Task and Message Co-scheduling Strategies in Real-time Cyber-Physical Systems

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy
in
Computer Science and Engineering

Submitted by
Sanjit Kumar Roy

Under the guidance of
Dr. Arnab Sarkar

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

September, 2022

Abstract

Cyber-Physical Systems (CPSs), like those in the automotive and avionic domains, smart grids, nuclear plants, etc., often consist of multiple control subsystems running on distributed processing platforms. Most of these control systems are modeled as real-time *independent tasks* or *Precedence-constrained Task Graph* (PTG), depending on the nature of interactions between their functional components. These tasks typically read their input parameters via sensors. The sensed inputs are then transmitted as messages over communication channels to processing elements where corresponding control outputs are computed. The outputs in turn, are communicated to actuators as messages through communication channels. *This dissertation presents several novel real-time task-message co-scheduling strategies for safety-critical CPSs, consisting of various types of task and execution platform scenarios.*

The entire thesis work is composed of multiple contributions categorized into four phases, each of which is targeted towards a distinct task/platform scenario. The first phase delves with the design of co-scheduling strategies for independent periodic real-time tasks, with associated input and output messages, on a bus-based homogeneous multiprocessor system. Although most scheduling approaches have traditionally been oriented towards homogeneous multiprocessors, continuous demands for higher performance and reliability along with better thermal and power efficiencies, have created an increasing trend towards distributed heterogeneous processor platforms. In the second phase, we have considered the problem of scheduling real-time systems modeled as PTGs on fully-connected heterogeneous systems. The tasks considered in both the first and second phases, may have multiple implementations designated as service-levels/quality-levels, with higher service-levels producing more accurate results and contributing to higher *rewards*/*Quality of Service* (QoS) for the system. In the third phase, we extend the problem of scheduling PTGs on fully-connected platforms, to CPS systems where the processors are connected through a limited number of bus based shared communication channels. While the third phase considers the problem of scheduling a single PTG, the final phase solves the problem of scheduling multiple independent periodic real-time PTGs. The works proposed in the third and fourth phases, endeavour towards the maximization of slack within the generated schedule, which can then be used to minimize energy dissipation in the system. The thesis proposes both optimal and heuristic solution approaches for all its phases. Practical applicability and efficacy of the presented schemes have been extensively evaluated through simulation-based experiments as well as real-world benchmarks.

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Sanjit Kumar Roy

Certificate

This is to certify that this thesis entitled, **"Task and Message Co-scheduling Strategies in Real-time Cyber-Physical Systems"**, being submitted by **Sanjit Kumar Roy**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under my supervision and guidance. The thesis, in my opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of my knowledge, it has not been submitted elsewhere for the award of the degree.

....

Dr. Arnab Sarkar
Associate Professor
Advanced Technology Development Centre
IIT Kharagpur

Dedicated to

My son, parents and wife

Acknowledgments

I would like to thank several people who played an essential role in the concretization of this thesis, directly and indirectly.

I would like to begin by thanking my Ph.D supervisor Prof. Arnab Sarkar. I am immensely grateful and wish to express my deepest gratitude to Prof. Sarkar for providing me guidance, inspiration, advice, support in every step of my work, constantly worrying about my future career and well-being. I feel very privileged to have had the opportunity to learn from and work with him. His constant guidance and support paved the way for my development as a research scientist and changed my personality, ability, and nature in many ways. I have been fortunate to have such an advisor who gave me the freedom to explore on my own and at the same time, provide the guidance to recover when my steps faltered. I would also like to thank my administrative supervisor Prof. Chandan Karfa for taking care of all the official formalities that make my life easy at IIT Guwahati and for his continuous encouragement during this journey.

Besides my advisors, I would like to thank other members of my doctoral committee, Prof. Hemangee K. Kapoor, Prof. Arijit Sur, and Prof. Sukanta Bhattacharjee, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives. I would also like to thank Prof. Santosh Biswas for his feedback on my work. I am thankful to the external thesis examiners as well as the anonymous reviewers of my papers for their thoughtful suggestions and feedback.

