}{2}\leq 1.5$. Hence, this result can be seen as a generalization of the result obtained for SA algorithm in Chapter 3; however, LPG_{IM} algorithm itself is not a generalization of SA algorithm as it is designed in an entirely different manner.

To the best of our knowledge, for the problem of intra-migrative task assignment on t-type heterogeneous multiprocessors, no previous algorithm exists and hence our algorithm, LPG_{IM}, is the first of its kind. It can be further justified as follows. Although some of the non-migrative algorithms from state-of-the-art (such as the algorithms presented in [HS76, LST90]) can be "adapted" to the intra-migrative model, these "adapted" algorithms will be inefficient compared to the LPG_{IM} algorithm, either in terms of the speed competitive ratio or in terms of the time-complexity. For example, the adapted version of the algorithm in [LST90] will have inferior speed competitive ratio and the adapted version of the algorithm in [HS76] will continue to have a significantly higher time-complexity (which will severely limit the practicality of this algorithm).

Chapter 7

Non-migrative Scheduling on T-type Heterogeneous Multiprocessors

7.1 Introduction

In this chapter, we consider the problem of non-migrative scheduling of tasks on t-type (where $t \geq 2$) heterogeneous multiprocessors. Recall that, we studied the non-migrative task assignment problem earlier in Chapter 4 but for two-type heterogeneous multiprocessors. Hence, several algorithms discussed in that chapter (i.e., FF-3C, SA-P, LPC and PTAS_{NF}) are only applicable to two-type heterogeneous multiprocessors and unfortunately cannot be generalized for t-type ($t \geq 2$) heterogeneous multiprocessors. Hence, in this chapter, we aim to design a non-migrative task assignment algorithm for t-type heterogeneous multiprocessors.

Recall that, in the *non-migrative* model, every task is statically assigned to a processor before run-time and all its jobs must execute only on that processor at run-time. The challenge is to find, before run-time, a *task-to-processor* assignment such that, at run-time, on each processor, the given scheduling algorithm meets all deadlines of the tasks assigned on that processor. Scheduling tasks to meet deadlines is a well-understood problem in the non-migrative model. One may use Earliest Deadline First (EDF) [LL73] on each processor, for example. EDF is an *optimal* scheduling algorithm on a uniprocessor system [LL73, Der74], with the interpretation that, for every valid arrival pattern, if a schedule exists that meets all deadlines then EDF succeeds as well to construct such a schedule in which all deadlines are met. Therefore, assuming that an optimal scheduling algorithm is used on every processor to schedule the tasks, the challenging part is to find a task-to-processor assignment such that, *there exists* a schedule that meets all deadlines — such an assignment is said to be *feasible* assignment hereafter. It can be shown that the problem of non-migrative task assignment on t-type heterogeneous multiprocessors is NP-Complete in the strong sense (by reducing an instance of the 3-PARTITION problem to an instance of our problem). Therefore, for this problem, we propose a *polynomial* time-complexity algorithm, LPG_{NM}, with a *finite* speed competitive ratio. This algorithm is an extended version of LPG_{IM} algorithm discussed in Chapter 6.

Computing Platform	Adversary	Task Assignment Algorithms			
	Task migration	Algorithm	Task migration	Speed competitive ratio	Complexity
t-type[a]	non-migrative	[Bar04b]	non-migrative	2	$O(P)$[c]
t-type	non-migrative	[Bar04c]	non-migrative	2	$O(P)$
t-type	non-migrative	[LST90]	non-migrative	2	$O(P)$
t-type	fully-migrative	[CSV12]	non-migrative	4	$O(P)$
t-type	non-migrative	[HS76]	non-migrative	PTAS[d]	exponential in procs
t-type	non-migrative	[JP99]	non-migrative	PTAS	exponential in procs and $O(P)$
t-type	non-migrative	[WBB13]	non-migrative	PTAS	exponential in $1/\varepsilon$ and $O(P)$
2-type[b]	non-migrative	FF-3C (Chap. 4, Sec. 4.3)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	intra-migrative	SA (Chapter 3)	intra-migrative	$1+\frac{\alpha}{2}$[e] ≤ 1.5	low-degree polynomial
2-type	intra-migrative	SA-P (Chap. 4, Sec. 4.4)	non-migrative	$1+\alpha \leq 2$	low-degree polynomial
2-type	non-migrative	LPC (Chap. 4, Sec. 4.5)	non-migrative	1.5 (and 3 extra processors)	$O(P)$
2-type	non-migrative	PTAS_{NF} (Chap. 4, Sec. 4.6)	non-migrative	PTAS	exponential in $1/\varepsilon$
t-type	intra-migrative	LPG_{IM} (Chapter 6)	intra-migrative	$1+\alpha \times \frac{t-1}{t}$	$O(P)$
t-type	intra-migrative	LPG_{NM}	non-migrative	$1+\alpha \leq 2$	$O(P)$

[a] A heterogeneous multiprocessor platform having two or more processor types.
[b] A heterogeneous multiprocessor platform having only two processor types.
[c] The time-complexity O(P) indicates that the algorithm relies on solving a Linear Program (LP) formulation — note that though a linear program can be solved in polynomial time, the polynomial generally has a higher degree.
[d] A PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal.
[e] The parameter $0 < \alpha \leq 1$ is a property of the task set — it is the maximum of all the task utilizations that are no greater than one.

Table 7.1: Summary of state-of-the-art task assignment algorithms along with the LPG_{NM} algorithm proposed in this chapter.

Problem Statement. In this chapter, we consider the problem of non-migrative scheduling of implicit-deadline sporadic tasks on t-type heterogeneous multiprocessors. That is, assuming that an optimal uniprocessor scheduling algorithm (such as EDF) is used on every processor of each type to schedule the tasks, we design a task assignment algorithm for determining a feasible task-to-processor assignment.

Hardness of the Problem. It is trivial to see that the problem of non-migrative task assignment on t-type heterogeneous multiprocessors is NP-Complete in the strong sense. This is because, even in the simpler case of two-type heterogeneous multiprocessors, the problem of non-migrative task assignment is NP-Complete in the strong sense — this was shown in Section 4.2 in Chapter 4 on page 76 (by reducing an instance of the 3-PARTITION problem, which is known to be NP-Complete in the strong sense [Joh73], to an instance of our problem). Hence, this result continues to hold for t-type ($t \geq 2$) heterogeneous multiprocessors as well.

Related work. The non-migrative task assignment problem on heterogeneous multiprocessors has been studied in the past [Bar04c, Bar04b, RABN12, RAB13, RN12b, WBB13, HS76, LST90, JP99, CSV12][1]. In [Bar04c, Bar04b, LST90], a couple of non-migrative algorithms are proposed

[1] It is a well-known fact that the non-migrative task assignment problem is equivalent to the problem of scheduling

each with a speed competitive ratio of 2 against an equally powerful non-migrative adversary. The approach discussed in [LST90] comes closest to our work since it formulates the task assignment problem as a Mixed Integer Linear Program (MILP) and then relaxes it to a Linear Program (LP) and finally uses a rounding technique to obtain the non-migrative task assignment. We also follow the same approach in this work; however, by formulating MILP in a different way and using different techniques while rounding, we obtain a *better* speed competitive ratio for our non-migrative task assignment algorithm than the one in [LST90].

The non-migrative algorithms discussed earlier in Chapter 4, such as, FF-3C, SA-P, LPC and PTAS_{NF}, are applicable only on two-type heterogeneous multiprocessors (a special case of t-type in which $t = 2$) and unfortunately cannot be extended for generic t-type ($t \geq 2$) heterogeneous multiprocessors.

Moving to algorithms whose speed competitive ratios have been proven against a more powerful adversary, recently, in [CSV12], authors propose a non-migrative algorithm with a speed competitive ratio of 4 against the fully-migrative adversary. Further, it is also shown that, this bound is *exact*, i.e., it is *impossible* to design a non-migrative algorithm with a speed competitive ratio smaller than 4 against the fully-migrative adversary [CSV12].

In [HS76, JP99, RN12b, WBB13], *polynomial-time approximation scheme*s (PTASs) have been proposed for the problem of non-migrative task assignment. Recall that, a PTAS takes an instance of an optimization problem and a parameter $\varepsilon > 0$ as inputs and, in time polynomial in the problem size (although not necessarily in the value of ε), produces a solution that is within a factor $1+\varepsilon$ of being optimal. PTAS is theoretically a significant result since such algorithms partition the task set in polynomial time, to any desired degree of accuracy. However, most often, their practical significance is severely limited due to a very high run-time complexity that they incur.

The state-of-the-art along with the contributions of this chapter are summarized in Table 7.1.

Contributions and Significance of the work discussed in this chapter. Consider a t-type heterogeneous multiprocessor platform π and an implicit-deadline sporadic task set τ in which, it holds that: $\forall k \in \{1,2,\ldots,t\}$, for every task in τ, utilization of each task on a type-k processor is either no greater than α or is equal to ∞, where $0 < \alpha \leq 1$. We present a non-migrative algorithm, LPG_{NM}, which offers the following guarantee. If there exists a feasible *intra-migrative* assignment of the task set τ on the platform π then LPG_{NM} succeeds as well, in finding a feasible *non-migrative* assignment of τ but on a platform $\pi^{(1+\alpha)}$, in which *every processor* is $1+\alpha$ times faster than the corresponding processor in π.

We believe that the significance of this work is as follows. For the problem of non-migrative task assignment on t-type heterogeneous multiprocessors, our algorithm, LPG_{NM}, has a superior performance compared to state-of-the-art. This can be seen from Table 7.1 since (i) LPG_{NM} has a better speed competitive ratio compared to algorithms in [Bar04b, Bar04c, LST90]. This is because its speed competitive ratio is $1+\alpha \leq 2$ against a *more powerful* intra-migrative adversary

a set of non-real-time jobs, arriving at time zero, on unrelated parallel machine, so that they all finish before a specified time. This equivalent problem has been studied in [HS76, LST90, JP99, CSV12].

and as can be seen, it is quantified using the *parameter*, $0 < \alpha \leq 1$, which is a characteristic of the task set. However, the speed competitive ratio of all the algorithms in [Bar04b, Bar04c, LST90] is 2 against *equally powerful* non-migrative adversary and as can bee seen, it is a *constant* (the speed competitive ratio of LPG_{NM} reaches this constant 2 only when $\alpha = 1$ and for all other values of α, it is smaller than 2), (ii) among algorithms with speed competitive ratio proven against an adversary with a migration model of intra-migrative or greater power [CSV12], LPG_{NM} offers the best speed competitive ratio and (iii) compared to PTAS algorithms [HS76, JP99, WBB13] whose practical significance is severely limited as they incur a very high time-complexity (exponential in processors or exponential in $1/\varepsilon$), our algorithm offers a significantly lower time-complexity.

A global view. The context of the new algorithm, LPG_{NM}, can be visualized as shown in Figure 7.1.

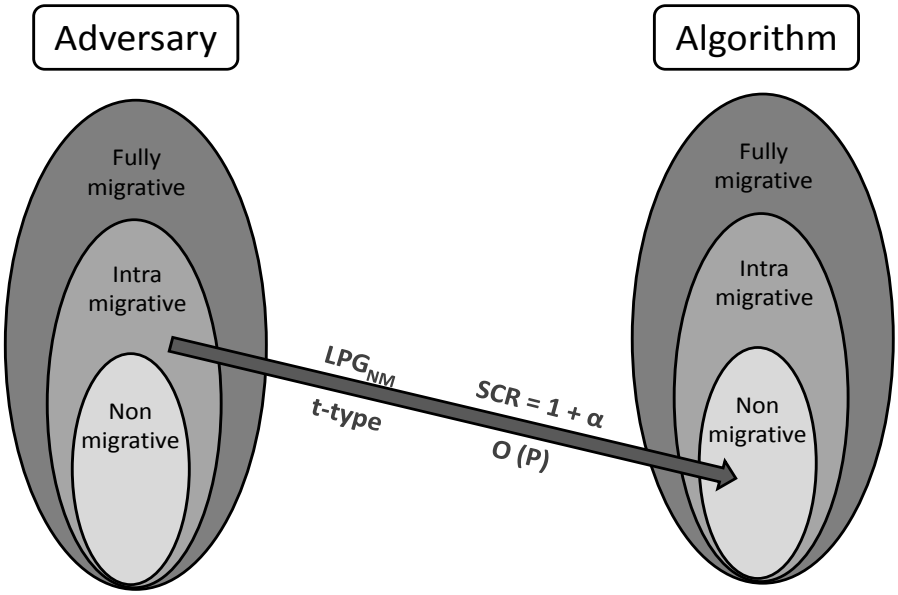


Figure 7.1: A global view of the new algorithm, LPG_{NM}, proposed in this chapter. Here, SCR denotes the "speed competitive ratio", α is a property of the task set — it is the maximum of all the task utilizations that are no greater than one (and hence can take a value in the range $(0,1]$) and $O(P)$ indicates that the algorithm relies on solving a Linear Program formulation.

Organization of the chapter. The rest of the chapter is organized as follows. Section 7.2 briefs the system model. Section 7.3 presents our new non-migrative algorithm, LPG_{NM}, and proves its speed competitive ratio. Finally, Section 7.4 concludes.

7.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n independent implicit-deadline sporadic tasks on a t-type heterogeneous multiprocessor platform π comprising m processors. In platform π, the set of m_k processors of type-k is denoted by $\pi^k = \{p_1, p_2 \ldots, p_{m_k}\}$, where

$1 \leq k \leq t$ and p_j denotes a processor of type-k, where $1 \leq j \leq m_k$. It then holds that: $\bigcup_{k=1}^{t} \pi^k = \pi$ and $\bigcap_{k=1}^{t} \pi^k = \emptyset$ and finally $\sum_{k=1}^{t} m_k = m$.

The minimum inter-arrival time of a task τ_i is denoted by T_i. On a t-type platform, the WCET of every task depends on the type of the processor on which the task executes. We denote by C_i^k the WCET of task τ_i when executed on a type-k processor, where $k \in \{1,2,\ldots,t\}$. We denote by $u_i^k \stackrel{\text{def}}{=} C_i^k/T_i$ the utilization of task τ_i on a type-k processor and u_i^k is a real number in $[0,1] \cup \{\infty\}$ — if τ_i cannot be executed on a type-k processor then u_i^k is set to ∞. Let α be a real number defined as follows:

$$\alpha \stackrel{\text{def}}{=} \max_{\tau_i \in \tau, k \in \{1,2,\ldots,t\}} \left\{ u_i^k : u_i^k \leq 1 \right\}$$

Then it holds that the utilization of any task on any processor type is either no greater than α or is equal to ∞, i.e.,

$$\forall k \in \{1,2,\ldots,t\}, \forall \tau_i \in \tau: \ \ (u_i^k \leq \alpha) \ \vee \ (u_i^k = \infty) \tag{7.1}$$

7.3 LPG_{NM}: The non-migrative task assignment algorithm

We now present a *non-migrative* task assignment algorithm, LPG_{NM}, for assigning tasks to *individual* processors on a t-type platform. This algorithm is an enhanced version of the intra-migrative algorithm, LPG_{IM}, discussed in Chapter 6. We also prove the speed competitive ratio of LPG_{NM}, against a more powerful intra-migrative adversary.

7.3.1 The description of LPG_{NM} algorithm

The non-migrative algorithm, LPG_{NM}, works as follows.

Step 1. Assign tasks in the given task set τ to processor types in platform π' using LPG_{IM} algorithm (described in the previous chapter); in platform π', *only one processor of each type* is $1 + \alpha \times \frac{t-1}{t}$ times faster compared to π. Recall that LPG_{IM} assigns tasks to processor types and not to processors.

Step 2. Assign the tasks, that are assigned to processor type-k (i.e., to processor type), to individual processors of type-k ($\forall k \in \{1,2,\ldots,t\}$), using next-fit but allowing *splitting* of tasks between consecutive processors. Such an assignment ensures that [LFS+10]: at most $m_k - 1$ tasks are *split* between processors of type-k with at most one task split between each pair of consecutive processors.

Step 3. Copy the assignment obtained in Step 2 onto a faster platform, $\pi^{(1+\alpha)}$, in which every processor is $1+\alpha$ times faster than the corresponding processor in π.

Step 4. On platform $\pi^{(1+\alpha)}$, $\forall k \in \{1,2,\ldots,t\}$, assign a task split between consecutive processors, say p and $p+1$, of type-k, to processor p, where $p_1 \leq p < p_{m_k}$.

With this description of LPG_{NM} algorithm, we now derive its speed competitive ratio.

7.3.2 The speed competitive ratio of LPG_{NM} algorithm

In this section, we prove the speed competitive ratio of the non-migrative task assignment algorithm, LPG_{NM}, against a more powerful intra-migrative adversary.

Theorem 22 (**Speed competitive ratio of** LPG_{NM}**.**). *If there exists a feasible intra-migrative assignment of an implicit-deadline sporadic task set τ on a t-type heterogeneous multiprocessor platform π then* LPG_{NM} *succeeds as well, in finding a feasible non-migrative assignment of τ but on a platform $\pi^{(1+\alpha)}$ in which every processor is $1+\alpha$ times faster than the corresponding processor in π.*

Proof. Recall from Corollary 8 in Chapter 6 (page 234) that, if a task set τ is intra-migrative feasible on a platform π then the intra-migrative algorithm, LPG_{IM}, succeeds to output such a feasible intra-migrative assignment of τ but on a platform π', in which *only one processor of each type* is $1+\alpha\times\frac{t-1}{t}$ times faster and the remaining processors are of the same speed as the corresponding processors in π. Let p_{m_k} denote the processor of type-k ($\forall k \in \{1,2,\ldots,t\}$) whose speed is $1+\alpha\times\frac{t-1}{t}$ times faster. So, in platform π', before assigning any tasks, it holds by definition that, $\forall k \in \{1,2,\ldots,t\}$ of π':

$$\forall p \in \text{ type-k } \wedge p \neq p_{m_k} : \mathscr{FC}[p] = 1 \qquad \text{and} \tag{7.2}$$

$$p \in \text{ type-k } \wedge p = p_{m_k} : \mathscr{FC}[p] = 1 + \alpha \times \frac{t-1}{t} \tag{7.3}$$

where $\mathscr{FC}[p]$ denotes the current *free*/available capacity on processor p. Since τ is intra-migrative feasible on π, after Step 1 of LPG_{NM}, it holds (by Corollary 8) that, $\forall k \in \{1,2,\ldots,t\}$ of π':

$$\sum_{\tau_i \in \tau^k} u_i^k \;\leq\; m_k + \left(\alpha \times \frac{t-1}{t}\right) \tag{7.4}$$

where τ^k denotes the set of tasks assigned to type-k processors (i.e., to processor types and not to individual processors). We also know from Equation (7.1) and Equation (6.3) that:

$$\forall k \in \{1,2,\ldots,t\} : \tau_i \in \tau^k : u_i^k \leq \alpha \tag{7.5}$$

In Step 2, LPG_{NM} assigns tasks to individual processors using "wrap-around" technique, which allows splitting of tasks between processors of same type. Combining such an assignment with Equations (7.2)–(7.4), it holds that,

$\forall k \in \{1,2,\ldots,t\}$ of π':

$$\forall p \in \text{ type-k } \wedge p \neq p_{m_k} : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^k \leq 1 \tag{7.6}$$

$$p \in \text{ type-k } \wedge p = p_{m_k} : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^k \;\leq\; 1 + \left(\alpha \times \frac{t-1}{t}\right) \tag{7.7}$$

$$\forall p \in \text{ type-k } : \mathscr{FC}[p] \geq 0 \qquad \text{and} \tag{7.8}$$

at most $m_k - 1$ tasks are fractionally assigned between type-k processors with each task split between consecutive processors (7.9)

where $\tau[p]$ and $\mathscr{UC}[p]$ denote the set of tasks assigned on processor p and the capacity currently *used* on processor p, respectively.

On step 3, LPG$_{\text{NM}}$ copies this assignment onto the faster platform, $\pi^{(1+\alpha)}$. In platform $\pi^{(1+\alpha)}$, before assigning any tasks, it holds by definition that, $\forall k \in \{1,2,\ldots,t\}$ of $\pi^{(1+\alpha)}$:

$$\forall p \in \text{ type-k } : \mathscr{FC}[p] = 1 + \alpha \tag{7.10}$$

From Equation (7.6)–(7.10) and since the assignment is "copied" on $\pi^{(1+\alpha)}$, we have, $\forall k \in \{1,2,\ldots,t\}$ of $\pi^{(1+\alpha)}$:

$$\forall p \in \text{ type-k } \wedge p \neq p_{m_k} : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^k \leq 1 \tag{7.11}$$

$$p \in \text{ type-k } \wedge p = p_{m_k} : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^k \;\leq\; 1 + \left(\alpha \times \frac{t-1}{t}\right) \tag{7.12}$$

$$\forall p \in \text{ type-k } \wedge p \neq p_{m_k} : \mathscr{FC}[p] \geq \alpha \tag{7.13}$$

$$p \in \text{ type-k } \wedge p = p_{m_k} : \mathscr{FC}[p] \geq \alpha/t \qquad \text{and} \tag{7.14}$$

at most $m_k - 1$ tasks are fractionally assigned between type-k processors with each task split between consecutive processors (7.15)

From Equation (7.13), Equation (7.15) and Equation (7.5), it can be seen that, each of the at most $m_k - 1$ fractional tasks can be integrally assigned to each of the $m_k - 1$ processors of type-k (i.e., $\forall p \in \text{ type-k } \wedge p \neq p_{m_k}$) in platform, $\pi^{(1+\alpha)}$, in their respective free capacities. Combining this with Equation (7.12) yields: $\forall k \in \{1,2,\ldots,t\}$ of $\pi^{(1+\alpha)}$:

$$\forall p \in \text{ type-k } : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^k \;\leq\; 1 + \alpha \tag{7.16}$$

Observe that u_i^k is the utilization of a task, τ_i, on a processor of type-k on platform π. Let $u_i^{k'}$ denote the utilization of task τ_i on a processor of type-k on platform $\pi^{(1+\alpha)}$. Then it holds (by definition of these platforms) that: $\forall \tau_i \in \tau : \frac{u_i^{k'}}{u_i^k} = \frac{1}{1+\alpha}$. Applying this on Equation (7.16) yields:

$\forall k \in \{1,2,\ldots,t\}$ of $\pi^{(1+\alpha)}$:

$$\forall p \in \text{ type-k } : \mathscr{UC}[p] = \sum_{\tau_i \in \tau[p]} u_i^{k'} \;\leq\; 1 \tag{7.17}$$

Since Equation (7.17) is a necessary and sufficient feasibility condition for task assignment on a uniprocessor [LL73], the non-migrative assignment of τ on $\pi^{(1+\alpha)}$ returned by LPG_{NM} is feasible. Hence the proof. □

7.4 Conclusions

In this chapter, we considered the problem of non-migrative scheduling of implicit-deadline sporadic tasks on t-type heterogeneous multiprocessors. Recall that, this problem can be solved in two steps: first, assign tasks to individual processors and then schedule the tasks that are assigned to each processor using an optimal uniprocessor scheduling algorithm, such as EDF. So, assuming that such an optimal scheduling algorithm is used to schedule the tasks on each processor, the challenge is to assign tasks to individual processors.

This problem is known to be NP-Complete in the strong sense. Hence, for this problem, we proposed an algorithm, LPG_{NM}, with a finite speed competitive ratio. This algorithm is an extension of the (intra-migrative) algorithm, LPG_{IM} (which is discussed in Chapter 6), and hence also relies on solving linear programming formulation and uses graph theory techniques to output the task assignment. LPG_{NM} has polynomial time-complexity and has a speed competitive ratio of $1+\alpha$ against a more powerful intra-migrative adversary, where the parameter $0 < \alpha \leq 1$ is a property of the task set; it is the maximum of all the task utilizations that are no greater than one.

For the problem of non-migrative task assignment on t-type platforms, our algorithm, LPG_{NM}, has a superior performance compared to state-of-the-art since (i) LPG_{NM} has a *tighter bound* compared to algorithms in [Bar04b, Bar04c, LST90], i.e., its speed competitive ratio is $1+\alpha$ (a *parametrized* value) against a *more powerful* intra-migrative adversary whereas the speed competitive ratio of all the algorithms in [Bar04b, Bar04c, LST90] is 2 (a *constant*) but against an *equally powerful* non-migrative adversary, (ii) among algorithms with speed competitive ratio proven against an adversary with a migration model of intra-migrative or greater power [CSV12], LPG_{NM} offers the best speed competitive ratio and (iii) compared to PTAS algorithms [HS76, JP99, WBB13] whose practical significance is severely limited as they incur a very high time-complexity (exponential in processors or exponential in $1/\varepsilon$), our algorithm offers a lower (i.e., polynomial) time-complexity.

Chapter 8

Shared Resource Scheduling on T-type Heterogeneous Multiprocessors

8.1 Introduction

In many computing systems, apart from sharing processors, tasks also share other resources such as data structures, sensors, etc. and tasks must operate on such resources in a *mutually exclusive* manner. Recall from Chapter 5 that, even on a single processor, the sharing of such resources can have a profound effect on timing behavior as witnessed by the near failure of the NASA mission, *Mars Pathfinder*, because the resource-sharing protocol in the operating system was not enabled [Jon97]. Scheduling real-time tasks that share resources on a *heterogeneous multiprocessor* platform is even more complex. Therefore, in this chapter, we aim to address this problem (partially) by designing an algorithm with a *finite* speed competitive ratio.

Problem Statement. We consider the problem of scheduling a task set τ of implicit-deadline sporadic tasks to meet all deadlines on a t-type heterogeneous multiprocessor platform where a task may access multiple shared resources. There are m_k processors of type-k, where $k \in \{1, 2, \ldots, t\}$. The execution time of a task depends on the processor type on which it executes. There is a set R of resources. For each task τ_i, there is a resource set $R_i \subseteq R$ such that, for each job of τ_i, during one phase of its execution, the job requests to hold the resource set R_i exclusively with the interpretation that (i) the job makes a single request to hold all the resources in the resource set R_i and (ii) at all times, when a job of τ_i holds the resource set R_i, no other job holds any resource in R_i. We assume that each job of task τ_i may request the resource set R_i at most once during its execution. We also assume (like the previous work on D-PCP [RSL88]) that a job is allowed to migrate when it requests a resource set and when it releases a resource set but a job is not allowed to migrate at other times.

Hardness of the Problem. The problem under consideration can be shown to be NP-Complete in the strong sense by reducing an instance of the 3-PARTITION problem to an instance of our problem. Intuitively, it can be reasoned as follows: (i) it is trivial to see that the resource sharing problem (on two-type heterogeneous multiprocessors) that we studied earlier in Chapter 5 is a

restricted version of the problem under consideration and (ii) it was shown in Chapter 5 (see Section 5.3 on page 190) that this restricted version of the problem is NP-Complete in the strong sense. From (i) and (ii), it can be concluded that the problem under consideration in this chapter is NP-Complete in the strong sense as well. Hence, our goal is to design a polynomial time-complexity algorithm for this problem and prove its speed competitive ratio.

Related Work. Scheduling a collection of jobs that share resources is well-studied in operations research (see [BLK83], for example) but unfortunately these algorithms deal with jobs which make them less suited for real-time systems because real-time systems tend to be implemented with tasks that generate a (potentially infinite) sequence of jobs. The problem of scheduling a set of implicit-deadline sporadic tasks on heterogeneous multiprocessors has been studied in the past [Bar04a, Bar04b, Bar04c, CSV12, LST90, ARB10, RAB13, RABN12, RN12a, WBB13, HS76, JP99] but without considering the case when tasks share resources. The resource sharing algorithm, FF-3C-vpr, discussed in Chapter 5 is only applicable for two-type heterogeneous multiprocessors and unfortunately, cannot be extended for the generic t-type heterogeneous multiprocessors. Recently, a run-time synchronization protocol, PSRP, is proposed in [HBL12] for the problem of scheduling parallel tasks on a platform comprising multiple heterogeneous resources. It considers a parallel task model in which a task may execute on several processors at the same time whereas we consider a sequential task model in which a task can execute on at most one processor at any time. In this respect, the task model considered in [HBL12] is more generic than the one considered in this chapter. However, the PSRP algorithm of [HBL12] does not have a proven speed competitive ratio whereas we prove the speed competitive ratio for our algorithm. More importantly, the work in [HBL12] proposes a "run-time synchronization mechanism" and thus assumes that an assignment of tasks to processors is given; however, in this work, we propose an algorithm which assigns tasks to processors before run-time and handles synchronization at run-time. So, the problem addressed and the goals of [HBL12] are different than this work although both are related to sharing multiple resources on multiprocessors.

For the problem of scheduling tasks that share resources on heterogeneous multiprocessors, one might also consider an obvious solution of assigning tasks to processors and then applying a resource-sharing protocol conceived for identical multiprocessors, for example, D-PCP [RSL88]. However, protocols for resource sharing on an identical multiprocessor (such as D-PCP) are less effective in minimizing priority inversion when used in heterogeneous multiprocessors as they are in minimizing priority inversion when used in identical multiprocessors. The reason for this is that, a task holding a shared resource may be executing on a processor where it runs slowly — causing large priority inversion to other tasks and poor schedulability. Therefore, a resource-sharing protocol for heterogeneous platforms ought to be cognizant of the execution rate of each task on each processor type. It should also provide a bound on how much worse it performs, compared to an optimal scheme.

This work. In this chapter, we propose an algorithm, LP-EE-vpr, for the problem of scheduling implicit-deadline sporadic tasks that share resources on a t-type heterogeneous multiprocessor platform, which is formally defined earlier in the section. We also prove the speed competitive

ratio of LP-EE-vpr algorithm.

A key idea of our new algorithm is to organize the resource sets into *resource request partitions* so that for every pair of tasks, τ_i and $\tau_{i'}$, if there is a resource shared between these two tasks (that is, if $R_i \bigcap R_{i'} \neq \emptyset$) then the resource sets (R_i and $R_{i'}$) belong to the same resource request partition. Hence, if two resource sets of different tasks belong to different resource set partitions then we know that, these tasks do not share resources. We will create a procedure for forming the resource request partitions and then we let P denote the set of resource request partitions and MAXP denote the number of elements in the resource request partition with the largest number of elements. (P and MAXP will be defined formally in Section 8.2.)

The algorithm, LP-EE-vpr, offers the guarantee that *if* a task set is schedulable on a t-type heterogeneous multiprocessor platform to meet all deadlines by an optimal scheduling algorithm that allows a job to migrate only when it requests or releases the resources, *then* our algorithm succeeds to meet all deadlines as well with the same restriction on the job migration but given processors $4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{\min\{m_1, m_2, \ldots, m_t\}} \right\rceil\right)$ times faster. In order to prove this bound, we create a new algorithm, ra-np-pEDF-fav, which is used by LP-EE-vpr and prove a lemma which compares the *feasibility* of tasks on a multiprocessor with the *schedulability* of tasks scheduled by ra-np-pEDF-fav and as a corollary of this lemma, we obtain a new, tighter, speed competitive ratio of uniprocessor non-preemptive EDF scheduling — we improve the (previously known [AE10]) bound from *three* to *two*. This is an interesting result in its own right. For the special case in which each task requests at most one resource, the bound of LP-EE-vpr collapses to $4 \times \left(1 + \left\lceil \frac{|R|}{\min\{m_1, m_2, \ldots, m_t\}} \right\rceil\right)$.

Contributions and Significance of the work discussed in this chapter. This chapter makes two contributions. First, for the problem of scheduling implicit-deadline sporadic tasks that share multiple resources on t-type heterogeneous multiprocessors, no previous algorithm exists and hence our algorithm, LP-EE-vpr, is the first for this problem with a proven speed competitive ratio. Second, for the problem of non-preemptive scheduling of tasks on a uniprocessor, this work improves the previously known [AE10] speed competitive ratio of uniprocessor non-preemptive EDF algorithm from *three* to *two*. This improvement is presented because it is a natural by-product of our proof of the speed competitive ratio of LP-EE-vpr.

Organization of the chapter. The rest of the chapter is organized as follows. Section 8.2 briefs the system model. Section 8.3 gives an overview of our algorithm and Section 8.4 describes the algorithm in detail. Section 8.5 proves the speed competitive ratio of ra-np-pEDF-fav (an intermediate result) as well as the speed competitive ratio of LP-EE-vpr (the main result). Section 8.6 discusses useful properties of the proposed algorithm and finally, Section 8.7 concludes.

8.2 System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n implicit-deadline sporadic tasks that share a set $R = \{r_1, r_2, \ldots, r_\rho\}$ of ρ resources on a t-type heterogeneous multiprocessor platform $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprising m processors, of which m_k processors are of

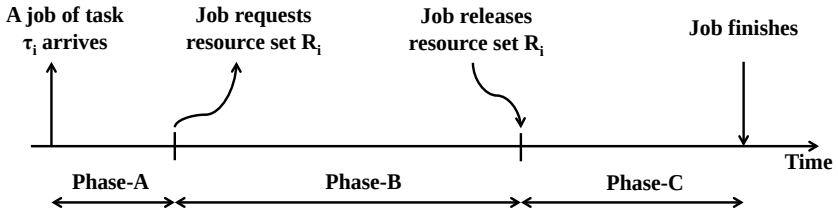


Figure 8.1: Categorization of the execution of a task that requests a resource set into three phases.

type-k, where $k \in \{1,2,\ldots,t\}$.

In the task set, each implicit-deadline sporadic task τ_i generates a (potentially infinite) sequence of *jobs*, with the first job arriving at any time and subsequent jobs arriving *at least* T_i time units apart (referred to as the *minimum inter-arrival time*). Each job of a task τ_i has to complete its execution within $D_i = T_i$ time units from its arrival (referred to as the *deadline*).

In the computing platform, a processor $\pi_p \in \pi$, belongs to one of the t different types of processors. The computing platform consists of m_k processors of type-k, where $k \in \{1,2,\ldots,t\}$, i.e., it consists of m_1 processors of type-1, m_2 processors of type-2, ..., m_t processors of type-t; hence, $m_1 + m_2 + \cdots + m_t = m$.

The tasks share resources from the set $R = \{r_1, r_2, \ldots, r_\rho\}$ of ρ resources. Specifically, for each task, $\tau_i \in \tau$, there is a resource set $R_i \subseteq R$ such that, for each job of τ_i, during one phase of its execution, the job requests to hold the resource set R_i exclusively, that is, at all times, when a job of τ_i holds the resource set R_i, no other job holds any resource in R_i. We assume that each job of task τ_i may request the corresponding resource set R_i at most once during its execution and further each job must request all the resources in this set together. We also assume that a job of a task can execute on at most one processor at any given time; in other words, it cannot execute simultaneously on more than one processor.

For a *job of a task* τ_i *such that* $R_i \neq \emptyset$, we categorize the execution into three phases as follows. Let phase-A execution of a job of task τ_i denote the execution the job performs from when it arrives until it requests the resource set R_i. Let phase-B execution of a job of task τ_i denote the execution the job performs from when it requests the resource set R_i until it releases R_i. Let phase-C execution of a job of task τ_i denote the execution the job performs from when it releases the resource set R_i until it finishes execution. This is illustrated in Figure 8.1. For a *job of a task* τ_i *such that* $R_i = \emptyset$, we categorize its execution into a single phase, phase-A, which denotes the entire execution of the job, i.e., the execution the job performs from when it arrives until it finishes execution.

In our model, we allow a job of task τ_i to migrate at the time when it requests the resource set R_i and when it releases the resource set R_i but the job is not allowed to migrate at other times. (This assumption is similar to previous work on D-PCP [RSL88].) We assume that the processors a job migrates to/from is determined by the task that generated the job and consequently, all jobs of the same task migrate between the same processors. Specifically, phase-A executions of all jobs

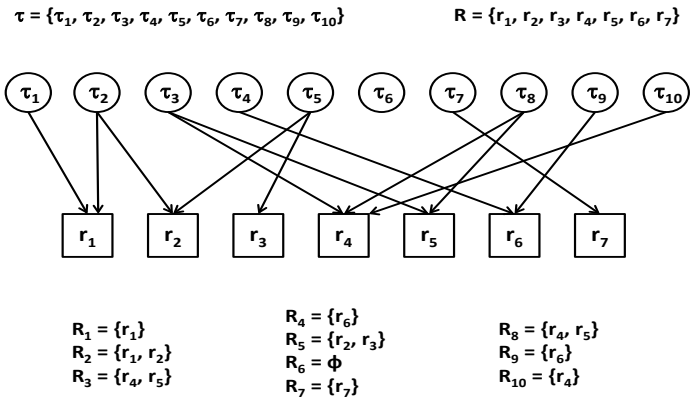
of task τ_i are assigned to the same processor (let $p_{i,a}$ denote this processor). Analogously, phase-B executions of all jobs of task τ_i are assigned to the same processor (let $p_{i,b}$ denote this processor). Phase-C executions of all jobs of task τ_i are assigned to the same processor (let $p_{i,c}$ denote this processor). Thus, all jobs of task τ_i only migrate between these (at most *three*[1]) processors. Note that for a given task τ_i, it can happen that the processors $p_{i,a}$, $p_{i,b}$ and $p_{i,c}$ are of different types. We refer to such assumption of migration as *restricted migration*.

Since a job executing within a phase cannot migrate, we can speak about the execution time of a job in a phase for a given processor type. Let CA_i^k denote an upper bound on the execution time of phase-A of a job of task τ_i if this phase-A execution is assigned to a processor of type-k. Analogously, let CB_i^k denote an upper bound on the execution time of phase-B of a job of task τ_i if this phase-B execution is assigned to a processor of type-k. Let CC_i^k denote an upper bound on the execution time of phase-C of a job of task τ_i if this phase-C execution is assigned to a processor of type-k. For convenience, we introduce the symbol C_i^k as follows: For a task τ_i whose jobs access a resource set, $C_i^k \stackrel{\text{def}}{=} CA_i^k + CB_i^k + CC_i^k$. For a task τ_i whose jobs do not access a resource set, $C_i^k \stackrel{\text{def}}{=} CA_i^k$. Intuitively, C_i^k denotes an upper bound on the execution time of a job of task τ_i *if* all its phases would be assigned to a processor of type-k. For convenience, we also use the following notation. The *utilization* of a task τ_i on a type-k processor (assuming that all phases of the task are assigned to processors of type-k) is denoted by u_i^k and is defined as $u_i^k \stackrel{def}{=} \frac{C_i^k}{T_i}$.

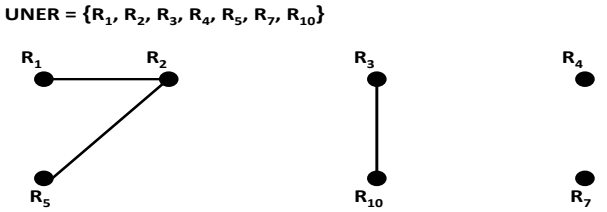
As mentioned earlier, in this work, we consider *implicit-deadline* sporadic tasks, that is, for each task $\tau_i : D_i = T_i$. In some parts of our discussion, however, we discuss *constrained-deadline* sporadic tasks, that is, for each task $\tau_i : D_i \leq T_i$. For a constrained-deadline sporadic task τ_i, its *density* on a type-k processor is denoted as δ_i^k and is defined as $\delta_i^k \stackrel{def}{=} \frac{C_i^k}{\min(D_i,T_i)} = \frac{C_i^k}{D_i}$.

Recall that, tasks request resources from set R of resources. This is illustrated in Figure 8.2a. It is helpful to introduce auxiliary variables and form a graph describing the potential conflicts of resource requests. Let UNER denote the set of unique non-empty resource sets that tasks request. Formally, UNER is defined as, UNER $\stackrel{\text{def}}{=} \bigcup_{\tau_i \in \tau \wedge R_i \neq \emptyset} \{R_i\}$. The graph, ($V$,$E$), with the set V of vertices and the set E of edges is then formed as follows: (i) there is a function fun that maps an element in UNER to an element in V, and this is a one-to-one correspondence, and (ii) there is an edge between vertex, V_{k1}, and vertex, V_{k2}, if and only if, $(\text{fun}^{-1}(V_{k1})) \bigcap (\text{fun}^{-1}(V_{k2})) \neq \emptyset$. Such a graph is shown in Figure 8.2b. Let $PV = \left\{PV_1, PV_2, \ldots, PV_{|PV|}\right\}$ denote the set of $|PV|$ connected components of this graph. The connected components in a graph can be found in linear time using a standard technique [HT73]. For a connected component and the set of connected components, we introduce symbols that describe potential conflicts between resource sets. Let P_j denote the set of unique non-empty resource sets that correspond to the vertices in PV_j. We refer to P_j as a *resource request partition*. Formally, $P_j \stackrel{\text{def}}{=} \left\{\text{UNER}_k : (\text{UNER}_k \in \text{UNER}) \wedge (\text{fun}(\text{UNER}_k) \in PV_j)\right\}$. Let P be defined as follows: $P \stackrel{\text{def}}{=} \left\{P_j : PV_j \in PV\right\}$ and let MAXP be defined as follows: MAXP $\stackrel{\text{def}}{=}$ $\max_{P_j \in P} |P_j|$. These concepts are illustrated in Figure 8.2c. Let $\mathbb{R}(\text{P}_\text{j})$ be defined as follows:

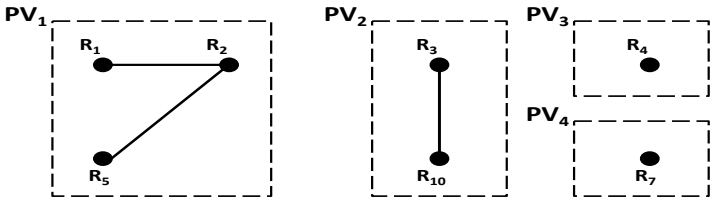
[1]Later in the chapter, it will be shown that all jobs of task τ_i only migrate between *two* processors as Phase-A and Phase-C of task τ_i will be assigned to the same processor.



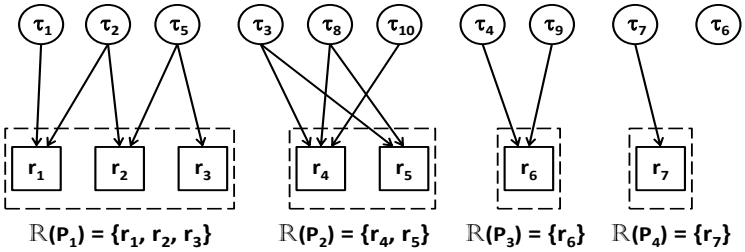
(a) A visualization of the resources requested by tasks. An arrow from a task to a resource indicates that the task requests the resource.