As a Ph.D scholar, life without friends would be a nightmare. I am blessed to have a myriad of friends. I am thankful to my dearest friend Projit, whose belief in me, continuous support, and encouragement gave me the strength and confidence to pursue a Ph.D. I would especially thank Niladri da, Biswajit da, Suddhasil da, Shounak, Subhrendu and Ranajit da for their constant support. I am deeply indebted to Rajesh, Rahul, Ramanuj, Supriyo, and Diptesh for encouraging me throughout my Ph.D journey and providing me their unconditional support with love, to sort out technical as well as personal problems. I was fortunate to share many cherished memories at IIT Guwahati with all of them. Many thanks to Sayani, Kankana, Rishabh, Debabrata, Abothula for collaborating with me in research during my Ph.D tenure. I would like to thank Piyoosh, Basina, Ujjwal da, Subrata, Sonjoy, Sharma ji, Rakesh da, Sujoy for their support.

I would like to thank all the members of IIT Guwahati, especially the Department of Computer Science and Engineering, for providing such a great research environment. I also thank all the security personnel, mess workers, canteen and housekeeping staff, who made my life smooth at the campus. While it is not possible to mention each of them, I would like to express my deepest gratitude to all the people who helped me directly or indirectly during my Ph.D.

I am grateful to my parents for their unconditional love, sacrifice, encouragement and consistent support, in every event of my life. I am also very fortunate to have wonderful in-laws who encouraged me at every step of my Ph.D journey. I am forever indebted to my wife, Sudipta Ghosh. Without her love, sacrifice, constant encouragement and faith in me, none of this would have been possible. She has been incredibly supportive of me through the ups and downs over the past few years. I cannot thank her enough for being such a wonderful wife and friend. Finally, my wholehearted thanks to our little son Sri Pratyush Roy, who is the ultimate source of inspiration for me.

Sanjit Kumar Roy

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

ABS	*Anti-lock Braking System*
ACC	*Adaptive Cruise Controller*
ADAS	*Advanced Driver Assistance System*
ALAP	*As Late As Possible*
ALOLA	*Accurate Low Overhead Level Allocator*
ASAP	*As Soon As Possible*
BSAMS	*Baseline Slack Aware Multi-PTG Scheduler*
BF	*Boundary Fair*
CCR	*Communication to Computation Ratio*
CC-TMS	*Contention Cognizant Task and Message Scheduler*
CPOP	*Critical Path On a Processor*
CPS	*Cyber-Physical System*
CPSs	*Cyber-Physical Systems*
CSP	*Constraint Satisfaction Problem*
DAG	*Directed Acyclic Graph*