(b) Construction of the graph from resource sets requested. Each vertex has an associated resource set.



(c) The set $PV = \{PV_1, PV_2, PV_3, PV_4\}$ of connected components. From PV, we obtain set $P = \{P_1, P_2, P_3, P_4\}$ of resource request partitions where $P_1 = \{R_1, R_2, R_5\}$, $P_2 = \{R_3, R_{10}\}$, $P_3 = \{R_4\}$, $P_4 = \{R_7\}$ and MAXP $= 3$.



(d) The resource partition $\mathbb{R}(P_j)$ for each resource request partition P_j.

Figure 8.2: An example to illustrate the resource request information of tasks and how to construct the graph and connected components using this information.

$\mathbb{R}(\mathrm{P_j}) \stackrel{\text{def}}{=} \{r_\ell : \exists \tau_i \in \tau$ such that $R_i \in P_j$ and $r_\ell \in R_i\}$. Informally, $\mathbb{R}(\mathrm{P_j})$ denotes all the resources in resource request partition P_j. We refer to $\mathbb{R}(\mathrm{P_j})$ as a *resource partition*.

Note that, for each $P_j \in P$ and $P_{j'} \in P$ such that $P_j \neq P_{j'}$, the following statements are true:

1. $\mathbb{R}(P_j) \cap \mathbb{R}(P_{j'}) = \emptyset$ and
2. $\forall R_i \in P_j, \forall R_{i'} \in P_{j'}$, it holds that, $R_i \cap R_{i'} = \emptyset$

Also, note that for each task, τ_i, it holds that, there is at most one element, $P_k \in P$, such that: $R_i \in P_k$. Hence, the tasks in the given task set can be partitioned based on the resources they request. With this partitioning, it holds that, for two tasks in different partitions, there is no resource that they share. This is illustrated in Figure 8.2d.

Figure 8.3 and Figure 8.4 show two algorithms, ra-np-pEDF and ra-np-pEDF-fav, which we will use as building blocks in the design of our new algorithm. The ra-np-pEDF algorithm runs on an identical multiprocessor whereas the ra-np-pEDF-fav algorithm runs on a t-type heterogeneous multiprocessor. The ra-np-pEDF algorithm executes a task on a processor specific for its resource set and hence the execution of a task can only be delayed because of execution of another task whose resource set intersects with it. The ra-np-pEDF-fav algorithm works like ra-np-pEDF but it assumes that each task is assigned to a processor that is its favorite type (a type such that there is no other type for which the task has smaller execution time).

8.3 Overview of our algorithm

The algorithm, LP-EE-vpr, can be summarized in four steps as shown in Figure 8.5. Steps 1-3 are executed *before* run-time and only step 4 is executed *at* run-time. Step 1 produces subtasks from each task so that if the deadlines are met for these subtasks then the original task meets its deadline as well. Step 2 creates virtual processors from physical processors. Step 3 assigns subtasks to virtual processors. Finally, in Step 4, jobs are dispatched at run-time. We now provide more details about each of these steps.

Step 1 – Creation of subtasks. Categorize the execution of a task that requests a resource set into three phases as shown in Figure 8.6. The three phases of execution are phase-A, phase-B and phase-C, as mentioned in Section 8.2. Then create three constrained-deadline sporadic subtasks (one corresponding to each phase) out of each implicit-deadline sporadic task that requests a resource set and make different scheduling provisions for each of these subtasks. A task that does not request a resource set is categorized into phase-A alone and only one subtask is created for such a task.

For a task *that requests* a resource set, the "arrival" of both phase-B and phase-C subtasks have fixed offsets from the arrival of the respective phase-A subtask. This guarantees that the subtasks have the same inter-arrival time as the original task thereby exhibiting no jitter in their arrival times. Section 8.4.1 shows how these constrained-deadline subtasks are created and their parameters (worst-case execution times, minimum inter-arrival times and deadlines) are determined.

ra-np-pEDF (Resource-Aware-Non-Preemptive-Partitioned-EDF)	
Assumptions:	Consider a set R of resources and a task set such that whenever a task performs execution it must be holding its resource set. Consider a computer platform with $\|UNER\|$ or more identical processors.
Before run-time:	Select $\|UNER\|$ processors and call them ACT-processors and call the other processors NACT-processors. For ACT-processors, associate a resource set to each ACT-processor so that the following holds: (i) no two ACT-processors are associated with the same resource set in $UNER$ and (ii) no two resource sets in $UNER$ are associated with the same ACT processor and (iii) every ACT processor is associated with exactly one resource set in $UNER$ and (iv) every resource set in $UNER$ is associated with exactly one ACT processor. For NACT-processors, do not associate any resource set to these processors. A task is assigned to an ACT-processor whose associated resource set is equal to the resource set of the task.
At run-time:	A job is said to be *active* at time t if the arrival time of the job is $\leq t$ and the finishing time of the job is $\geq t$. A job J is said to be *eligible* at time t if it is active and no currently executing job holds a resource set that intersects with the resource set of job J. At each instant t, consider the set of active jobs in earliest-deadline-first order. If the current job is eligible then start its execution on the processor to which its corresponding task is assigned. If the current job is not eligible then do not execute it; consider the next job in the set of active jobs.

Figure 8.3: The behavior of ra-np-pEDF algorithm.

Step 2 – Creation of virtual processors. Virtual processors are logical constructs, used as task assignment targets by our algorithm[2]. Create two sets of virtual processors, namely, $\mathrm{VP_{AC}}$ and $\mathrm{VP_B}$ virtual processors, from the given physical processors. The $\mathrm{VP_B}$ virtual processors are then grouped together so as to create $|P|$ virtual processor groups, one group for every resource request partition in P. The virtual processor group corresponding to the resource request partition P_j is denoted as $\mathrm{Group_B[j]}$. The specification of the virtual processors (i.e., number of virtual processors and their speeds), their creation and grouping technique is discussed in Section 8.4.2.

Step 3 – Task assignment. The phase-A and phase-C subtasks created from a task τ_i are assigned to the same virtual processor in $\mathrm{VP_{AC}}$. The phase-B subtask created from task τ_i requesting the resource set, R_i, which is in a resource request partition, say P_j, i.e., $R_i \subseteq \mathbb{R}(\mathrm{P_j})$, is assigned to $\mathrm{Group_B[j]}$. This step is discussed in detail in Section 8.4.3.

Step 4 – Task scheduling. All phase-A and phase-C subtasks are scheduled using *preemptive* Earliest-Deadline-First (EDF) algorithm [LL73] on their assigned virtual processors in $\mathrm{VP_{AC}}$. All

[2]A virtual processor acts equivalent to a physical processor with speed $\frac{1}{f}$ and we assume that it can be "emulated" on a physical processor of speed 1, using no more than $\frac{1}{f}$ of its processing capacity. One intuitive way of achieving this is by dividing time into short slots of length S and using $\frac{1}{f} \times S$ time units in each slot to serve the workload of virtual processor. By selecting S, we can then make the speed of the emulated processor arbitrarily close to $\frac{1}{f}$ (and in practice, S need rarely be impractically short) [BA09].

ra-np-pEDF-fav (Resource-Aware-Non-Preemptive-Partitioned-EDF-Favorite-Processor)	
Assumptions:	Consider a set R of resources and a task set such that whenever a task performs execution it must be holding its resource set. Consider a t-type heterogeneous multiprocessor platform with $\|UNER\|$ or more identical processors of each type.
Before run-time:	For each type $k \in \{1,2,\ldots,t\}$, select $\|UNER\|$ processors and call them ACT-processors and call the other processors NACT-processors. For ACT-processors, associate a resource set to each ACT-processor so that for each type $k \in \{1,2,\ldots,t\}$ the following holds: (i) no two ACT-processors of type-k are associated with the same resource set in $UNER$ and (ii) no two resource sets in $UNER$ are associated with the same ACT processor of type-k and (iii) every ACT processor of type-k is associated with exactly one resource set in $UNER$ and (iv) every resource set in $UNER$ is associated with exactly one ACT processor of type-k. For NACT-processors, do not associate any resource set to these processors. A task is assigned to an ACT-processor whose associated resource set is equal to the resource set of the task and whose type is such that there is no other type where the task has smaller execution time.
At run-time:	A job is said to be *active* at time t if the arrival time of the job is $\leq t$ and the finishing time of the job is $\geq t$. A job J is said to be *eligible* at time t if it is active and no currently executing job holds a resource set that intersects with the resource set of job J. At each instant t, consider the set of active jobs in earliest-deadline-first order. If the current job is eligible then start its execution on the processor to which its corresponding task is assigned. (Note that since every task is assumed to be assigned to its favorite processor type, the jobs of each task execute on the respective favorite processor types). If the current job is not eligible then do not execute it; consider the next job in the set of active jobs.

Figure 8.4: The behavior of ra-np-pEDF-fav algorithm.

phase-B subtasks that are assigned to virtual processors in a VP_B virtual processor group are scheduled using ra-np-pEDF-fav.

Remark: In the rest of the chapter, to avoid tedium, we skip special mentioning of tasks that *do not request* a resource set (which are split into only phase-A) and hence, for such tasks, the discussion about phase-B and phase-C does not apply.

8.4 The new algorithm, LP-EE-vpr

In this section, we describe all the four steps of our new algorithm, LP-EE-vpr, in detail and also provide its pseudo-code.

8.4.1 Creating the subtasks

LP-EE-vpr creates three subtasks from each task, one subtask for each phase of the task and it assigns minimum inter-arrival time, deadlines and execution times to each subtask. Specifically,

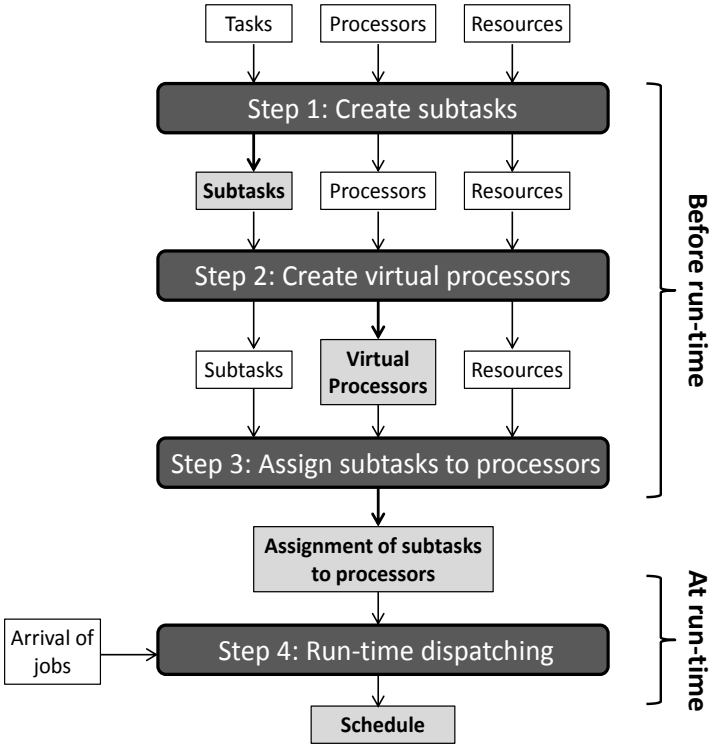


Figure 8.5: Four steps of our new algorithm, LP-EE-vpr. Each of the three first steps takes three inputs and produces outputs. Some outputs are identical to the inputs (e.g., in Step 1, “processors" are inputs and they are outputs) and they are marked in white. Some outputs, however, are produced (e.g., “subtasks" are outputs from Step 1 and they are not inputs to Step 1) and they are marked in gray.

each subtask will have t different execution times, one for each processor type and each subtask will also have t different deadlines, one for each processor type. When a subtask is assigned to a processor, only one of its execution times is applicable and only one of its deadlines is applicable; the type of processor on which the subtask is assigned determines this. The algorithm assigns parameters (minimum inter-arrival time, deadlines and execution times) to subtasks and assigns subtasks to processors so that when subtasks are scheduled at run-time it holds that (i) the three subtasks of a task execute in sequence (that is, one of the subtasks of τ_i must finish execution before another subtask of τ_i can start execution) and (ii) if each subtask of a task meets its deadline then the task from which these subtasks are formed meets its deadline as well.

From each implicit-deadline sporadic task, $\tau_i \in \tau$, the algorithm creates three constrained-deadline sporadic subtasks denoted by $\tau_{i,A}, \tau_{i,B}$ and $\tau_{i,C}$ corresponding to phase-A, phase-B and phase-C execution of task τ_i, respectively. In the rest of the chapter, the *subscript* A, B and C will be used in the notations corresponding to phase-A, phase-B and phase-C *subtasks*, respectively. Also, the superscript k will be used in the notations corresponding to a processor of type-k. For example, $C^k_{i,A}, C^k_{i,B}$ and $C^k_{i,C}$ denote the worst-case execution time of task $\tau_i \in \tau$ on a processor of type-k before requesting the resource set R_i (phase-A subtask), while holding the resource set

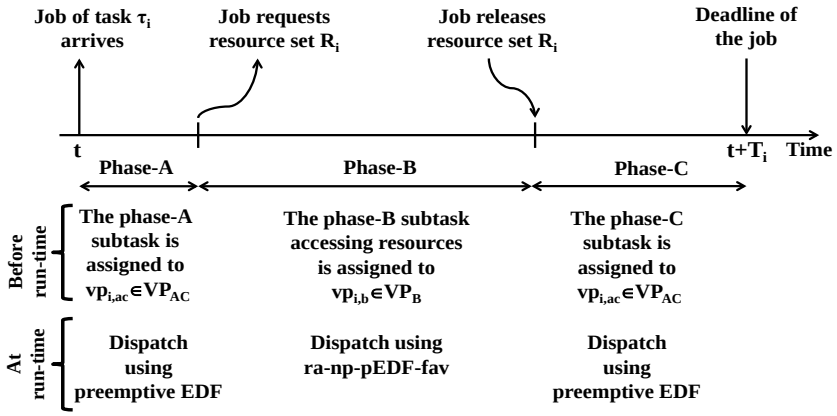


Figure 8.6: Three execution phases of a job along with the design-time and run-time decisions of LP-EE-vpr algorithm.

(phase-B subtask) and after releasing the resource set (phase-C subtask), respectively[3].

The parameters of the three subtasks, $\tau_{i,A}$, $\tau_{i,B}$ and $\tau_{i,C}$, that are derived from the corresponding task, $\tau_i \in \tau$, are set as shown in Table 8.1. It is easy to see that the following property holds: for each task $\tau_i \in \tau$ and for each pair of processor types k and k', it holds that: $D^k_{i,A} + D^{k'}_{i,B} + D^k_{i,C} \leq T_i = D_i$. This implies that, if for each task $\tau_i \in \tau$, it holds that, phase-A and phase-C of τ_i are assigned to the same processor type then if at run-time we can ensure that all subtasks meet their deadlines then the corresponding tasks meet their deadlines as well. Indeed, later in Section 8.4.3 while assigning subtasks to processors, we ensure that this property holds.

We group these derived subtasks into the following task sets:

$$\begin{aligned} \tau^A &= \{\tau_{i,A} \mid i \in \{1,2,\ldots,n\}\} \\ \tau^{B,\mathbb{R}(\mathrm{P_j})} &= \{\tau_{i,B} \mid i \in \{1,2,\ldots,n\} \wedge R_i \subseteq \mathbb{R}(\mathrm{P_j})\} \\ \tau^C &= \{\tau_{i,C} \mid i \in \{1,2,\ldots,n\}\} \end{aligned}$$

[3]Recall that, for a task that does not request a resource set, $C^k_{i,B}$ and $C^k_{i,C}$ do not exist.

Subtask of τ_i	WCET on type-k	Deadline on type-k	Minimum inter-arrival time
$\tau_{i,A}$	$C^k_{i,A} = CA^k_i$	$D^k_{i,A} = \frac{C^k_{i,A}}{C^k_i} \times \frac{T_i}{2}$	$T_{i,A} = T_i$
$\tau_{i,B}$	$C^k_{i,B} = CB^k_i$	$D^k_{i,B} = \frac{T_i}{2}$	$T_{i,B} = T_i$
$\tau_{i,C}$	$C^k_{i,C} = CC^k_i$	$D^k_{i,C} = \frac{C^k_{i,C}}{C^k_i} \times \frac{T_i}{2}$	$T_{i,C} = T_i$

Table 8.1: The parameters of the three constrained-deadline subtasks, $\tau_{i,A}$, $\tau_{i,B}$ and $\tau_{i,C}$, that are derived from the given implicit-deadline sporadic task, τ_i, that requests a resource set. For a task that does not request a resource set, only one subtask corresponding to phase-A execution, i.e., $\tau_{i,A}$, is derived and hence for such a task, $\tau_{i,B}$ and $\tau_{i,C}$ do not exist.

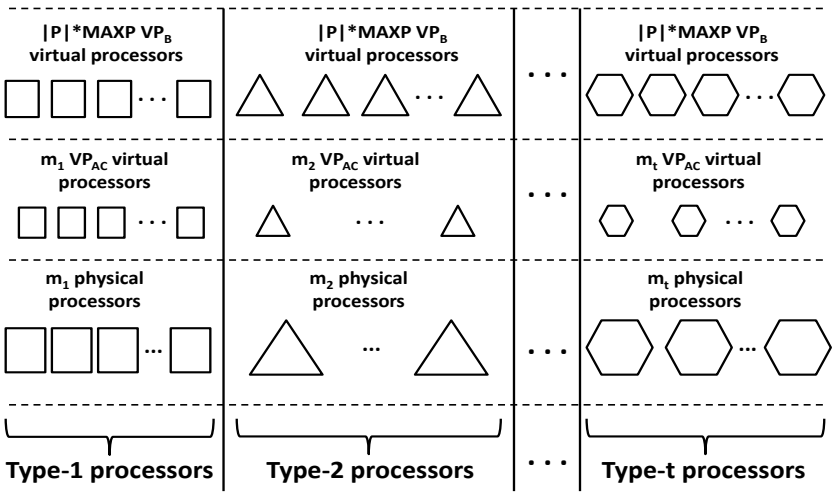


Figure 8.7: $m + t \times |P| \times$ MAXP virtual processors created from m physical processors of a t-type heterogeneous multiprocessor platform.

Note that $\tau_{i,A}$ refers to a *subtask* and τ^A refers to a *set of subtasks*. Analogously, for $\tau_{i,B}$ and $\tau^{B,\mathbb{R}(\mathrm{P_j})}$. Analogously, for $\tau_{i,C}$ and τ^C.

As opposed to the given task set τ which contains implicit-deadline sporadic tasks, these derived task sets contain constrained-deadline sporadic subtasks. Also, observe that, the task set τ^A is derived such that, on a processor of type-k, the density of every subtask, $\tau_{i,A} \in \tau^A$, is twice the utilization of the corresponding task, $\tau_i \in \tau$. Formally,

$$\forall \tau_{i,A} \in \tau^A : \ \delta^k_{i,A} = \frac{C^k_{i,A}}{D^k_{i,A}} = \frac{C^k_{i,A}}{\frac{C^k_{i,A} \times T_i}{C^k_i \times 2}} = \frac{2 \times C^k_i}{T_i} = 2 \times u^k_i \text{ of } \tau_i \in \tau \tag{8.1}$$

Analogously, it can be seen that, the density of every subtask, $\tau_{i,C} \in \tau^C$, is twice the utilization of the corresponding task, $\tau_i \in \tau$.

8.4.2 Creating virtual processors

In this section, we describe the creation of virtual processors from the given physical processors of a t-type heterogeneous multiprocessor platform.

LP-EE-vpr creates $m + t \times |P| \times$ MAXP virtual processors from the given m physical processors as shown in Figure 8.7. The main idea is as follows. LP-EE-vpr treats physical processors of each type as an identical multiprocessor platform and creates a certain number of virtual processors of the corresponding type from this platform. To be precise, m_k physical processors of type-k are treated as an identical multiprocessor platform and $m_k + |P| \times$ MAXP virtual processors of type-k are created from them (see different columns in Figure 8.7, separated by "solid vertical lines") and ordered as shown in Figure 8.7. Now, if we look at the first and the second row in

Figure 8.7 (separated by "dashed horizontal lines"), each of these rows represent a t-type heterogeneous multiprocessor platform of virtual processors — the first row represents a t-type heterogeneous multiprocessor platform with $t \times |P| \times \text{MAXP}$ virtual processors of which $|P| \times \text{MAXP}$ virtual processors are of type-k ($\forall k : k \in \{1,2,\ldots,t\}$) and the second row represents a t-type heterogeneous multiprocessor platform with m virtual processors of which m_k virtual processors are of type-k ($\forall k : k \in \{1,2,\ldots,t\}$). In this manner, $m + t \times |P| \times \text{MAXP}$ virtual processors are created from m physical processors of a t-type heterogeneous multiprocessor platform. Precisely, LP-EE-vpr creates virtual processors with the following specifications:

- m **virtual processors (denoted as** VP_{AC}**):** From m_k physical processors of type-k, it creates m_k virtual processors of type-k ($\forall k : k \in \{1,2,\ldots,t\}$) each of speed $\frac{1}{1+\text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}$ times the speed of a corresponding physical processor of type-k. So, in total, m such virtual processors are created from m physical processors. These are later used to schedule phase-A and phase-C subtasks and are referred to as 'VP_{AC} virtual processors'.

- $t \times |P| \times \text{MAXP}$ **virtual processors (denoted as** VP_{B}**):** From m_k physical processors of type-k, it creates $|P| \times \text{MAXP}$ virtual processors of type-k ($\forall k : k \in \{1,2,\ldots,t\}$) each of speed $\frac{\text{MAXP}}{1+\text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}$ times the speed of a corresponding physical processor of type-k. So, in total, $t \times |P| \times \text{MAXP}$ such virtual processors are created from m physical processors of a t-type heterogeneous multiprocessor platform. These are later used to schedule phase-B subtasks and are referred to as 'VP_{B} virtual processors'.

In other words, from each processor type, say type-k, LP-EE-vpr creates $m_k + |P| \times \text{MAXP}$ virtual processors of type-k, i.e., $m_k\,\text{VP}_{\text{AC}}$ virtual processors of type-k and $|P| \times \text{MAXP}\,\text{VP}_{\text{B}}$ virtual processors of type-k. The way these virtual processors are created is as follows. From each processor π_p of type-k ($\forall k : k \in \{1,2,\ldots,t\}$):

- initially, one VP_{AC} virtual processor of type-k is created which is of speed $\frac{1}{1+\text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}$ times the speed of π_p

- later, $\left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil$ VP_{B} virtual processors of type-k are created each of which is of speed $\frac{\text{MAXP}}{1+\text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}$ times the speed of π_p

Lemma 47. *The earlier specified set of virtual processors,* VP_{AC} *and* VP_{B}*, can be created from the given t-type heterogeneous multiprocessor platform* π *as described above. This procedure to create the virtual processors ensures that the capacity of a virtual processor comes from a single physical processor.*

Proof. The proof is a direct consequence of the fact that each physical processor of type-k can emulate one VP_{AC} virtual processor of type-k ($\forall k : k \in \{1,2,\ldots,t\}$) and $\left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil$ VP_{B} virtual processors of type-k, as per the specifications of the virtual processors. Indeed, for each $\pi_p \in \pi$,

we have

$$\underbrace{1 \times \frac{1}{1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}}_{\text{VP}_{\text{AC}} \text{ virtual processor}} + \underbrace{\left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil \times \frac{\text{MAXP}}{1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil}}_{\text{VP}_{\text{B}} \text{ virtual processors}} = 1$$

Thus, m_k physical processors of type-k can emulate m_k VP_{AC} virtual processors of type-k and $\left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil \times m_k \geq |P| \times \text{MAXP}\,\text{VP}_{\text{B}}$ virtual processors of type-k. Overall, m physical processors of a t-type heterogeneous multiprocessor platform can emulate m VP_{AC} virtual processors and $t \times |P| \times \text{MAXP}$ VP_{B} virtual processors.

From the above discussion, it is trivial to see that no virtual processor is created using two or more physical processors and hence it holds that, the capacity of a virtual processor comes from a single physical processor alone. Hence the proof. □

We now describe the rest of the steps in the algorithm, LP-EE-vpr, for assigning and scheduling the tasks with the help of pseudo-code.

8.4.3 Pseudo-code of LP-EE-vpr algorithm

The pseudo-code of LP-EE-vpr algorithm is shown in Algorithm 16. The algorithm works as follows.

On line 1, it creates the sets τ^A, $\tau^{B,\mathbb{R}(\text{P}_\text{j})}$ and τ^C of constrained-deadline sporadic subtasks from the given set τ of implicit-deadline sporadic tasks as described in Section 8.4.1.

On line 2, it creates $m\,\text{VP}_{\text{AC}}$ and $t \times |P| \times \text{MAXP}\,\text{VP}_{\text{B}}$ virtual processors from the given t-type heterogeneous multiprocessor platform of m physical processors as discussed in Section 8.4.2.

On line 3, it groups $t \times |P| \times \text{MAXP}$ VP_{B} virtual processors into $|P|$ *groups of* VP_{B} *virtual processors*; each group contains $t \times \text{MAXP}\,\text{VP}_{\text{B}}$ virtual processors, with MAXP virtual processors of each type, i.e., MAXP virtual processors of type-1, MAXP virtual processors of type-2 and so on. Each group of virtual processors, denoted by $\text{Group}_{\text{B}}[\text{j}]$, where $j = \{1, 2, \ldots, |P|\}$, is used for scheduling phase-B subtasks that access a subset of resources from resource partition $\mathbb{R}(\text{P}_\text{j})$.

On line 4, it assigns the set of phase-A subtasks, τ^A, to VP_{AC} virtual processors using LP-EE algorithm[Bar04c][4]. The algorithm, LP-EE, is designed for non-migratively scheduling a set of implicit-deadline sporadic tasks that *do not* share resources on t-type heterogeneous multiprocessors. The internals of LP-EE and its performance bound are described in detail in [Bar04c]. The average-case performance of LP-EE is discussed in [RAB13]. Therefore, we only give an overview of LP-EE here. The algorithm, LP-EE, has two steps: first, it assigns the tasks to processors and then schedules the tasks on each processor using preemptive EDF. The task assignment step works as follows:

- The assignment problem is formulated as Mixed Integer Linear Program (MILP) and then relaxed to Linear Program (LP). The LP formulation is solved using an LP solver (such as

[4]We selected LP-EE because it is simple to implement and easy to explain and it has a proven speed competitive ratio. However, a couple of other algorithms can be used instead as discussed later in Section 8.6.5

Algorithm 16: LP-EE-vpr($\tau, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}), R$): Algorithm for scheduling implicit-deadline sporadic tasks that share resources on t-type heterogeneous multiprocessors

```
// Lines 1-10 execute before run-time; line 11 executes at
   run-time.
```

1 Create the sets τ^A, $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ and τ^C of constrained-deadline sporadic subtasks from the given task set τ of implicit-deadline sporadic tasks as described in Section 8.4.1.

2 Create m $\mathrm{VP_{AC}}$ and $t \times |P| \times \mathrm{MAXP}$ $\mathrm{VP_B}$ virtual processors from the given m physical processors of a t-type heterogeneous multiprocessor platform as described in Section 8.4.2.

3 Form $|P|$ virtual processor *groups* out of $t \times |P|$ $\mathrm{VP_B}$ virtual processors as follows. Take MAXP $\mathrm{VP_B}$ virtual processors of each type (i.e., $t \times$ MAXP virtual processors, in total) and form a virtual processor group, $\mathrm{Group_B}[1]$; then take MAXP more $\mathrm{VP_B}$ virtual processors of each type and form another virtual processor group, $\mathrm{Group_B}[2]$ and so on. Overall, we will have $|P|$ $\mathrm{VP_B}$ virtual processor groups; every group containing $t \times$ MAXP $\mathrm{VP_B}$ virtual processors of which MAXP virtual processors are of type-k, where $k \in \{1, 2, \ldots, t\}$.

4 Assign all the subtasks $\tau_{i,A} \in \tau^A$ to $\mathrm{VP_{AC}}$ virtual processors using the algorithm LP-EE [Bar04c] (more details in the description of the algorithm in Section 8.4.3).

5 **foreach** $\tau_i \in \tau$ **do**

6 | **if** $\left(\exists j : j \in \{1, 2, \ldots, |P|\} \ \wedge \ R_i \subseteq \mathbb{R}(\mathrm{P_j})\right)$ **then**

7 | | Assign $\tau_{i,B}$ to the MAXP virtual processors in the j'th $\mathrm{VP_B}$ virtual processor *group*, $\mathrm{Group_B}[\mathrm{j}]$, on which subtask $\tau_{i,B}$ has the smallest execution time.

8 | **end**

9 **end**

10 Assign every subtask $\tau_{i,C} \in \tau^C$ to that virtual processor in $\mathrm{VP_{AC}}$ to which the corresponding subtask $\tau_{i,A} \in \tau^A$ has been assigned on line 4.

11 Schedule the subtasks of τ^A and τ^C that are assigned on each $\mathrm{VP_{AC}}$ virtual processor using *preemptive* EDF on that virtual processor. Schedule the subtasks of $\tau_{i,B}$ that are assigned to each $\mathrm{VP_B}$ virtual processor group using ra-np-pEDF-fav, on the respective virtual processor group.

GUROBI Optimizer [Gur13] or IBM ILOG CPLEX [IBM12]). Tasks are then assigned to the processors according to the values of the respective *indicator variables* in the solution provided by the solver. Using certain tricks [Pot85], it is shown that, there exists a solution (for example, the solution that lies on the vertex of the feasible region) to the LP formulation in which all but at most $m-1$ tasks are integrally assigned to processors and such a solution can be obtained, where m denotes the number of processors.

- The remaining at most $m-1$ tasks are integrally assigned on the remaining capacity of the processors using "exhaustive enumeration".

The abbreviation LP-EE comes from the fact that the algorithm makes use of **L**inear **P**rogramming and **E**xhaustive **E**numeration techniques to provide the solution [Bar04c].

On lines 5–9, it assigns all the phase-B subtasks that request the "related" resources, i.e., resources that belong to the same resource partition, to the same $\mathrm{VP_B}$ virtual processor group. Specifically, all the subtasks requesting (a subset of) resources from resource partition $\mathbb{R}(\mathrm{P_j})$,

$\forall j \in \{1, 2, \ldots, |P|\}$, are assigned to the virtual processors in the j'th VP_B virtual processor group, $\text{Group}_\text{B}[\text{j}]$, on which these subtasks have the smallest execution time.

On line 10, it assigns every phase-C subtask, $\tau_{i,C}$, to that virtual processor in VP_AC to which the corresponding phase-A subtask, $\tau_{i,A}$, has been assigned (on line 4). Such an assignment does not endanger the schedulability of the tasks assigned on the VP_AC virtual processors as there is a precedence constraint between these subtasks — this is formally proven later in Lemma 55 in Section 8.5.3. Also, such an assignment ensures that the *number of migrations per job is restricted to at most two*. This is easy to verify because both phase-A and phase-C of a task execute on the same physical processor as they are assigned to the same virtual processor (recall that the capacity of a virtual processor comes from a single physical processor — Lemma 47) and only the phase-B subtask *might* have to execute on a different physical processor as the virtual processor to which phase-B of the task is assigned might have been created from a different physical processor. Hence, it can be seen that, for a given job, one migration may happen when the job requests the resource set and another migration may happen when the job releases the resource set.

On line 11, it schedules the subtasks of τ^A and τ^C that are assigned to each VP_AC virtual processor using *preemptive* EDF on that virtual processor. It schedules the subtasks of $\tau^{B,\mathbb{R}(\text{P}_\text{j})}$ that are assigned to each VP_B virtual processor group, $\text{Group}_\text{B}[\text{j}]$, using ra-np-pEDF-fav algorithm (listed in Figure 8.4), on the respective virtual processor group. Recall that, all the tasks in $\tau^{B,\mathbb{R}(\text{P}_\text{j})}$ request (a subset of) resources from resource partition $\mathbb{R}(\text{P}_\text{j})$ and hence are assigned to VP_B virtual processor group, $\text{Group}_\text{B}[\text{j}]$.

For *preemptive EDF* scheduling, the following result is well-known (an easily obtained generalization of the result shown in [LL73]), which we make use of while proving the performance of LP-EE-vpr.

Lemma 48. *(utilization-based schedulability test)*
Let $\tau[\pi_p]$ denote the tasks assigned on a processor π_p of type-k. If $\sum_{\tau_i \in \tau[\pi_p]} u_i^k \leq 1$ and tasks are scheduled with preemptive EDF on π_p then all deadlines are met.

Note that in Algorithm 16, lines 1–10 execute *before* run-time and only line 11 executes *at* run-time. The algorithm, LP-EE-vpr, is named after the fact that it makes use of the algorithm, LP-EE, for assigning some of the subtasks on ***v***irtual ***pr***ocessors.

8.5 Speed competitive ratio of LP-EE-vpr algorithm

In this section, we prove the speed competitive ratio of the proposed algorithm. But first we present notations (in Section 8.5.1) and then prove the speed competitive ratio of ra-np-pEDF algorithm (in Section 8.5.2). After that, we present some useful results (a previously known and a few new results, in Section 8.5.3) and the speed competitive ratio of ra-np-pEDF-fav algorithm that are used later while proving the speed competitive ratio of LP-EE-vpr algorithm (in Section 8.5.4).

8.5.1 Notations

Let $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ denote a t-type heterogeneous multiprocessor platform of m processors of which m_k processors are of type-k, where $k \in \{1,2,\ldots,t\}$ and $\forall k : m_k > 0$; note that $m = m_1 + m_2 + \ldots + m_t$.

Let $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times \langle \mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_\mathrm{t} \rangle$ denote a t-type platform in which, $\forall k \in \{1,2,\ldots,t\}$, it holds that, the speed of every type-k processor is s_k times the speed of a corresponding type-k processor in $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$, where $s_k > 0$ is a real number. As a special case of the above, we use $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times \langle \mathrm{s}, \mathrm{s}, \ldots, \mathrm{s} \rangle$ to denote a t-type platform in which, for each $k \in \{1,2,\ldots,t\}$, the speed of every type-k processor is s times the speed of a corresponding type-k processor in $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$, where $s > 0$ is a real number. For convenience, we sometimes denote $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times \langle \mathrm{s}, \mathrm{s}, \ldots, \mathrm{s} \rangle$ as $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times \mathrm{s}$.

If τ is a task set and y, y', y'' are positive real numbers then we let $\mathrm{mulCDT}(\tau, y, y', y'')$ denote a task set where for each task in τ: its execution time is multiplied by y; its deadline is multiplied by y' and its minimum inter-arrival time is multiplied by y''.

We will now introduce three types of predicates (i) predicates that state if a task set is *schedulable* for a given scheduling algorithm, (ii) predicates that state if a task set is *feasible* and (iii) predicates that state if a task set is *schedulable* for a given scheduling algorithm *according to a certain class of schedulability tests*.

For a task set τ where tasks do not share resources, let $\mathrm{sched}(A, \tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}))$ be a predicate that indicates that, if task set τ is scheduled by algorithm A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ then for each set of jobs that τ can generate according to the model described in Section 8.2, it holds that, all jobs meet their deadlines and the constraint of restricted migration is satisfied (which in this case means that no migration is allowed because there are only phase-A executions).

For a task set τ where tasks may share resources from a set R of resources, we let the symbol $\mathrm{sched}(A, \tau, R, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}))$ be a predicate that indicates that, if τ is scheduled by algorithm A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ then for each set of jobs that τ can generate according to the model described in Section 8.2, it holds that, all jobs meet their deadlines and the constraint of restricted migration is satisfied and there is no instant where a resource in R is held by more than one job. Analogously, for a task set τ where tasks may share resources in R, and where P_j is a resource set and $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ is the task set derived as in Section 8.4.1, we let the symbol $\mathrm{sched}\big(A, \tau^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})\big)$ be a predicate that indicates that, if $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ is scheduled by algorithm A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ then for each set of jobs that $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ can generate according to the model described in Section 8.2, it holds that, all jobs meet their deadlines and the constraint of restricted migration is satisfied (which in this case means that, no migration is allowed because there are only phase-B executions) and there is no instant where a resource in $\mathbb{R}(\mathrm{P_j})$ is held by more than one job.

For a task set τ where tasks do not share resources, let $\text{nmig-feas}(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}))$ be a predicate that indicates that, for each set of jobs that τ can generate according to the model described in Section 8.2, it holds that, there exists a schedule that meets all deadlines of all jobs

and the constraint of restricted migration is satisfied (which in this case means that, no migration is allowed because there are only phase-A executions).

For a task set τ where tasks may share resources from a set R of resources, we let the symbol rmig-feas $(\tau, R, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}))$ be a predicate that indicates that, for each set of jobs that τ can generate according to the model described in Section 8.2, it holds that, there exists a schedule that meets all deadlines of all jobs and the constraint of restricted migration is satisfied and there is no instant where a resource in R is held by more than one job. Analogously, for a task set τ where tasks may share resources in R, and where P_j is a resource set and $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ is the task set derived from τ as in Section 8.4.1, we let the symbol rmig-feas $\left(\tau^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})\right)$ be a predicate that indicates that, for each set of jobs that $\tau^{B,\mathbb{R}(\mathrm{P_j})}$ can generate according to the model described in Section 8.2, it holds that, there exists a schedule that meets all deadlines of all jobs and the constraint of restricted migration is satisfied (which in this case means that, no migration is allowed because there are only phase-B executions) and there is no instant where a resource in $\mathbb{R}(\mathrm{P_j})$ is held by more than one job.

Some of these predicates will be used by adding a suffix "-δ" to the scheduling algorithm or algorithm class where applicable, for example, for non-migrative scheduling of constrained-deadline sporadic subtasks corresponding to different phases. Such predicates with suffix -δ signify that the *schedulability* of the task set other than just being established via some exact test, must additionally be ascertainable via a (potentially pessimistic) *density-based uniprocessor schedulability test* (similar to Lemma 48). That is, for $\tau[\pi_p]$ of tasks assigned on a processor π_p of type-k, to meet all deadlines, it must hold that: $\sum_{\tau_i \in \tau[\pi_p]} \delta_i^k \leq 1$. For example, sched(A-$\delta, \tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$) denotes a predicate that is true if for the task set τ which does *not* share resources is ascertained schedulable by algorithm A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ using the above mentioned density-based schedulability test.

We use a function, `create-fav-taskset`$(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}))$. This function takes a task set τ as input in which each task, $\tau_i \in \tau$, is characterized by its minimum inter-arrival time, T_i, and its deadline, D_i, and its t worst-case execution times (one WCET on each processor type), C_i^1, C_i^2, ..., C_i^t. The function outputs a task set τ' in which each task, $\tau_i' \in \tau'$, is characterized by its minimum inter-arrival time, T_i', its deadline, D_i', and its single worst-case execution time, C_i'. For each task, $\tau_i' \in \tau'$, it sets $T_i' = T_i$ and $D_i' = D_i$ and $C_i' = \min_{k \in \{1,2,\ldots,t\}} C_i^k$. Informally, from the given task set, it constructs another task set in which, the execution time of each task is equal to the execution time of its corresponding task on its favorite processor type and the minimum inter-arrival time of each task is equal to the minimum inter-arrival time of its corresponding task and the deadline of each task is equal to the deadline of its corresponding task.

We also use a function `create-fav-platform`$(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}), m')$ which generates a multiprocessor platform with m' identical processors where each processor is such that for each task in τ it holds that the execution time is as if it executed on the processor type in $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$ for which its execution time is the smallest.

8.5.2 Speed competitive ratio of ra-np-pEDF-fav algorithm

Recall (from step 11 of Algorithm 16 in Section 8.4.3), that the LP-EE-vpr algorithm uses the ra-np-pEDF-fav algorithm (defined in Section 8.2) to schedule phase-B execution of tasks. For this reason, we need to show that, ra-np-pEDF-fav algorithm has a finite speed competitive ratio. We will do so by first showing the speed competitive ratio of ra-np-pEDF algorithm and later show (in Section 8.5.3) how it translates to a heterogeneous multiprocessor.

As a by-product of our proof of the speed competitive ratio of ra-np-pEDF algorithm, we obtain a corollary which is a new result on the speed competitive ratio of non-preemptive EDF scheduling on a single processor. Previously, it was known that the speed competitive ratio of non-preemptive EDF on a single processor is at most *three*. In this section, we see that it is at most *two*.

We start by proving a relationship between the feasibility of a set of tasks that executes always holding a resource and the feasibility of this task set on an identical multiprocessor.

Lemma 49. *$\forall\tau$, $\forall\Pi(m_1, m_2, \ldots, m_t)$, $\forall R$, $v \geq |UNER|$ such that τ is an implicit-deadline sporadic task set and $\forall\tau_i \in \tau : R_i \neq \emptyset$ and $\forall\tau_i \in \tau$, it holds that whenever τ_i executes, it holds the resource set R_i:*

$$\text{rmig-feas}\Big(\tau, R, \Pi(m_1, m_2, \ldots, m_t)\Big) \Longrightarrow$$
$$\text{rmig-feas}\Big(\texttt{create-fav-taskset}\Big(\tau, \Pi(m_1, m_2, \ldots, m_t)\Big), R,$$
$$\texttt{create-fav-platform}\Big(\tau, \Pi(m_1, m_2, \ldots, m_t), v\Big)\Big)$$

Proof. The lemma follows from two observations:

1. The task set τ is such that at each instant, there can be at most $|UNER|$ jobs executing at this instant.

2. If a task set is feasible then giving each task an execution time as if it executed on the processor where its execution time is smallest cannot violate feasibility.

The truth of the first observation can be seen as follows: Suppose that the first observation was false. Then there would exist a feasible schedule such that there exists an instant where $|UNER|$+1 or more jobs execute at that instant. Then it follows that, there are two or more jobs that execute holding the same resource set in $UNER$. Consequently, this schedule is not feasible. Hence the first observation is true.

The truth of the second observation can be seen as follows: For a feasible schedule, if we change the execution time of a job to a smaller value then we can simply idle the processor so that the schedule for all other jobs are the same and hence feasibility is not violated by reducing the execution time of a job. □

We can then show (below) how the *feasibility* relates to the *schedulability* of ra-np-pEDF.