DECM *Downward Energy Consumption Minimization*

DP *Dynamic Programming*

DP-Fair *Deadline Partitioning Fair*

DUECM *Downward and Upward Energy Consumption Minimization*

DVFS *Dynamic Voltage and Frequency Scaling*

EDF *Earliest Deadline First*

EFT *Earliest Finish Time*

EPS *Electric Power Steering*

ERfair *Early-Release fair*

EST *Earliest Start Time*

FMS *Flight Management System*

G-SAQA *Global Slack Aware Quality-level Allocator*

HEFT *Heterogeneous Earliest Finish Time*

HLF *Highest Level First*

HSV *Heterogeneous Selection Value*

ILP *Integer Linear Programming*

ILP-ES *ILP for Energy-aware Scheduling*

ILP-ETR *ILP with Explicit Time Reduced*

ILP-NC *ILP with Non-overlapping Constraints*

ILP-SANC *ILP - Service-level Allocation with Non-overlapping Constraints*

ILP-SATC *ILP - Service-level Allocation with Timed Constraints*

LCM *Least Common Multiple*

LHS	*left hand side*
LLREF	*Largest Local Remaining Execution time First*
MCP	*Modified Critical Path*
MILP	*Mixed-Integer Linear Programming*
MMCKP	*Multi-dimensional Multiple-Choice Knapsack formulation*
MWSTR	*Multiple-Workflows-Slack-Time-Reclaiming*
PEFT	*Predict Earliest Finish Time*
Pfair	*Proportional fair*
PTG	*Precedence-constrained Task Graph*
PTGs	*Precedence-constrained Task Graphs*
QoS	*Quality of Service*
RHS	*right hand side*
RM	*Rate-Monotonic*
RT-CPS	*Real-Time Cyber-Physical System*
RT-CPSs	*Real-Time Cyber-Physical Systems*
RUN	*Reduction to UNiprocessor*
SAFLA	*Slack Aware Frequency Level Allocator*
SMT	*Satisfiability Modulo Theories*
TC	*Traction Controller*
TPG	*Task Priority Generator*
T-SAQA	*Total Slack Aware Quality-level Allocator*
TMC	*Task and Message Co-scheduler*

List of Symbols

T_i	i^{th} task
M_i	i^{th} message
A_i	Arrival time of task T_i
S_i	Start time of task T_i/message M_i
F_i	Finish time of task T_i/message M_i
D_i	Deadline of task T_i
e_i	Execution time of task T_i
π_i	Period of task T_i
n	Number of task nodes
m	Number of message nodes
sl_{il}	l^{th} service-level of task T_i
SL_i	$SL_i = \{sl_{i1}, sl_{i2}, \ldots, sl_{i\|SL_i\|}\}$; Set of service-levels of task T_i
mx_{ij}	Input message of task T_i at service-level sl_{ij}
my_{ij}	Output message of task T_i at service-level sl_{ij}
wt_{ij}	Computation demand of task T_i at service-level sl_{ij}

wm_{ij}	Communication demand of task T_i at service-level sl_{ij}
p	Number of processors
b	Number of buses
P	$P = \{P_1, P_2, \ldots, P_p\}$; Set of processors
P_r	r^{th} processor
B	$B = \{B_1, B_2, \ldots, B_b\}$; Set of buses
B_r	r^{th} bus
G	A *Directed Acyclic Graph* (DAG)/*Precedence-constrained Task Graph* (PTG)
T	$T = \{T_1, T_2, \ldots, T_n\}$; Set of tasks
$\mathcal{V}$	Set of nodes in a PTG
$\mathcal{E}$	Set of edges in a PTG
$\mathcal{V}_i$	i^{th} node of a PTG
T_{source}	Source task node of a PTG
T_{sink}	Sink task node of a PTG
m_{ij}	Message from task T_i to task T_j
e_{ir}	Execution time of task T_i on processor P_r
e_{ilr}	Execution time of task T_i at service-level sl_{il} on processor P_r
c_{kr}	Communication time of message M_k on bus B_r
N	Number of PTGs
$\mathcal{G}$	$\mathcal{G} = \{G^1, G^2, \ldots, G^N\}$; Set of PTGs
G^g	g^{th} PTG
$\mathcal{V}^g$	Set of nodes in PTG G^g

$\mathcal{E}^g$	Set of edges in PTG G^g
$\mathcal{V}_i^g$	i^{th} node in PTG G^g
n^g	Number of task nodes in PTG G_g
m^g	Number of message nodes in PTG G_g
T^g	$T^g = \{T_1^g, T_2^g, \ldots, T_{n^g}^g\}$; Set of task nodes in PTG G^g
M^g	$M^g = \{M_1^g, M_2^g, \ldots, M_{m^g}^g\}$; Set of message nodes in PTG G^g
T_i^g	i^{th} task node in PTG G^g
M_k^g	k^{th} message node in PTG G^g
$indeg(\mathcal{V}_i)$	In degree of node $\mathcal{V}_i$
$outdeg(\mathcal{V}_i)$	Out degree of node $\mathcal{V}_i$
$pred(\mathcal{V}_i)$	Predecessor nodes of $\mathcal{V}_i$
$succ(\mathcal{V}_i)$	Successor nodes of $\mathcal{V}_i$
R	$R = \{R_1, R_2, \ldots, R_r\}$; Set of resources
R_r	r^{th} resource
L_r	$L_r = \{1, 2, \ldots, \|L_r\|\}$; Set of voltage/frequency levels of processor P_r
V	Voltage
V_{rl}	Voltage of processor P_r at voltage/frequency level l
f_{rl}	Frequency of processor P_r at voltage/frequency level l
e_{irl}^g	Execution time of task T_i^g on processor P_r at voltage/frequency level l
c_{kr}^g	Communication time of message M_k^g on bus B_r
D^g	Deadline of PTG G^g
$\mathcal{H}$	$\mathcal{H}$ is the hyperperiod