Lemma 50. *$\forall\tau$, $\forall\Pi(m_1, m_2, \ldots, m_t)$, $\forall R$, $\forall x \geq 1$, $v \geq |UNER|$ such that τ is an implicit-deadline sporadic task set and $\forall\tau_i \in \tau : R_i \neq \emptyset$ and $\forall\tau_i \in \tau$ it holds that whenever τ_i executes it holds resource set R_i:*

$$\text{rmig-feas}\left(\texttt{create-fav-taskset}\left(\tau,\Pi(m_1,m_2,\ldots,m_t)\right),R,\right.$$
$$\left.\texttt{create-fav-platform}\left(\tau,\Pi(m_1,m_2,\ldots,m_t),v\right)\right) \Longrightarrow$$
$$\text{sched}\left(\text{ra-np-pEDF, mulCDT}\left(\texttt{create-fav-taskset}\left(\tau,\Pi(m_1,m_2,\ldots,m_t)\right),\right.\right.$$
$$\left.\left.\frac{1}{2\times v\times x},\frac{1}{x},1\right),R,\texttt{create-fav-platform}\left(\tau,\Pi(m_1,m_2,\ldots,m_t),v\right)\right)$$

Proof. The proof is by contradiction. Suppose that the claim of the lemma is false. Then there exists a $\tau,\Pi(m_1,m_2,\ldots,m_t),R,x \geq 1, v \geq |UNER|$ such that τ is an implicit-deadline sporadic task set and $\forall\tau_i \in \tau : R_i \neq \emptyset$ and $\forall\tau_i \in \tau$, it holds that, whenever τ_i executes, it holds resource set R_i for which it holds that:

$$\text{(Expression (8.2) is true)} \ \wedge \ \text{(Expression (8.3) is false)}$$

where Expression (8.2) and Expression (8.3) are defined as:

$$\text{rmig-feas}\left(\texttt{create-fav-taskset}\left(\tau,\Pi(m_1,m_2,\ldots,m_t)\right),R,\right.$$
$$\left.\texttt{create-fav-platform}\left(\tau,\Pi(m_1,m_2,\ldots,m_t),v\right)\right) \tag{8.2}$$

$$\text{sched}\left(\text{ra-np-pEDF, mulCDT}\left(\texttt{create-fav-taskset}\left(\tau,\Pi(m_1,m_2,\ldots,m_t)\right),\right.\right.$$
$$\left.\left.\frac{1}{2\times v\times x},\frac{1}{x},1\right),R,\texttt{create-fav-platform}\left(\tau,\Pi(m_1,m_2,\ldots,m_t),v\right)\right) \tag{8.3}$$

Note that both Expression (8.2) and Expression (8.3) make statements about a task set and a multiprocessor platform with identical processors. Since it is an identical multiprocessor, we do not need to specify execution times as depending on processor type and hence, we let C_j denote the execution time of task τ_j for the task set in Expression (8.2). Because of our assumption that the task set τ is an implicit-deadline sporadic task set and because Expression (8.2), it follows that:

$$(C_1 \leq D_1 = T_1) \ \wedge \ (C_2 \leq D_2 = T_2) \ \wedge \ \cdots \ \wedge \ (C_n \leq D_n = T_n) \tag{8.4}$$

We will now discuss the implication of Expression (8.3) being false. Since Expression (8.3) is false, it follows that, there exist an assignment of arrival times to jobs such that a deadline is missed. Let t_0 denote the earliest time when a deadline is missed. Let us choose a job whose

deadline expires at time t_0 and let us call it DMJ (deadline miss job). Let t_2 denote the arrival time of the job DMJ. Let τ_k denote the task that generated DMJ. From Expression (8.4), we get:

$$C_k \leq D_k = T_k \tag{8.5}$$

Let $S(\tau_k)$ be defined as:

$$S(\tau_k) = \{\tau_{k'} : \big(\tau_{k'} \in \texttt{create-fav-taskset}\left(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_t)\right)\big) \\ \wedge\ (\tau_{k'} \neq \tau_k)\ \wedge\ (|R_{k'} \cap R_k| \geq 1)\} \tag{8.6}$$

$S(\tau_k)$ is the set of tasks that can share a resource with task τ_k. If $|S(\tau_k)| = 0$ then DMJ would have executed immediately when it arrived and because of Expression (8.5) and because $\frac{1}{2\times v\times x} \leq \frac{1}{x}$ it would follow that τ_k would have met its deadline and this would be a contradiction. Hence, we know that:

$$|S(\tau_k)| \geq 1 \tag{8.7}$$

Let $BLT(\tau_k, DMJ, t_2)$ be defined as:

$$BLT(\tau_k, DMJ, t_2) = \{\tau_{k'} : (\tau_{k'} \in S(\tau_k))\ \wedge\ (\text{there is a job of task } \tau_{k'} \text{ executing at time } t_2)\} \tag{8.8}$$

Informally, $BLT(\tau_k, DMJ, t_2)$ is the set of tasks in $S(\tau_k)$ such that these tasks executed at time t_2. Let $BLJ(\tau_k, DMJ, t_2)$ be defined as the set of jobs generated by $BLT(\tau_k, DMJ, t_2)$ such that the jobs executed at time t_2. Clearly, for each element in $BLJ(\tau_k, DMJ, t_2)$, there is a corresponding element in $BLT(\tau_k, DMJ, t_2)$. Intuitively, BLT means "blocking-tasks" and BLJ means "blocking-jobs".

Let us explore two cases:

1. $|BLT(\tau_k, DMJ, t_2)| \geq 1$.

   Let t_1 denote maximum of the finishing times of the jobs in $BLJ(\tau_k, DMJ, t_2)$. Let us choose a job in $BLJ(\tau_k, DMJ, t_2)$ that finished at time t_1 and let the task that generated this job be denoted τ_i and let t_b denote the starting time of this job. From the definition of t_2, we have $t_b \leq t_2$.

   We will now discuss the time interval [t_b,t_0) and we let L denote the duration of this time interval (that is $L = t_0 - t_b$). During this time, at each instant t, at least one of the following is true: (i) the set of jobs executing at time t includes a job of task τ_i or (ii) the set of jobs executing at time t includes DMJ (the job of task τ_k) or (iii) the set of jobs executing at time t includes a job of a task in $S(\tau_k) \setminus \{\tau_i\}$.

Since we had a deadline miss, we obtain that:

$$\frac{C_i}{2\times v\times x}+\max(0,\lfloor\frac{L-\frac{D_k}{x}}{T_k}\rfloor+1)\times\frac{C_k}{2\times v\times x}+\sum_{\tau_{i'}\in(S(\tau_k)\setminus\{\tau_i\})}\max(0,\lfloor\frac{L-\frac{D_{i'}}{x}}{T_{i'}}\rfloor+1)\times\frac{C_{i'}}{2\times v\times x}\ >\ L \tag{8.9}$$

Using Expression (8.4) on Expression (8.9) and rewriting yields:

$$\frac{C_i}{2\times v\times x}+\max(0,\lfloor\frac{L-\frac{T_k}{x}+T_k}{T_k}\rfloor)\times\frac{C_k}{2\times v\times x}+\sum_{\tau_{i'}\in(S(\tau_k)\setminus\{\tau_i\})}\max(0,\lfloor\frac{L-\frac{T_{i'}}{x}+T_{i'}}{T_{i'}}\rfloor)\times\frac{C_{i'}}{2\times v\times x}\ >\ L \tag{8.10}$$

Since at time t_2, there is a job of task τ_i executing and it follows that this job of task τ_i started to execute at time t_2 or earlier. Since t_b is defined as the starting time of this job we obtain: $t_b \leq t_2$. This gives us:

$$t_0 - t_2\ \leq\ t_0 - t_b \tag{8.11}$$

Note that $t_0 - t_2 = D_k/x$. Also note that, $t_0 - t_b = L$. This gives us:

$$\frac{D_k}{x} \leq L \tag{8.12}$$

Using Expression (8.4) on Expression (8.12) yields:

$$\frac{T_k}{x} \leq L \tag{8.13}$$

We will now discuss the implication of Expression (8.2) being true. Since Expression (8.2) is true, it follows that, for every possible assignment of arrival times to jobs in the task set `create-fav-taskset` $(\tau,\Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t}))$, all deadlines are met on an identical multiprocessor with v processors and where it is required that the resource sharing constraints are respected. Let us consider the case that, tasks arrive periodically. Then it follows that, there exist a time when a job of task τ_i arrives. And since deadlines are met, this job must have finished at most T_i time units later and hence there exist a time when a job of task τ_i executed. Let tarbegin denote the time when this job of task τ_i started to execute and let tarend denote the time L' time units later. (Clearly, tarend-tarbegin = L'.) We can also observe that, for some other task $\tau_{i'}$, it holds that, at each instant, a job of task $\tau_{i'}$ arrives at most $T_{i'}$ time units later. Hence, during this time interval [tarbegin,tarend] (of duration L'),

there are at least

$$\lfloor \frac{L'}{T_{i'}} \rfloor \tag{8.14}$$

jobs of task $\tau_{i'}$ with arrival time within [tarbegin,tarend].

Hence, during this time interval [tarbegin,tarend] (of duration L'), there are at least

$$\max(0, \lfloor \frac{L' - D_{i'}}{T_{i'}} \rfloor) \tag{8.15}$$

jobs of task $\tau_{i'}$ with arrival time and deadline within [tarbegin,tarend].

Using Expression (8.4) gives us that during this time interval [tarbegin,tarend] (of duration L'), there are at least

$$\max(0, \lfloor \frac{L' - T_{i'}}{T_{i'}} \rfloor) \tag{8.16}$$

jobs of task $\tau_{i'}$ with arrival time and deadline within [tarbegin,tarend].

Note that, for the feasible schedule, at each instant, there can be at most v jobs executing (because otherwise there would be two jobs executing while holding the same resource set). With this observation and using Expression (8.16) gives us:

$$\min(C_i, L') + \max(0, \lfloor \frac{L' - T_k}{T_k} \rfloor) \times C_k + \sum_{\tau_{i'} \in (S(\tau_k) \setminus \{\tau_i\})} \max(0, \lfloor \frac{L' - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \leq v \times L' \tag{8.17}$$

Expression (8.17) applies for any choice of L'. Applying it with $L' = 2L \times x$ gives us:

$$\min(C_i, 2L \times x) + \max(0, \lfloor \frac{2L \times x - T_k}{T_k} \rfloor \times C_k) + \sum_{\tau_{i'} \in (S(\tau_k) \setminus \{\tau_i\})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \leq v \times 2L \times x \tag{8.18}$$

Let us explore two cases.

(a) $C_i > 2L \times x$

We will show that, if this case is true then it contradicts Expression (8.2). Note that τ_i and τ_k share at least one resource and hence it is impossible for them to execute simultaneously. Recall that, Expression (8.2) states that there is a feasible schedule so in this feasible schedule, it must hold that, τ_i and τ_k never execute simultaneously. With reasoning similar to Expression (8.16), we obtain that, for the case of periodically arriving tasks, in a time interval of duration $2T_k$, there is at least one job of task τ_k that has arrived and whose deadline expired. Hence, from Expression (8.2), it follows

that, in a time interval of duration $2T_k$, there is at least one job of task τ_k that has executed entirely. Using Expression (8.13) and the condition of the case gives us that: $C_i > 2T_k$. Hence, during the time when a job of τ_i executes, there is at least one job of τ_k executing. But this is impossible because τ_i and τ_k share resources. Hence, this is a contradiction.

(b) $C_i \leq 2L \times x$

Using the condition of the case on Expression (8.18) and dividing by $2v \times x$ gives us:

$$\frac{C_i}{2 \times v \times x} + \max(0, \lfloor \frac{2L \times x - T_k}{T_k} \rfloor) \times \frac{C_k}{2 \times v \times x} + \sum_{\tau_{i'} \in (S(\tau_k) \setminus \{\tau_i\})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times \frac{C_{i'}}{2 \times v \times x} \leq L \tag{8.19}$$

Combining Expression (8.19) with Expression (8.10) and multiplying by $2v \times x$ and observing that the resulting equation has the same term on both sides and this can be canceled out gives us:

$$\begin{aligned} &\max(0, \lfloor \frac{2L \times x - T_k}{T_k} \rfloor) \times C_k + \sum_{\tau_{i'} \in (S(\tau_k) \setminus \{\tau_i\})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \\ &< \\ &\max(0, \lfloor \frac{L - \frac{T_k}{x} + T_k}{T_k} \rfloor) \times C_k + \sum_{\tau_{i'} \in (S(\tau_k) \setminus \{\tau_i\})} \max(0, \lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \end{aligned} \tag{8.20}$$

Observe that, the left-hand side can be rewritten as a single sum. And also observe that, the right-hand side can be rewritten as a single sum. Rewriting each of the sums as two sums gives us:

$$\begin{aligned} &\sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} + \\ &\sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} > L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \\ &< \\ &\sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} + \\ &\sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} > L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \end{aligned} \tag{8.21}$$

Observing that the last sum is zero and relaxing the second term on the left-hand side

gives us:

$$\sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} < \sum_{\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \quad (8.22)$$

Hence, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\})) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'}) \wedge \Big(\max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} < \max(0, \lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \Big) \quad (8.23)$$

Hence, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\})) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'}) \wedge \Big(2L \times x - T_{i'} < L - \frac{T_{i'}}{x} + T_{i'}\Big) \quad (8.24)$$

Hence, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in ((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\})) \wedge (\frac{T_{i'}}{x} \leq L) \wedge ((2x-1) \times L < (2 - 1/x) \times T_{i'}) \quad (8.25)$$

Hence, there exists a task $\tau_{i'}$ such that

$$\Big(\tau_{i'} \in \big((S(\tau_k) \cup \{\tau_k\}) \setminus \{\tau_i\}\big)\Big) \wedge \Big((2x-1) \times \frac{T_{i'}}{x} < (2 - 1/x) \times T_{i'}\Big) \quad (8.26)$$

This is a contradiction.

2. $|BLT(\tau_k, DMJ, t_2)| = 0$

From the case, we obtain that there is no task in $S(\tau_k)$ such that this task executed at the time when DMJ arrived. We will now discuss the time interval [t_2,t_0). We let L denote the duration of this time interval. Clearly,

$$L = \frac{D_k}{x} \quad (8.27)$$

Using Expression (8.4) on Expression (8.27) yields:

$$L = \frac{T_k}{x} \quad (8.28)$$

During this time interval [t_2,t_0), at each instant, either (i) the set of jobs executing includes a job of task τ_k or (ii) the set of jobs executing includes a job of a task in $S(\tau_k)$.

Since we had a deadline miss, we obtain that:

$$\frac{C_k}{2\times v\times x}+\sum_{\tau_{i'}\in S(\tau_k)}\max(0,\lfloor\frac{L-\frac{D_{i'}}{x}}{T_{i'}}\rfloor+1)\times\frac{C_{i'}}{2\times v\times x}\ >\ L \tag{8.29}$$

Using Expression (8.4) on Expression (8.29) and rewriting yields:

$$\frac{C_k}{2\times v\times x}+\sum_{\tau_{i'}\in S(\tau_k)}\max(0,\lfloor\frac{L-\frac{T_{i'}}{x}+T_{i'}}{T_{i'}}\rfloor)\times\frac{C_{i'}}{2\times v\times x}\ >\ L \tag{8.30}$$

We can discuss the implication of Expression (8.2) being true just like in Case 1 and this gives us:

$$\begin{aligned}\max(0,\lfloor\frac{2L\times x-T_k}{T_k}\rfloor)\times\frac{C_k}{2\times v\times x}+\\ \sum_{\tau_{i'}\in S(\tau_k)}\max(0,\lfloor\frac{2L\times x-T_{i'}}{T_{i'}}\rfloor)\times\frac{C_{i'}}{2\times v\times x}\ \leq\ L\end{aligned} \tag{8.31}$$

Combining Expression (8.31) with Expression (8.30) and multiplying by $2v\times x$ and observing that $\max(0,\lfloor\frac{2L\times x-T_k}{T_k}\rfloor)=\max(0,\lfloor\frac{2T_k-T_k}{T_k}\rfloor)=1$ and rewriting gives us:

$$\sum_{\tau_{i'}\in S(\tau_k)}\max(0,\lfloor\frac{2L\times x-T_{i'}}{T_{i'}}\rfloor)\times C_{i'}\ <\ \sum_{\tau_{i'}\in S(\tau_k)}\max(0,\lfloor\frac{L-\frac{T_{i'}}{x}+T_{i'}}{T_{i'}}\rfloor)\times C_{i'}$$

Rewriting each of the sums as two sums gives us:

$$\begin{aligned}\sum_{\tau_{i'}\in(S(\tau_k))\wedge(T_{i'}\leq L-\frac{T_{i'}}{x}+T_{i'})}\max(0,\lfloor\frac{2L\times x-T_{i'}}{T_{i'}}\rfloor)\times C_{i'}+\\ \sum_{\tau_{i'}\in(S(\tau_k))\wedge(T_{i'}>L-\frac{T_{i'}}{x}+T_{i'})}\max(0,\lfloor\frac{2L\times x-T_{i'}}{T_{i'}}\rfloor)\times C_{i'}\\ <\\ \sum_{\tau_{i'}\in(S(\tau_k))\wedge(T_{i'}\leq L-\frac{T_{i'}}{x}+T_{i'})}\max(0,\lfloor\frac{L-\frac{T_{i'}}{x}+T_{i'}}{T_{i'}}\rfloor)\times C_{i'}+\\ \sum_{\tau_{i'}\in(S(\tau_k))\wedge(T_{i'}>L-\frac{T_{i'}}{x}+T_{i'})}\max(0,\lfloor\frac{L-\frac{T_{i'}}{x}+T_{i'}}{T_{i'}}\rfloor)\times C_{i'}\end{aligned} \tag{8.32}$$

Observing that the last sum is zero and relaxing the second term on the left-hand side gives

us:

$$\sum_{\tau_{i'} \in (S(\tau_k)) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(0, \lfloor \frac{2L \times x - T_{i'}}{T_{i'}} \rfloor) \times C_{i'} < \sum_{\tau_{i'} \in (S(\tau_k)) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'})} \max(\lfloor \frac{L - \frac{T_{i'}}{x} + T_{i'}}{T_{i'}} \rfloor) \times C_{i'} \tag{8.33}$$

Observing Expression (8.33) gives us that, there is at least one term on the left-hand side that is smaller than the corresponding term on the right-hand side. This together with Expression (8.28) give us that, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in S(\tau_k)) \wedge (T_{i'} \leq L - \frac{T_{i'}}{x} + T_{i'}) \wedge (L = \frac{T_k}{x}) \wedge (2L \times x - T_{i'} < L - \frac{T_{i'}}{x} + T_{i'})$$

Hence, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in S(\tau_k)) \ \wedge \ (\frac{T_{i'}}{x} \leq L) \ \wedge \ (L = \frac{T_k}{x}) \ \wedge \ ((2x-1) \times L < (2 - \frac{1}{x}) \times T_{i'}) \tag{8.34}$$

Hence, there exists a task $\tau_{i'}$ such that

$$(\tau_{i'} \in S(\tau_k)) \ \wedge \ \left(\frac{T_{i'}}{x} \leq L\right) \ \wedge \ \left((2x-1) \times \frac{T_{i'}}{x} < (2 - \frac{1}{x}) \times T_{i'}\right) \tag{8.35}$$

This is a contradiction.

Hence, if the lemma is false then we obtain a contradiction. Consequently, the lemma is true. □

Combining the two previous lemmas gives us (below) a relationship between the feasibility on a heterogeneous multiprocessor and the schedulability of ra-np-pEDF algorithm.

Lemma 51. *$\forall \tau$, $\forall \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$, $\forall R$, $\forall x \geq 1$, $v \geq |UNER|$ such that τ is an implicit-deadline sporadic task set and $\forall \tau_i \in \tau : R_i \neq \emptyset$ and $\forall \tau_i \in \tau$ it holds that whenever τ_i executes it holds resource set R_i :*

$$\text{rmig-feas}(\tau, R, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})) \implies$$
$$\text{sched}\Bigg(\text{ra-np-pEDF}, \text{mulCDT}(\texttt{create-fav-taskset}(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})), \frac{1}{2 \times v \times x}, \frac{1}{x}, 1), R, \texttt{create-fav-platform}(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}), v)\Bigg)$$

Proof. Follows from Lemma 49 and Lemma 50. □

Corollary 9. *Consider an implicit-deadline sporadic tasks set that is offline non-preemptive feasible on a single processor. If this task set is scheduled by the* non-preemptive EDF *algorithm on a processor with twice the speed then this task set is schedulable.*

Proof. Follows from specializing Lemma 51 with $v = 1$ and $x = 1$ and a system with a single processor and a single resource and all tasks share this single resource and whenever a task executes it needs to hold this resource. □

8.5.3 Useful results

In this section, we present a previously known (Lemma 52) result and some new results (Lemma 53-56 and Corollary 10) that we use while proving the speed competitive ratio of our algorithm, LP-EE-vpr, in Section 8.5.4.

Lemma 52 states that the speed competitive ratio of algorithm, LP-EE, proposed in [Bar04c] is *two*. The algorithm, LP-EE, non-migratively schedules a set of implicit-deadline sporadic tasks that *do not* share resources on a t-type heterogeneous multiprocessor platform.

Lemma 52 (*From Theorem 3 in [Bar04c]*)**.**

$$\text{nmig-feas}\left(\tau, \Pi\left(m_1, m_2, \ldots, m_t\right)\right) \Rightarrow \text{sched}\left(\text{LP-EE}, \tau, \Pi\left(m_1, m_2, \ldots, m_t\right) \times 2\right)$$

We now show that, if an implicit-deadline sporadic task set τ in which tasks do not share resources is non-migrative-offline schedulable on a t-type heterogeneous multiprocessor platform, $\Pi\left(m_1, m_2, \ldots, m_t\right)$, then the constrained-deadline sporadic task set τ^A (in which tasks do not share resources as well) which is derived from τ (as described in Section 8.4.1) is also non-migrative offline schedulable but on platform $\Pi\left(m_1, m_2, \ldots, m_t\right) \times 2$ (e.g., by non-migrative preemptive EDF). This is shown with the help of a density-based schedulability test by exploiting the fact that, on a processor π_p of type-k, the density $\delta_{i,A}^k$ of a task $\tau_{i,A} \in \tau^A$ is twice the utilization u_i^k of the corresponding task $\tau_i \in \tau$ (see Expression (8.1)). Hence, the density of the task $\tau_{i,A} \in \tau^A$ on a twice faster platform $\Pi\left(m_1, m_2, \ldots, m_t\right) \times 2$ is equal to the utilization of the corresponding task $\tau_i \in \tau$ on platform $\Pi\left(m_1, m_2, \ldots, m_t\right)$.

Lemma 53.

$$\text{nmig-feas}\left(\tau, \Pi\left(m_1, m_2, \ldots, m_t\right)\right) \Rightarrow \text{nmig-feas-}\delta\left(\tau^A, \Pi\left(m_1, m_2, \ldots, m_t\right) \times 2\right)$$

Proof. Suppose that the left-hand side predicate, nmig-feas $\left(\tau, \Pi\left(m_1, m_2, \ldots, m_t\right)\right)$, is true. Then let us arbitrarily choose one set of jobs *JS* generated by the task set τ. Since it holds that nmig-feas $\left(\tau, \Pi\left(m_1, m_2, \ldots, m_t\right)\right)$ is true, there exists a non-migrative-offline schedule for this job set on platform $\Pi\left(m_1, m_2, \ldots, m_t\right)$ in which all the deadlines are met. Since jobs do not migrate and since there is only one phase per job (because there are no resource requests) and since it holds (as stated in Section 8.2) that, all phase-A executions of a given task execute on the same

processor, we can form, from this schedule, a partitioning of the tasks. In this schedule, let $\tau[\pi_p]$ be the set of tasks assigned to processor π_p. This gives us:

$$\forall k,\ \ \forall \pi_p \text{ of type-k} \in \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) : \sum_{\tau_i \in \tau[\pi_p]} u_i^k \leq 1 \tag{8.36}$$

We now show that there must also exist a non-migrative-offline schedule for the derived task set τ^A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2$ in which all the deadlines are met. By definition of τ^A, we know that, for every task $\tau_i \in \tau$, there exists a corresponding task, $\tau_{i,A} \in \tau^A$. Also, from Expression (8.1), we know that, on a processor of type-k, where $k \in \{1, 2, \ldots, t\}$, density $\delta_{i,A}^k$ of task $\tau_{i,A} \in \tau^A$ is twice the utilization u_i^k of the corresponding task, $\tau_i \in \tau$.

Let us assign task set τ^A on platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2$ as follows: if a task, $\tau_i \in \tau$, is assigned to a processor of type-k, say π_p of type-k $\in \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})$, in the non-migrative-offline schedule which meets all deadlines, then we assign its corresponding task, $\tau_{i,A} \in \tau^A$, to the corresponding processor in the faster platform, i.e., to processor π_p of type-k $\in \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2$. From the fact that this assignment of τ^A, which is identical to the assignment of τ, is made on a platform twice faster (on which the densities of tasks will be halved) and from Expressions (8.1) and (8.36), we get:

$$\forall k,\ \ \forall \pi_p \text{ of type-k} \in \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2 : \sum_{\tau_{i,A} \in \tau^A[\pi_p]} \delta_{i,A}^k \leq 1 \tag{8.37}$$

which satisfies density-based schedulability test of non-migrative EDF on a t-type heterogeneous multiprocessor platform. We can repeat this reasoning for any choice of *JS*. Hence, the task set τ^A is non-migrative-offline feasible on the platform $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2$. Hence the lemma. □

Corollary 10.

$$\text{nmig-feas-}\delta\left(\tau^A, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2\right) \Rightarrow \text{nmig-feas}\left(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})\right)$$

Proof. Follows from reasoning analogous to the reasoning for the proof of Lemma 53. □

The following lemma is an extension of Lemma 52 obtained by applying density-based test instead of utilization-based test and on twice faster platforms.

Lemma 54.

$$\text{nmig-feas-}\delta\left(\tau^A, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2\right) \Rightarrow \text{sched}\left(\text{LP-EE-}\delta, \tau^A, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 4\right)$$

Proof. Let us assume that the left-hand side predicate nmig-feas-$\delta\left(\tau^A, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2\right)$ of the claim is true. From Corollary 10, it holds that, the predicate nmig-feas$\left(\tau, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t})\right)$ is also true. Then, from Lemma 52, the predicate sched(LP-EE, τ, $\Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}) \times 2$) must hold true as well. From Expression (8.1), we know that, on a processor of type-k, density $\delta_{i,A}^k$ of every task $\tau_{i,A} \in \tau^A$ is twice the utilization u_i^k of the corresponding task $\tau_i \in \tau$, and hence it must

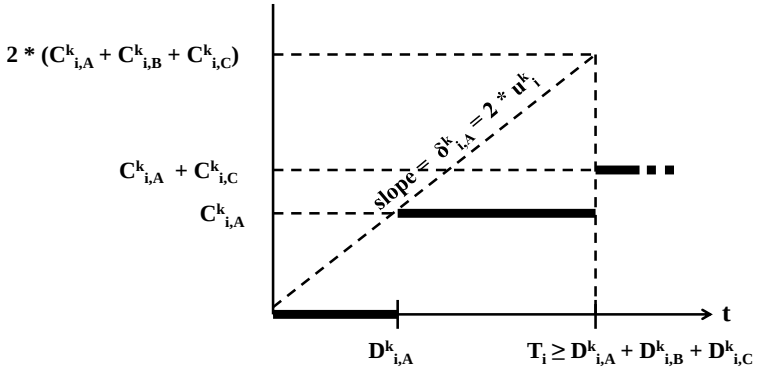


Figure 8.8: Assigning phase-C subtasks to the same virtual processor as the respective phase-A subtasks (earlier assigned using a density-based test) preserves schedulability.

hold that the predicate sched $\big($LP-EE-$\delta, \tau^A, \Pi(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_t) \times 4\big)$ is true as well, from a similar reasoning as used in Lemma 53. Hence the proof. □

The following lemma states that, if tasks from τ^A are preemptive EDF schedulable on a processor π_p of type-k then we can assign the respective phase-C subtasks from τ^C as well onto processor π_p and after this assignment, the entire set of tasks assigned to processor π_p is preemptive EDF schedulable.

Lemma 55. *Let $\tau^A[\pi_p]$ denote the set of phase-A subtasks assigned on processor π_p of type-k. If $\tau^A[\pi_p]$ is preemptive-EDF schedulable, ascertainable with a density-based test on π_p, i.e.,*

$$\delta^k_{\tau^A[\pi_p]} \stackrel{def}{=} \sum_{\tau_{i,A} \in \tau^A[\pi_p]} \delta^k_{i,A} \leq 1$$

then $\tau^A[\pi_p] \cup \tau^C[\pi_p]$ (where $\tau^C[\pi_p]$ is the set of respective phase-C subtasks whose arrivals have fixed offset from the arrival of respective phase-A subtasks) is preemptive-EDF schedulable on processor π_p of type-k.

Proof. We know that the task set $\tau^A[\pi_p]$ is preemptive-EDF schedulable, ascertainable with a density-based test, on processor π_p of type-k, i.e., $\delta^k_{\tau^A[\pi_p]} \leq 1$. To show that $\tau^A[\pi_p] \cup \tau^C[\pi_p]$ is schedulable on processor π_p, it is sufficient to show that the *demand-bound function*[5], DBF$(\tau^A[\pi_p] \cup \tau^C[\pi_p], t)$, of task set $\tau^A[\pi_p] \cup \tau^C[\pi_p]$, never exceeds $\delta^k_{\tau^A[\pi_p]} \times t$ at any instant t [BMR90].

The following holds for every phase-A subtask, $\tau_{i,A} \in \tau^A$, and respective phase-C subtask, $\tau_{i,C} \in \tau^C$:

$$\mathrm{DBF}(\{\tau_{i,A}\} \cup \{\tau_{i,C}\}, t) \leq t \times \delta^k_{i,A} = t \times \frac{C^k_{i,A}}{D^k_{i,A}} \tag{8.38}$$

[5]The *demand bound function* of a *task* τ_i, dbf(τ_i, t), is the maximum possible execution demand by jobs of τ_i, that have both arrival and deadline within any interval of length t. The demand bound function of a *task set* τ is defined as: DBF$(\tau, t) = \sum_{\tau_i \in \tau}$ dbf(τ_i, t) [BMR90].

This can be verified from Figure 8.8 since the maximum "slope" to any point in the graph of $\text{DBF}(\{\tau_{i,A}\} \cup \{\tau_{i,C}\}, t)$ from the origin is $\delta^k_{i,A} = \frac{C^k_{i,A}}{D^k_{i,A}}$ (which is equal to $2 \times u^k_i$ of $\tau_i \in \tau$, as per our choice of $D^k_{i,A}$), at abscissa $t = D^k_{i,A}$. Summing Expression (8.38) for all the subtasks $\tau_{i,A} \in \tau^A[\pi_p]$ and the corresponding subtasks $\tau_{i,C} \in \tau^C[\pi_p]$ yields:

$$\text{DBF}(\tau^A[\pi_p] \cup \tau^C[\pi_p], t) \leq t \times \sum_{\tau_{i,A} \in \tau^A[\pi_p]} \delta^k_{i,A} = t \times \delta^k_{\tau^A[\pi_p]}$$

Hence the proof. □

We will now prove a guarantee on the schedulability of ra-np-pEDF-fav.

Lemma 56. *Let τ denote an implicit-deadline sporadic task set. Let R denote the set of resources in the system. Let P_j denote one resource request partition of R and let $\mathbb{R}(\mathrm{P_j})$ denote the resources belonging to this resource request partition.*

$$\begin{aligned} &\text{rmig-feas}(\tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})) \Rightarrow \\ &\text{sched}\left(\text{ra-np-pEDF-fav}, \tau^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times 4 \times |P_j|\right) \end{aligned} \tag{8.39}$$

Proof. Let τ' denote the subset of tasks in τ that request a resource set in P_j. Let τ'' denote a set of tasks derived from τ' but where a task in τ'' does not perform any execution before requesting a resource set and a task in τ'' does not perform any execution after releasing a resource set.

Then consider the three claims below:

1. $\text{rmig-feas}(\tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})) \Rightarrow \text{rmig-feas}(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}))$
2. $\text{rmig-feas}(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})) \Rightarrow$
   $\text{sched}\left(\text{ra-np-pEDF-fav}, \tau''^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times 4 \times |P_j|\right)$
3. $\tau''^{B,\mathbb{R}(\mathrm{P_j})} = \tau^{B,\mathbb{R}(\mathrm{P_j})}$

If we can prove these three claims then the correctness of the lemma follows. Hence, we prove the claims below.

Proving 1. This claim follows from the fact that the feasibility cannot be violated by only considering a subset of the tasks and by only considering a subset of the resources and by only considering some of the execution of a task.

Proving 2. Applying Lemma 51 with the task set τ'' and the resource set $\mathbb{R}(\mathrm{P_j})$ and with $x = 2$ and $v = |P_j|$ yields:

$$\begin{aligned} &\text{rmig-feas}\left(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right) \Longrightarrow \\ &\text{sched}\Bigg(\text{ra-np-pEDF}, \text{mulCDT}(\texttt{create-fav-taskset}\left(\tau'', \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right), \\ &1, \frac{1}{2}, 1), \mathbb{R}(\mathrm{P_j}), \texttt{create-fav-platform}\left(\tau'', \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}), v\right) \times 4 \times |P_j|\Bigg) \end{aligned} \tag{8.40}$$

The order in which the functions mulCDT and `create-fav-taskset` are applied can be changed without affecting the result. And the result of the `create-fav-platform` function when taken τ'' as input is the same as when taken mulCDT$(\tau'', 1, \frac{1}{2}, 1)$ as input. This gives us:

$$\begin{aligned}&\text{rmig-feas}\left(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right) \Longrightarrow \\ &\text{sched}\left(\text{ra-np-pEDF}, \texttt{create-fav-taskset}\left(\text{mulCDT}(\tau'', 1, \frac{1}{2}, 1), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right),\right. \\ &\left.\mathbb{R}(\mathrm{P_j}), \texttt{create-fav-platform}\left(\text{mulCDT}(\tau'', 1, \frac{1}{2}, 1), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}), v\right) \times 4 \times |P_j|\right)\end{aligned} \tag{8.41}$$

Observing that mulCDT$(\tau'', 1, \frac{1}{2}, 1) = \tau''^{B,\mathbb{R}(\mathrm{P_j})}$ gives us:

$$\begin{aligned}&\text{rmig-feas}\left(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right) \Longrightarrow \\ &\text{sched}\left(\text{ra-np-pEDF}, \texttt{create-fav-taskset}\left(\tau''^{B,\mathbb{R}(\mathrm{P_j})}, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right), \mathbb{R}(\mathrm{P_j}),\right. \\ &\left.\texttt{create-fav-platform}\left(\tau''^{B,\mathbb{R}(\mathrm{P_j})}, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}), v\right) \times 4 \times |P_j|\right)\end{aligned} \tag{8.42}$$

Observe that the schedule generated by the ra-np-pEDF scheduling policy of tasks in the task set `create-fav-taskset` $\left(\tau''^{B,\mathbb{R}(\mathrm{P_j})}, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right)$ on processors in the computing platform `create-fav-platform` $\left(\tau''^{B,\mathbb{R}(\mathrm{P_j})}, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}), v\right)$ is identical to the schedule generated by ra-np-pEDF-fav scheduling of tasks in $\tau''^{B,\mathbb{R}(\mathrm{P_j})}$ on $\Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right)$. Combining this observation with (8.42) gives us:

$$\begin{aligned}&\text{rmig-feas}\left(\tau'', \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right) \Longrightarrow \\ &\text{sched}\left(\text{ra-np-pEDF-fav}, \tau''^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times 4 \times |P_j|\right)\end{aligned} \tag{8.43}$$

This states the Claim 2.

Proving 3. The correctness of this claim ($\tau''^{B,\mathbb{R}(\mathrm{P_j})} = \tau^{B,\mathbb{R}(\mathrm{P_j})}$) can be seen directly from the definition of $\tau''^{B,\mathbb{R}(\mathrm{P_j})}$.

Hence the lemma. □

8.5.4 The speed competitive ratio of LP-EE-vpr algorithm

We now prove the speed competitive ratio of the LP-EE-vpr algorithm.

Theorem 23. *The* LP-EE-vpr *algorithm has the following speed competitive ratio:* $4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{\min\{m_1, m_2, \ldots, m_t\}} \right\rceil\right)$.

Proof. We prove the claim by considering the scheduling of tasks in each of the three phases independently and then merging the results from these three scenarios.

Consider phase-A scheduling. Combining Lemma 53 and Lemma 54, yields:

$$\text{rmig-feas}\left(\tau, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right)\right) \Rightarrow \text{sched}\left(\text{LP-EE-}\delta, \tau^A, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right) \times 4\right) \tag{8.44}$$

Consider phase-C scheduling. Note that LP-EE-vpr assigns a phase-C subtask, $\tau_{i,C} \in \tau^C$, to the same VP_{AC} virtual processor to which the corresponding phase-A subtask, $\tau_{i,A} \in \tau^A$, is assigned (see line 10 in Algorithm 16). For convenience, let LP-EE-δ-cp denote such a task assignment policy, i.e., using LP-EE-δ to assign phase-A subtasks and 'copying' the assignment for respective phase-C subtasks. Lemma 55 showed that such an assignment preserves schedulability of the relevant tasks. From Lemma 55 and Expression (8.44), we get:

$$\text{rmig-feas}\left(\tau, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right)\right) \Rightarrow \text{sched}\left(\text{LP-EE-}\delta\text{-cp}, \tau^A \cup \tau^C, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right) \times 4\right) \tag{8.45}$$

Now let us discuss phase-B scheduling. From Lemma 56 we obtain:

$$\begin{aligned} &\forall P_j \in P : \text{rmig-feas}\left(\tau, R, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right)\right) \Longrightarrow \\ &\text{sched}\left(\text{ra-np-pEDF-fav}, \mathbb{R}(\mathrm{P_j}), \tau^{B,\mathbb{R}(\mathrm{P_j})}, P_j, \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times 4 \times |P_j|\right) \end{aligned} \tag{8.46}$$

We know that, $\text{MAXP} = \max_{P_j \in P}\left|P_j\right|$. Using this, rewriting Expression (8.46) yields:

$$\begin{aligned} &\forall P_j \in P : \text{rmig-feas}\left(\tau, \mathbb{R}(\mathrm{P_j}), \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right)\right) \Longrightarrow \\ &\text{sched}\left(\text{ra-np-pEDF-fav}, \tau^{B,\mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times \text{MAXP} \times 4\right) \end{aligned} \tag{8.47}$$

Let us now combine the results obtained for task sets $\tau^A \cup \tau^C$ and $\tau^{B,\mathbb{R}(\mathrm{P_j})}$. Dividing the type-k ($\forall k : k \in \{1, 2, \ldots, t\}$) processor speeds in Expression (8.45) by $4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil\right)$, we get:

$$\begin{aligned} &\text{rmig-feas}\left(\tau, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right) \times \left\langle \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_1} \right\rceil\right)}, \ldots, \right.\right. \\ &\qquad \left.\left. \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_t} \right\rceil\right)} \right\rangle\right) \Longrightarrow \\ &\text{sched}\left(\text{LP-EE-}\delta\text{-cp}, \tau^A \cup \tau^C, \Pi\left(\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_\mathrm{t}\right) \times \right. \\ &\qquad \left. \left\langle \frac{1}{1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_1} \right\rceil}, \ldots, \frac{1}{1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_t} \right\rceil} \right\rangle\right) \end{aligned} \tag{8.48}$$

Dividing the type-k ($\forall k : k \in \{1, 2, \ldots, t\}$) processor speeds in Expression (8.47) by a factor of

$4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_k} \right\rceil\right)$, we get:

$$\forall P_j \in P : \text{rmig-feas}\left(\tau, \mathbb{R}(\mathrm{P_j}), \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}) \times \left\langle \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_1} \right\rceil\right)}, \ldots, \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_t} \right\rceil\right)} \right\rangle\right) \Longrightarrow$$
$$\text{sched}\left(\text{ra-np-pEDF-fav}, \tau^{B, \mathbb{R}(\mathrm{P_j})}, \mathbb{R}(\mathrm{P_j}), \Pi\left(|\mathrm{P_j}|, |\mathrm{P_j}|, \ldots, |\mathrm{P_j}|\right) \times \left\langle \frac{\text{MAXP}}{1 + \text{MAXP} \times \left\lceil \frac{|P|}{m_1} \right\rceil}, \ldots, \frac{\text{MAXP}}{1 + \text{MAXP} \times \left\lceil \frac{|P|}{m_t} \right\rceil} \right\rangle\right) \tag{8.49}$$

The specifications of the processors in the right-hand side predicates of Expression (8.48) and Expression (8.49) match those of the virtual processors that LP-EE-vpr created (see Section 8.4.2). Recall that LP-EE-vpr assigned phase-A and phase-C subtasks to $\mathrm{VP_{AC}}$ virtual processors and phase-B subtasks to $\mathrm{VP_B}$ virtual processors. Hence, combining Expression (8.48) and $|P|$ instances of Expression (8.49), yields:

$$\text{rmig-feas}\left(\tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}) \times \left\langle \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_1} \right\rceil\right)}, \ldots, \frac{1}{4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{m_t} \right\rceil\right)} \right\rangle\right) \Longrightarrow$$
$$\text{sched}\left(\text{LP-EE-vpr}, \tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t})\right) \tag{8.50}$$

We know that higher speed processors do not jeopardize the feasibility of a task set. Hence, we can write:

$$\text{rmig-feas}\left(\tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}) \times \langle \min\{s_1, s_2, \ldots, s_t\}, \ldots, \min\{s_1, s_2, \ldots, s_t\} \rangle\right) \Rightarrow$$
$$\text{rmig-feas}\left(\text{rmo}, \tau, R, \Pi(\mathrm{m_1}, \mathrm{m_2}, \ldots, \mathrm{m_t}) \times \langle s_1, s_2, \ldots, s_t \rangle\right)$$

Substituting $s_k = \frac{1}{4\times\left(1+\text{MAXP}\times\left\lceil\frac{|P|\times\text{MAXP}}{m_k}\right\rceil\right)}$, $\forall k : k \in \{1,2,\ldots,t\}$, in the above expression and combining with Expression (8.50) and rewriting gives:

$$\begin{aligned}
&\text{rmig-feas}\left(\tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\times\right.\\
&\left\langle\frac{1}{4\times\left(1+\text{MAXP}\times\max\left\{\left\lceil\frac{|P|\times\text{MAXP}}{m_1}\right\rceil,\ldots,\left\lceil\frac{|P|\times\text{MAXP}}{m_t}\right\rceil\right\}\right)},\right.\\
&\left.\left.\ldots,\frac{1}{4\times\left(1+\text{MAXP}\times\max\left\{\left\lceil\frac{|P|\times\text{MAXP}}{m_1}\right\rceil,\ldots,\left\lceil\frac{|P|\times\text{MAXP}}{m_t}\right\rceil\right\}\right)}\right\rangle\right)\Longrightarrow\\
&\text{sched}\left(\text{LP-EE-vpr}, \tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\right)
\end{aligned} \tag{8.51}$$

Let us multiply the speeds of all the processors in Expression (8.51) by the following factor: $4\times\left(1+\text{MAXP}\times\max\left\{\left\lceil\frac{|P|\times\text{MAXP}}{m_1}\right\rceil,\ldots,\left\lceil\frac{|P|\times\text{MAXP}}{m_t}\right\rceil\right\}\right)$. This gives us:

$$\begin{aligned}
&\text{rmig-feas}\left(\tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\right)\Longrightarrow\\
&\text{sched}\left(\text{LP-EE-vpr}, \tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\times\right.\\
&\left\langle 4\times\left(1+\text{MAXP}\times\max\left\{\left\lceil\frac{|P|\times\text{MAXP}}{m_1}\right\rceil,\ldots,\left\lceil\frac{|P|\times\text{MAXP}}{m_t}\right\rceil\right\}\right),\ldots,\right.\\
&\left.\left.4\times\left(1+\text{MAXP}\times\max\left\{\left\lceil\frac{|P|\times\text{MAXP}}{m_1}\right\rceil,\ldots,\left\lceil\frac{|P|\times\text{MAXP}}{m_t}\right\rceil\right\}\right)\right\rangle\right)
\end{aligned}$$

By rewriting the right-hand side predicate of the above expression, we get:

$$\begin{aligned}
&\text{rmig-feas}\left(\tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\right)\Longrightarrow\\
&\text{sched}\left(\text{LP-EE-vpr}, \tau, R, \Pi(\mathrm{m}_1,\mathrm{m}_2,\ldots,\mathrm{m}_\mathrm{t})\times\right.\\
&\quad\left\langle 4\times\left(1+\text{MAXP}\times\left\lceil\frac{|P|\times\text{MAXP}}{\min\{m_1,m_2,\ldots,m_t\}}\right\rceil\right),\ldots,\right.\\
&\quad\left.\left.4\times\left(1+\text{MAXP}\times\left\lceil\frac{|P|\times\text{MAXP}}{\min\{m_1,m_2,\ldots,m_t\}}\right\rceil\right)\right\rangle\right)
\end{aligned}$$

Hence the theorem. □

Theorem 24. *Consider the case in which each task can request at most one resource, i.e,* $\forall\tau_i \in \tau : |R_i| \leq 1$. *For this case,* LP-EE-vpr *has a speed competitive ratio of* $4\times\left(1+\left\lceil\frac{|R|}{\min\{m_1,m_2,\ldots,m_t\}}\right\rceil\right)$.

Proof. If $\forall\tau_i \in \tau : |R_i| \leq 1$ then every connected component in the graph has one vertex and hence every resource request partition has one element. Thus, MAXP $= 1$. Also, the number of resource request partitions $|P|$ is no greater than $|R|$, i.e., $|P| \leq |R|$. Applying this on Theorem 23 gives us the theorem. □

8.6 Discussion

In this section, we briefly discuss run-time mechanisms for realizing virtual processors and the preemptions generated and also highlight a couple of useful properties of LP-EE-vpr algorithm such as deadlock-free property, nested resource access and the bound on number of migrations per job. Also, a couple of tricks to improve the performance of LP-EE-vpr algorithm are discussed as well.

8.6.1 Run-time mechanism for realizing virtual processors and the preemptions generated

Given that the research literature has been lacking a scheduling algorithm for heterogeneous multiprocessors with resource sharing such that the algorithm has a proven speed competitive ratio, our focus in this chapter has been to create one. We did not deal with the cost of preemption.

Assuming that there is no cost of a preemption, one can create a set of virtual processors from a single physical processor without losing capacity as follows. Choose a timeslot size (denoted as S) and subdivide time into time intervals, each being of duration equal to the timeslot size S. Then if we want to create a set $\mathrm{VP} = \left\{\mathrm{vp}_1, \mathrm{vp}_2, ..., \mathrm{vp}_{|\mathrm{VP}|}\right\}$ of virtual processors where virtual processor vp_l (where $l \in \{1,2,\ldots,|\mathrm{VP}|\}$) has speed SP_l and accomplish this as long as $\sum_{l\in\{1,2,...,|\mathrm{VP}|\}} \mathrm{SP}_l \leq 1$, then this can be done as follows. Create a reserve for vp_l in the timeslot so that this reserve has the duration $S \times \mathrm{vp}_l$ and let the time of this reserve supply time to the virtual processor vp_l. Then let S be arbitrarily small. This gives us the desired virtual processors and this is the idea we have assumed in this paper.

Unfortunately, this approach generates an infinite number of preemptions. One could generate virtual processors in two other ways. First, by choosing S being the greatest common denominator of the parameters of the subtasks, one can still form virtual processors as mentioned above and still utilize 100% of the capacity of a physical processor [AB08]. This approach has two problems (i) the greatest common divisor of the parameters of the subtasks may not exist (this is an issue for the case that parameters are not rational numbers) and (ii) even if the greatest common divisor of the parameters of the subtasks exists, it may still be very small and hence may generate a very large number of preemptions. A second way to choose S (which avoids this drawback) is to choose a positive integer δ and then choose S as the minimum of all parameters of subtasks divided by δ. This approach has been used for creating virtual processors in [AB08, BA09] so that as long as the sum of the speeds of the virtual processors desired to be formed does not exceed a given bound $\mathrm{UB}(\delta)$ (higher than 60% but lower than 100%), which is a function of δ, then all virtual processors can be formed. We can use such approaches at the cost of having a speed competitive ratio being multiplied by $1/\mathrm{UB}(\delta)$.

8.6.2 Bound on the number of migrations per job

The algorithm, LP-EE-vpr, by design, limits the number of migrations per job to at most *two*. Recall that, LP-EE-vpr assigns both phase-A and phase-C executions of a task τ_i to the same VP_{AC} virtual processor and phase-B of that task to another VP_{B} virtual processor. Since the algorithm creates the virtual processors in such a manner that the capacity of no virtual processor comes from more than one physical processor (Lemma 47 in Section 8.4.2), it is clear that both phase-A and phase-C of a task are assigned to the same physical processor. Since the virtual processor in VP_{B} to which phase-B of task τ_i is assigned may come from a different physical processor, migration of a job of task τ_i can only occur at time instants when the job requests or releases the resource set R_i. Thus, the algorithm limits the number of migrations per job to at most two.

8.6.3 Nested resource access

To enable our algorithm for handling tasks with nested resource access, one of the two below mentioned techniques can be used.

- **Group locking.** It is a previously known technique [BLBA07] in which the inner locks of a nested resource access are removed and only an outer lock (referred to as a *group lock*) is retained. The following example illustrates how nested resource access can be handled with the help of group locks. Consider a nested resource access in which jobs of a task τ_i request and release the resources in the following order: Each job of task τ_i does the following (in order):

  $$\begin{aligned}&\text{request}(r_1)\\&\text{request}(r_2)\\&\text{release}(r_2)\\&\text{request}(r_3)\\&\text{release}(r_3)\\&\text{release}(r_1)\end{aligned}$$

  With group locking, a new lock would be created, say r_{123} and then task τ_i would be changed such that each job of τ_i now does the following (in order):

  $$\begin{aligned}&\text{request}(r_{123})\\&\text{release}(r_{123})\end{aligned}$$

  If there is any other task that requests one or more of these resources (i.e., resource r_1, r_2 and r_3) then these tasks need to be changed as well.

- **A variant of group locking.** Another way to handle nested resource access is to request all the resources in the nested block at the beginning of the nested block and release all the

resources at the end of this block. With this technique, in the above example, task τ_i would be changed such that each job of task τ_i now does the following (in order):

$$\text{request}(r_1 \text{ and } r_2 \text{ and } r_3)$$
$$\text{release}(r_1 \text{ and } r_2 \text{ and } r_3)$$

Since we allow multiple resources to be requested simultaneously, we can use any of the above two techniques for handling tasks with nested resource access.

8.6.4 Deadlock free property

Partial allocation describes a situation where a task is "waiting" for additional resource(s) while "holding" previously acquired one(s). Partial allocation is a necessary condition for deadlock to occur — see Chapter 7 in [SGG09]. Recall that, we assume (as mentioned in Section 8.2) that a job of task τ_i performs a single request for the resource set R_i and then releases all the resources in the resource set R_i at once. And hence with this assumption, partial allocation never happens. And consequently, the algorithm LP-EE-vpr, for the assumptions stated in Section 8.2, cannot enter a deadlocked state.

8.6.5 Performance improvement

In this section, we describe a couple of tricks to improve the performance of the algorithm.

First, we dimensioned the phase-B virtual processors without considering the parameters of the subtasks that will execute on this virtual processor. A possible way to increase the performance of our algorithm though would be to determine, for each resource request partition, what is the lowest speed that is needed in order for the subtasks requesting the resources from the corresponding resource partition to be ra-np-pEDF-fav schedulable.

Second, our algorithm is based on LP-EE algorithm [Bar04c] for assigning phase-A and phase-C subtasks. We selected LP-EE because it is simple to implement and easy to explain and it has a proven speed competitive ratio. Unfortunately, this algorithm has a time-complexity that is exponential with the number of processors. But we can replace LP-EE with another algorithm that is proposed in [Bar04b]. This algorithm has the same speed competitive ratio as LP-EE but runs with polynomial time-complexity because it does not perform exhaustive enumeration. In addition, one could replace LP-EE with the task assignment algorithm in [WBB13] (which has a better speed competitive ratio than LP-EE). Then we would have a scheduling algorithm for our problem (with resource sharing), with a better speed competitive ratio but at the expense of having a time-complexity that is a polynomial of very high degree.

8.7 Conclusions

In many computer systems, apart from processors, tasks also share resources such as data structures, sensors, etc. and tasks must operate on such resources in a mutually exclusive manner while

accessing the resource. Scheduling real-time tasks that share resources on a heterogeneous multiprocessor platform is a complex problem. In this work, we took the first step to solve the issue via a scheduling algorithm with a proven speed competitive ratio for heterogeneous multiprocessors.

This work considered the problem of scheduling a task set of implicit-deadline sporadic tasks to meet all deadlines on a t-type heterogeneous multiprocessor platform where tasks may share multiple resources. The tasks must operate on such resources in a mutually exclusive manner while accessing the resource, that is, at all times, when a job of a task holds a resource, no other job of any task can hold that resource. Each job may request (a subset of) resources at most once during its execution and it has to request all the resources in the subset together. A job is allowed to migrate when it requests/releases the resources but a job is not allowed to migrate at other times.

We presented an algorithm, LP-EE-vpr, and proved its speed competitive ratio. Specifically, we proved that, if an implicit-deadline sporadic task set is schedulable to meet all deadlines on a t-type heterogeneous multiprocessor platform by an optimal scheduling algorithm that allows a job to migrate only when it requests or releases a resource set, then our algorithm succeeds to meet all deadlines as well with the same restriction on job migration but given processors $4 \times \left(1 + \text{MAXP} \times \left\lceil \frac{|P| \times \text{MAXP}}{\min\{m_1, m_2, \ldots, m_t\}} \right\rceil \right)$ times as fast. For the special case that each task requests at most one resource, the bound of LP-EE-vpr collapses to $4 \times \left(1 + \left\lceil \frac{|R|}{\min\{m_1, m_2, \ldots, m_t\}} \right\rceil \right)$. To the best of our knowledge, LP-EE-vpr is the first algorithm for real-time scheduling of sporadic tasks with resource sharing on t-type heterogeneous multiprocessors that has a provably good performance .

Further, as a by-product of the above result, for the problem of non-preemptive scheduling of tasks on a uniprocessor, we *improved* the previously known [AE10] speed competitive ratio of non-preemptive-EDF algorithm from *three* to *two*.

Part IV

Conclusions

Chapter 9

Conclusions, Discussions and Future Directions

A real-time software system is often modeled as a set of sporadic tasks where each task generates a (potentially infinite) sequence of jobs. Each job of a task may arrive at any time once a minimum inter-arrival time has elapsed since the arrival of the previous job of the same task. Each job has an execution time and a deadline within which it has to complete its execution. Many real-time systems can be effectively modeled using implicit-deadline sporadic tasks in which it is assumed that, for each task, its deadline is equal to its minimum inter-arrival time.

With the emergence of multicores, there is a strong interest in understanding the challenges involved in deploying embedded real-time systems on such computing platforms. Many multicore computing platforms as of today are heterogeneous in nature. The heterogeneous multiprocessor computational model (i.e., unrelated parallel machines) is more generic than identical or uniform multiprocessors, in terms of the systems that it can accommodate. Generally, this called for algorithms with a provably good performance for scheduling real-time workload on heterogeneous multiprocessors. We partially solve the issue by proposing several task assignment algorithms which in turn enable the scheduling of real-time workload on heterogeneous multiprocessors consisting of a constant number (denoted by $t \geq 2$) of distinct processor types. In a *t-type heterogeneous* multiprocessor (i) not all processors are of the same type, (ii) the execution time of a task depends on the type of processor on which it executes and (iii) the number of distinct *types* of processors is a constant and is given by $t \geq 2$. A *two-type heterogeneous* multiprocessor is a special case of t-type in which it holds that $t = 2$. These models are of great practical interest, as they capture many current/future single-chip heterogeneous multiprocessors [AMD13a, App13, Int13c, Int13b, Nvi12, Qua13, Sam13a, ST 12, Tex13, Jon, Int13a].

Specifically, this dissertation considered the following problems for implicit-deadline sporadic tasks on both two-type and t-type heterogeneous multiprocessors:

P1. *design efficient algorithms to assign tasks to individual processors (non-migrative task assignment) such that for every valid job arrival pattern "there exists" a schedule that meets all deadlines.*

P2. *design efficient algorithms to assign tasks to processor types (intra-migrative task assignment) such that for every valid job arrival pattern "there exists" a schedule that meets all deadlines.*

In many computing systems, apart from sharing a processor, tasks also share other resources such as data structures, sensors, etc. and tasks must operate on such resources in a mutually exclusive manner while accessing the resource, that is, at all times, when a job of a task holds a resource, no other job of any task can hold that resource. Hence, this dissertation also considered the following problem for implicit-deadline sporadic tasks on both two-type and t-type heterogeneous multiprocessors:

P3. *design efficient algorithms to assign tasks that share resources to processors such that for every valid job arrival pattern "there exists" a schedule that meets all deadlines.*

Unfortunately, all the three problems are hard to solve, in the sense that, it is not possible to design an optimal algorithm with polynomial time-complexity unless P = NP. Hence, this work proposed several (non-optimal) task assignment algorithms with polynomial time-complexity and proved their performance in terms of *speed competitive ratio*. The speed competitive ratio $\text{SCR}_{\mathscr{A}}$ of an algorithm $\mathscr{A}$ is informally defined as follows. If an optimal algorithm can find a solution for the problem (P1 or P2 or P3) then the algorithm $\mathscr{A}$ also succeeds to find such a solution but given $\text{SCR}_{\mathscr{A}}$ times faster processors compared to the processors used by the optimal algorithm. A summary of the contributions of this research is presented next.

9.1 Summary of results

9.1.1 Intra-migrative task assignment algorithms

This work proposed intra-migrative task assignment algorithms both for two-type heterogeneous multiprocessors as well as generic t-type heterogeneous multiprocessors. In Chapter 3, we proposed a low-degree polynomial time-complexity intra-migrative task assignment algorithm for two-type heterogeneous multiprocessors and showed that the speed competitive ratio of this algorithm is $1 + \alpha/2 \leq 1.5$ against an equally powerful intra-migrative adversary where the parameter $0 < \alpha \leq 1$ is the property of the task set; it is the maximum of all the task utilizations that are no greater than one. In Chapter 6, we designed a polynomial time-complexity intra-migrative task assignment algorithm for t-type heterogeneous multiprocessors and showed that the speed competitive ratio of this algorithm is $1 + \alpha \times \frac{t-1}{t}$ against an equally powerful intra-migrative adversary where the parameter, $t \geq 2$, denotes the number of distinct processor types in the platform.

9.1.2 Non-migrative task assignment algorithms

This work also proposed several non-migrative task assignment algorithms for two-type heterogeneous multiprocessors and also an algorithm for the generic t-type heterogeneous multiprocessors.

In Chapter 4, for *two-type* heterogeneous multiprocessors, we proposed (i) a low-degree polynomial time-complexity non-migrative task assignment algorithm and showed that the speed competitive ratio of this algorithms is $1+\alpha \leq 2$ against an equally powerful non-migrative adversary, where the parameter $0 < \alpha \leq 1$ is the maximum of all the task utilizations that are no greater than one; (ii) a low-degree polynomial time-complexity non-migrative task assignment algorithm and showed that the speed competitive ratio of this algorithms is $1+\alpha \leq 2$ against a more powerful intra-migrative adversary; (iii) a polynomial-time complexity non-migrative task assignment algorithm and showed that its speed competitive ratio is 1.5 and in addition it needs 3 extra processors against an equally powerful non-migrative adversary and finally (iv) a polynomial time approximation scheme with a speed competitive ratio of $1+3\varepsilon$ against an equally powerful non-migrative adversary where $\varepsilon > 0$ is an input parameter. In Chapter 7, for *t-type* heterogeneous multiprocessors, we proposed a non-migrative task assignment algorithm of polynomial time-complexity and showed that the speed competitive ratio of this algorithm is $1+\alpha \leq 2$ against an equally powerful intra-migrative adversary.

9.1.3 Resource sharing algorithms

Further, this work also proposed task assignment algorithms for resource sharing problem (in which tasks are allowed to migrate only when they request or release the resource) both for two-type and the generic t-type heterogeneous multiprocessors. In Chapter 5, for *two-type* heterogeneous multiprocessors, we proposed a low-degree polynomial time-complexity algorithm with a speed competitive ratio of $4+6\cdot\left\lceil\frac{|R|}{\min(m_1,m_2)}\right\rceil$ against an equally powerful adversary (which also permit a task to migrate only when it requests or releases a resource), where $|R|$ denotes the number of shared resources and m_1 (resp., m_2) denotes the number of processors of type-1 (resp., type-2). In Chapter 8, for *t-type* heterogeneous multiprocessors, we proposed a polynomial time-complexity algorithm with a speed competitive ratio of $4\times\left(1+\text{MAXP}\times\left\lceil\frac{|P|\times\text{MAXP}}{\min\{m_1,m_2,\ldots,m_t\}}\right\rceil\right)$ against an equally powerful adversary[1].

We describe the implication of these results in the next section.

9.2 Implication of the results

9.2.1 Bin-packing heuristics for heterogeneous multiprocessors

Bin-packing heuristics are popular for assigning tasks on identical and uniform multiprocessors because they are easy to implement, run fast and offer finite speed competitive ratio. Yet, straightforward application of bin-packing heuristics on heterogeneous multiprocessors perform poorly. Hence, before this research, bin-packing heuristics were not considered for assigning tasks to processors on heterogeneous multiprocessors. Only Integer Linear Programming (ILP) modeling, Linear Programming (LP) relaxation approaches for ILP and dynamic programming techniques were known to perform well on heterogeneous multiprocessors. As part of this work, we observed

[1]The terms P and MAXP in the speed competitive ratio are defined in Chapter 8

that the cause of low performance of bin-packing heuristics on two-type heterogeneous multiprocessors is that, by considering tasks one by one, they lack a "global view" of the problem and thus may assign a task to a processor where it executes slowly. Then we showed how to address this "lack of global view" issue while designing task assignment algorithms for heterogeneous multiprocessors. Overall, this work showed how bin-packing heuristics can be used (by providing a global view) to achieve a provably good performance for task assignment algorithms on two-type heterogeneous multiprocessors.

9.2.2 Cutting planes technique for heterogeneous multiprocessors

Although task assignment schemes with provably good performance have previously been developed by relaxing an Integer Linear Program to a Linear Program and cutting planes have been used to solve Integer Linear Program in different efforts, no work in the past had shown how cutting planes can be used to improve the performance of provably good algorithms for assigning real-time tasks to processors. This work showed that cutting planes can be used to design provably good task assignment algorithms with improved speed competitive ratio on two-type heterogeneous multiprocessors.

9.2.3 Low-degree polynomial time-complexity algorithms for heterogeneous multiprocessors

The special structure of two-type heterogeneous multiprocessors enables the designers to design task assignment algorithms with a low-degree polynomial time-complexity which is otherwise not possible (for the generic t-type heterogeneous multiprocessors, as of today). Prior to this work, approaches to solve the task assignment problem on heterogeneous multiprocessors relied on solving a linear programming formulation and/or dynamic programming techniques. Both these techniques do not facilitate achieving a low-degree polynomial time-complexity. This work showed that, for two-type heterogeneous multiprocessors, low-degree polynomial time-complexity can be achieved by relating the task assignment problem on two-type heterogeneous multiprocessors to fractional knapsack problem and by using bin-packing heuristics.

The next section lists some of the problems that could be explored in the heterogeneous multiprocessor scheduling topic.

9.3 Future directions

With a larger goal of deploying real-time systems workload on heterogeneous multiprocessor computing platforms, in this work, we studied the problem of scheduling real-time tasks on heterogeneous multiprocessors. In particular, we looked at the problem of assigning tasks to individual processors (to processor types, respectively) before run-time such that all the deadlines are met upon scheduling these tasks on each processor (processor type, respectively) using an optimal

uniprocessor (identical multiprocessor, respectively) scheduling algorithm at run-time. In doing so, we made a couple of assumptions about the workload and the computing platform.

These assumptions abstracted out the unnecessary details such as the architectural features of the computing platform (for example, shared hardware resources such as cache, memory, system bus, etc.), complex deadlines of the workload (for example, workloads in which deadline of a task is not equal to its minimum inter-arrival time), etc. and thereby served a couple of purposes. First, these assumptions were helpful in understanding and addressing the fundamental issues involved in the (real-time) task assignment problem on heterogeneous multiprocessors. Second, abstracting out the architectural features of the computing platform *generalized* the problem under consideration to a *variant* of the well-known bin-packing problem (in which the bins are of different types and each item has a different size for a different type of bin) so that the solutions designed here are applicable in several domains. However, one of the limitations of making such assumptions is that the solutions designed here are not directly applicable in real-world and/or industrial settings. Despite this limitation, we believe that, this work is significant since, it (i) is one of the initial works to explore the problem of scheduling real-time tasks on heterogeneous multiprocessors, (ii) provided a deeper understanding of the problem and (iii) made considerable inroads in solving the problem (at least the theoretical aspects of it).

In order to achieve the larger goal of deploying the (industrial) real-time systems on heterogeneous multiprocessors, it is very essential to relax some of these assumptions and accordingly either adapt the solutions proposed here or design the new solutions, if necessary. This can be the generic direction in which the work can be extended in near future. We now list a couple of specific options for extending this work.

9.3.1 Constrained-deadline and arbitrary-deadline sporadic tasks

Although the implicit-deadline sporadic task model has been a favorite model for researchers, unfortunately, not all the real-time systems workload can be captured by this model due to more complex deadlines. Specifically, many real-time systems need to process alarms or emergency events. These events occur infrequently (perhaps only once during the entire operation of the system — ideally they would never happen) but when they do occur, the computer system must perform processing within a very tight deadline: the *constrained-deadline* sporadic model captures such systems [Mok83b]. This model is like the implicit-deadline sporadic model but for each task it must hold that its deadline is no greater than its minimum inter-arrival time. In other systems (e.g., signal processing), a task generates jobs with a high rate (i.e., small inter-arrival time) to perform sampling at high rate but the deadline is larger than the minimum inter-arrival time: the *arbitrary-deadline* sporadic model captures such systems [Mok83b]. This model is like the implicit-deadline sporadic model but it allows for each task deadline to take any value. Because of the usefulness of the arbitrary-deadline sporadic model, researchers created a plethora of results for this model for scheduling tasks on a single processor [Leh90, BMR90] and for identical multiprocessors [BF05, BF07a, CC11]. Recently, an algorithm for assigning arbitrary-deadline

sporadic tasks on a heterogeneous multiprocessor has been developed [MSRvdSW12]. This algorithm has polynomial time-complexity and a proven speed competitive ratio. Unfortunately, this is the only work that exists for assigning arbitrary-deadline sporadic tasks on heterogeneous multiprocessors and it has a very large speed competitive ratio (shown to be 17.9). So, it is of interest to design algorithms with lower (that is better) speed competitive ratios.

9.3.2 Shared hardware resources

The presence of shared hardware resources such as memory, cache, bus, interconnect, etc. in multiprocessor computing platforms are going to have a significant impact on the execution of tasks and hence on the schedulability of real-time systems. For example, the presence of such shared hardware resources does not even facilitate and in fact complicates the computation of upper bounds on the key timing parameters such as the worst-case execution time of a task. Further, many chip manufacturers including Tilera and Intel have declared plans for many-core chips with hundreds of cores. On such computing platforms, the question on whether tasks meets their deadlines will increasingly become dependent on sharing of these hardware resources. Unfortunately, there is a very little effort to study the impact of such shared hardware resources on the schedulability of real-time systems on heterogeneous multiprocessors. Hence, extending the work in this direction will be of great interest to the community.

9.3.3 Parallel task model

One of the assumptions we made at the beginning of this work is that, a job cannot execute on more than one processor at any given time — referred to as the *sequential programming model*. This assumption was done in order to simplify the complexity of the problem. However, it is restrictive, in the sense that, it may not allow us to fully exploit the underlying parallelism of a multicore computing platform. In order to take advantage of the available parallelism, there is a need to shift to *parallel programming model*s. Such models introduce a new dimension to the problem as they allow jobs to be split into parallel execution segments at specific points thereby leading to shorter response times whenever possible. This in turn increases the schedulability of the system.

Recently, researchers have started studying the scheduling problem for parallel task model on multicores (e.g., [LKR10, SALG11, NBGM12, LALG13]). However, these are initial efforts and a lot of issues need to be addressed in this direction. Hence, extending the work in this direction may also be of great interest to the community.

9.4 Concluding remarks

With the emergence of multicores, there is a strong interest in deploying embedded real-time systems on multiprocessor computing platforms. Most of these multicores are heterogeneous in nature and unfortunately the current scheduling theory is not mature enough to address some of the

challenges that arise while deploying real-time systems on heterogeneous multicores. This called for the development of scheduling theory in order to facilitate the deployment of embedded real-time system on heterogeneous multicores. This dissertation addresses some of the fundamental problems in this regard and lays the foundation for the future research which can focus on extending the theory by removing some of the simplifying assumptions of this work, thereby increasing the number of real-time systems that may be deployed on heterogeneous multicores.

References

[AB08] Björn Andersson and Konstantinos Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 243–252, 2008.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1st edition, 2009.

[ABJ01] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202, 2001.

[AE10] Björn Andersson and Arvind Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.

[AJ03] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 33–40, 2003.

[AMD13a] AMD Inc. AMD Accelerated Processing Units. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx, 2013.

[AMD13b] AMD Inc. AMD Dual-Core Processors: Twice the Processing Power of Single-Core Chip. http://www.amd.com/us/products/technologies/multi-core-processing/Pages/dual-core-processing.aspx, 2013.

[And03] Björn Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, Chalmers University of Technology, 2003.

[App13] Apple Inc. Apple A5X: Dual-core CPU and Quad-core GPU. http://support.apple.com/kb/SP647, 2013.

[ARB10] Björn Andersson, Gurulingesh Raravi, and Konstantinos Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 239–248, 2010.

[ARM13a] ARM Inc. Cortex-A9 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php, 2013.

[ARM13b] ARM Inc. The big.LITTLE Processing with Cortex-A15 and Cortex-A7. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2013.

[AS00] James Anderson and Anand Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro conference on Real-time systems*, pages 35–43, 2000.

[AT07a] Björn Andersson and Eduardo Tovar. Competitive Analysis of Static-Priority of Partitioned Scheduling on Uniform Multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 111–119, 2007.

[AT07b] Björn Andersson and Eduardo Tovar. Competitive Analysis of Partitioned Scheduling on Uniform Multiprocessors. In *Proceedings of the 15th International Workshop on Parallel and Distributed Real-Time Systems*, pages 1–8, 2007.

[BA09] Konstantinos Bletsas and Björn Andersson. Notional Processors: An Approach for Multiprocessor Scheduling. In *Proceedings of the 15th IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2009.

[Bar04a] Sanjoy Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 37–46, 2004.

[Bar04b] Sanjoy Baruah. Partitioning real-time tasks among heterogeneous multiprocessors. In *Proceedings of the 33rd International Conference on Parallel Processing*, pages 467–474, 2004.

[Bar04c] Sanjoy Baruah. Task partitioning upon heterogeneous multiprocessor platforms. In *Proceedings of the 10th IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 536–543, 2004.

[Bar11] Sanjoy Baruah. Task assignment on two unrelated types of processors. In *Proceedings of the 19th International Conference on Real-Time and Network Systems*, pages 69–78, 2011.

[Bar13] Sanjoy Baruah. Partitioned EDF scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013.

[BC03] Gerald Borsuk and Timothy Coffey. Moore's law: A department of defense perspective. In *INSS CTNSP Defense Horizons*, volume 30, pages 1–8. Institute for National Strategic Studies (INSS), July 2003.

[BCL05] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[BCL09] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2009.

[BCPV96] Sanjoy Baruah, Neil Cohen, Greg Plaxton, and Donald Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[BF05] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 321–329, 2005.

[BF07a] Sanjoy Baruah and Nathan Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems*, 36(3):199–226, August 2007.

[BF07b] Sanjoy Baruah and Nathan Fisher. The Partitioned Dynamic-priority Scheduling of Sporadic Task Systems. *Real-Time Systems*, 36(3):199–226, August 2007.

[BLBA07] Aaron Block, Hennadiy Leontyev, Bjorn Brandenburg, and James Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2007.

[Ble07] Kanstantinos Bletsas. *Worst-case and Best-case Timing Analysis for Real-time Embedded Systems with Limited Parallelism*. PhD thesis, The University of York, 2007.

[BLK83] J. Blazewicz, J. K. Lenstra, and A. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[BMR90] Sanjoy Baruah, Aloysius Mok, and Louis Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[BRH90] Sanjoy Baruah, Louis Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, October 1990.

[BTW95] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

[Bur91] Alan Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116 –128, may 1991.

[CB11] Bipasa Chattopadhyay and Sanjoy Baruah. A lookup-table driven approach to partitioned scheduling. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 257–265, 2011.

[CC11] Jian-Jia Chen and Samarjit Chakraborty. Resource augmentation bounds for approximate demand bound functions. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 272–281, 2011.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*, chapter 30. Chapman & Hall/CRC, 2004.

[CG06] Liliana Cucu and Jöel Goossens. Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors. In *Proceedings of the 11th Conference on Emerging Technologies and Factory Automation*, pages 397–404, sept. 2006.

[CGJ97] Edward Coffman, Michael Garey, and David Johnson. Approximation algorithms for NP-hard problems. chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Ed.* McGraw-Hill, 2001.

[CSV12] José Correa, Martin Skutella, and José Verschae. The power of preemption on unrelated machines and applications to scheduling orders. *Math. Oper. Res.*, 37(2):379–398, May 2012.

[DAN+11] Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan Petters, Arvind Easwaran, and Jinkyu Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proceedings of the 8th IEEE International Conference on Embedded Software and Systems*, pages 1068–1075, nov 2011.

[DB11] Robert Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Survey*, 43(4):35:1–35:44, October 2011.

[Der74] Michael Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of IFIP Congress (IFIP'74)*, 1974.

[Dha77] Sudarshan K. Dhall. *Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems.* PhD thesis, University of Illinois at Urbana-Champaign, 1977.

[DJ06] Vivek Darera and Lawrence Jenkins. Utilization Bounds for RM Scheduling on Uniform Multiprocessors. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 315–321, 2006.

[DL78] Sudarshan K. Dhall and Chung Laung Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[DRBB09] Robert Davis, Thomas Rothvoß, Sanjoy Baruah, and Alan Burns. Exact quantification of the sub-optimality of uniprocessor fixed priority preemptive scheduling. *Real-Time Systems*, 43(3), November 2009.

[DVT12] Matthew DeVuyst, Ashish Venkat, and Dean Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 261–272, 2012.

[FB03] Shelby Funk and Sanjoy Baruah. Characteristics of EDF schedulability on uniform multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 211–218, july 2003.

[FBB06] Nathan Fisher, Sanjoy Baruah, and Theodore P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 118–127, 2006.

[GAB02] Paolo Gai, Luca Abeni, and Giorgio Buttazzo. Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS 2002)*, pages 231–238, Vienna, Austria, June 2002.

[Gee05] David Geer. Taking the Graphics processor Beyond Graphics. *IEEE Computer*, 38(9):14–16, 2005.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems.*, 25(2-3):187–205, September 2003.

[GHF+06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GJ78] Michael Garey and David Johnson. "Strong" NP-Completeness results: Motivation, examples, and implications. *Journal of ACM*, 25(3):499–508, July 1978.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman & Co, 1979.

[GSYY10] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 165–174, 2010.

[Gur13] Gurobi Optimization Inc. Gurobi optimizer. http://www.gurobi.com, 2013.

[HBL12] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Parallel-task scheduling on multiple resources. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 233–244, 2012.

[Hor74] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[HS76] Ellis Horowitz and Sartaj Sahni. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *Journal of the ACM*, 23:317–327, April 1976.

[HS86] Dorit Hochbaum and David Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. In *Proc. of the sixth conference on Foundations of software technology and theoretical computer science*, pages 382–393, 1986.

[HT73] John Hopcroft and Robert Tarjan. Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[IBM12] IBM Inc. IBM ILOG CPLEX Optimizer: High-performance mathematical programming solver for linear programming, mixed integer programming, and quadratic programming. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2012.

[Int13a] Intel Corp. Bay Trail: Multicore SoC Family for Mobile Devices. http://www.intel.com/newsroom/kits/idf/2013_fall/pdfs/bay_trail_fact_sheet.pdf, 2013.

[Int13b] Intel Corp. The 4th Generation Intel Core i7 Processor. http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html, 2013.

[Int13c] Intel Corporation. Intel Atom Processor Z6xx Series-Based Platform for Embedded Computing, 2013. http://www.intel.com/content/www/us/en/processors/atom/intel-atom-processor-z6xx-series.html.

[Int13d] Intel Corporation. Intel Core 2 Extreme Mobile Processor. http://www.intel.com/products/processor/core2xe/mobile/index.htm, 2013.

[Int13e] Intel Corporation. Intel Core 2 Quad Processors. http://www.intel.com/products/processor/core2quad/index.htm, 2013.

[Joh73] David Johnson. *Near-optimal Bin Packing Algorithm*. PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[Jon] Jonah Alben. NVIDIA Brings Kepler, World's Most Advanced Graphics Architecture, to Mobile Devices. http://blogs.nvidia.com/blog/2013/07/24/kepler-to-mobile/ as of Oct 1, 2013.

[Jon97] Mike Jones. What Happened on Mars?, 1997. http://www.ece.cmu.edu/~raj/mars.html.

[JP99] Klaus Jansen and Lorant Porkolab. Improved approximation schemes for scheduling unrelated parallel machines. In *Proceedings of the* 31^{st} *annual ACM symposium on Theory of computing*, pages 408–417, 1999.

[JPKA95] Todd Jochem, Dean Pomerleau, Bala Kumar, and Jeremy Armstrong. Pans: a portable navigation platform. In *Proceedings of the Intelligent Vehicles Symposium*, pages 107–112, 1995.

[Kar84] Narendra Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[KAS93] Daniel Katcher, Hiroshi Arakawa, and Jay Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, 1993.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[KV06] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 3rd edition, 2006.

[LALG13] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks. In *25th Euromicro Conference on Real-Time Systems*, pages 3–13, 2013.

[LBOS95] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, December 1995.

[LDG04] José López, José Díaz, and Daniel García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems Journal*, 28:39–68, 2004.

[Leh90] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201 –209, 1990.

[LFS+10] Greg Levin, Shelby Funk, Caitlin Sadowskin, Ian Pye, and Scott Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 3–13, 2010.

[LGDG03] José López, Manuel García, José Díaz, and Daniel García. Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Systems*, 24(1):5–28, January 2003.

[Liu69] Chung Laung Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, II:37–60, 1969.

[Liu00] Jane Liu. *Real-Time Systems*. Prentice Hall, 1st edition, 2000.

[LKR10] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium*, pages 259–268, 2010.

[LL73] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[LM08] Charles Leiserson and Ilya Mirman. How to survive the multicore software revolution. http://www.scribd.com/doc/95085266/How-to-Survive-the-Multicore-Software-Revolution, 2008.

[LR10] Karthik Lakshmanan and Ragunathan Rajkumar. Scheduling Self-Suspending Real-Time Tasks with Rate-Monotonic Priorities. In *Proceedings of the 16th IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 3–12, 2010.

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the 10th Real Time Systems Symposium*, pages 166 –171, dec 1989.

[LSL+09] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 57–67, 2009.

[LST90] Jan Lenstra, David Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46:259–271, 1990.

[LW82] Joseph Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[LY08] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming, 3rd Ed.* International Series in Operations Research & Management Science, 2008.

[LYGY10] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 339–349, 2010.

[Mok83a] Aloysius Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-time Environments*. PhD thesis, Massachusetts Institute of Technology, 1983.

[Mok83b] Aloysius K. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. Technical report, Electrical Engineering and Computer Science Dept., Massachusetts Institute of Technology (USA), 1983.

[MSRvdSW12] Alberto Marchetti-Spaccamela, Cyriel Rutten, Suzanne van der Ster, and Andreas Wiese. Assigning sporadic tasks to unrelated parallel machines. In *Proceedings of the 39th IEEE International Colloquium on Automata, Languages and Programming (ICALP 2012)*, pages 665–676, 2012.

[NBGM12] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Milojevic Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 321–330, 2012.

[NBN+12] Geoffrey Nelissen, Vandy Berten, Vincent Nélis, Joël Goossens, and Milojevic Milojevic. U-EDF: An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 13–23, 2012.

[Nvi12] Nvidia Inc. Tegra 2 and Tegra 3 Super Chip Processors. http://www.nvidia.com/object/tegra-3-processor.html, 2012.

[OB98] Dong-Ik Oh and Ted Baker. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, 15(2):183–192, September 1998.

[OS95] Yingfeng Oh and Sang H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical report, University of Virginia, Charlottesville, VA, USA, 1995.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pot85] Chris N. Potts. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics*, 10:155–164, 1985.

[PSC+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 741–746, 2010.

[PSTW97] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 140–149, 1997.

[Qua13] Qualcomm Inc. Snapdragon processors. http://www.qualcomm.com/snapdragon, 2013.

[RA13] Gurulingesh Raravi and Björn Andersson. Task assignment algorithm for two-type heterogeneous multiprocessors using cutting planes. Technical report, 2013.

[RAB13] Gurulingesh Raravi, Björn Andersson, and Konstantinos Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. *Real-Time Systems*, 49(1):29–72, 2013.

[RABN12] Gurulingesh Raravi, Björn Andersson, Konstantinos Bletsas, and Vincent Nélis. Task Assignment Algorithms for Two-Type Heterogeneous Multiprocessors. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 34–43, 2012.

[RAEP07] Jakob Rosén, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60, 2007.

[RM09] Austin Rogers and Aleksandar Milenković. Security extensions for integrity and confidentiality in embedded processors. *Microprocess. Microsyst.*, 33(5-6):398–414, August 2009.

[RN12a] Gurulingesh Raravi and Vincent Nélis. A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors. In *Proceedings of the 33rd IEEE International Real-Time Systems Symposium*, pages 117–126, 2012.

[RN12b] Gurulingesh Raravi and Vincent Nélis. A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 117–126, 2012.

[RS94] Krithi Ramamritham and John Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, jan 1994.

[RSL88] Raj Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259 –269, 1988.

[RSS90] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):184–194, April 1990.

[SAA+04] Lui Sha, Tarek Abdelzaher, Karl Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, nov 2004.

[SALG11] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 217–226, 2011.

[Sam13a] Samsung Inc. Samsung Exynos processor. www.samsung.com/exynos/, 2013.

[Sam13b] Samsung Inc. Samsung's new eight-core Exynos 5 Octa SoC promises not to hog battery. http://arstechnica.com/gadgets/2013/01/samsungs-new-eight-core-exynos-5-octa-soc-promises-efficiency/, 2013.

[SGG09] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 8th edition, 2009.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[SNE10] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 759–764, 2010.

[Spe] IEEE Spectrum. Cmu's autonomous car doesn't look like a robot. http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/cmu-autonomous-car-doesnt-looks-like-a-robot.

[SRL90] Lui Sha, Raj Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[ST 12] ST Ericsson. NOVA Processor Family – Highest Performance Application Processors, 2012. `http://www.stericsson.com/products/application_processors.jsp`.

[Tan07] Stewart Tansley. Trends in Embedded Systems–A Microsoft Perspective. In *Proceedings of the 3rd International Conference on Microelectronic Systems Education*, page 4, june 2007.

[Tex13] Texas Instruments. OMAP applications processors. http://www.ti.com/omap, 2013.

[WBB13] Andreas Wiese, Vincenzo Bonifaci, and Sanjoy Baruah. Partitioned EDF scheduling on a few types of unrelated multiprocessors. *Real-Time Systems*, 49(2):219–238, 2013.

[Wes00] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2000.

[Zha03] Feng Zhao. Programming Embedded Sensor Networks, 2003. Invited talk at the First International Conference on Hardware/Software Codesign and System Synthesis.