I^g	Number of iterations/instances of G^g within the hyperperiod $\mathcal{H}$
G^{gq}	q^{th} instance of G^g within the hyperperiod $\mathcal{H}$
$\mathcal{V}^{gq}$	Set of nodes in G^{gq}
$\mathcal{E}^{gq}$	Set of edges in G^{gq}
$\mathcal{V}_i^{gq}$	i^{th} node in G^{gq}
T_i^{gq}	i^{th} task node in G^{gq}
M_k^{gq}	k^{th} message node in G^{gq}

Chapter 1

Introduction

Today, *Cyber-Physical Systems* (CPSs) are becoming an important part of our daily lives. A CPS is composed of physical sub-systems together with computing and networking (cyber sub-systems), where embedded computers and networks monitor and control the physical processes. For example in a traditional aircraft, a pilot controls the aircraft using movable surfaces on the wings and tail, connected to the cockpit through mechanical and hydraulic sub-systems. On the other hand in a fly-by-wire aircraft, the control commands are electronically sent by a flight computer over a network to actuators at the wings and tail, making the aircraft much lighter than a traditional aircraft, resulting in better fuel efficiency.

Many CPSs in domains such as automotive, avionics, smart grids, nuclear plants, industrial process control, etc., often consist of multiple control sub-systems running on distributed processing platforms. For example, an automotive system consists of several distributed sub-systems like *Adaptive Cruise Controller* (ACC), *Traction Controller* (TC), *Anti-lock Braking System* (ABS), *Advanced Driver Assistance System* (ADAS), etc. Most of these control systems are modeled as real-time *independent tasks* or *Precedence-constrained Task Graphs* (PTGs), depending on the nature of interactions between their functional components. In order to meet specifications related to timing, reliability, energy, etc., while keeping the design methodology simple, each of these sub-systems often execute on its own dedicated processing units, making the overall system architecture federated in nature. Such federated architectures may result in higher design costs compared to more integrated execution of applications on smaller consolidated platforms. However, consolidated architectures lead to significantly increased design complexity due to a higher degree of contention for shared resources (such as processing elements, buses, memories, etc). On a different trajectory, continuous demands for higher performance and reliability within stringent resource budgets is driving a shift from homogeneous to heterogeneous processing platforms for the implementation of today's CPSs. Given a distributed platform consisting of a set of processing elements connected through communication channels, the successful execution of tasks and transmission of messages (while satisfying deadlines and other resource constraints), is essentially a real-time task-message co-scheduling problem.

The problem of scheduling real-time tasks on processors is being studied by researchers for many decades. A scheduler allocates resources to the tasks and decides their execution sequence taking schedulability constraints into consideration. Real-time task sets can be of different types, (i) *Independent task set:* Here, a task is not interrelated to any other tasks in the set through precedence constraints i.e. execution of the task does not depend on the data received from one or more other tasks, (ii) *Dependent task set:* Here, completion of execution of a task may be dependent on messages received from one or more other tasks in the set. To schedule a set of independent tasks, a scheduler needs to satisfy deadline and resource constraints. On the other hand to schedule a set of dependent tasks (represented as a PTG), a scheduler additionally needs to satisfy the precedence constraints among tasks. A distributed platform may consist of multiple homogeneous or heterogeneous processors which are either fully connected or connected through shared buses. Depending on processor types, a task may require same or distinct execution times on different processors. So, in addition to determining start/finish times of tasks, a scheduler also needs to find out the task-to-processor mappings while generating a schedule for tasks executing on a heterogeneous distributed platform. *This dissertation focuses towards co-scheduling strategies for real-time CPSs, where functionalities may be represented as independent tasks, PTGs or even multiple independent applications each represented as a separate PTG. The targeted platforms* *may consist of homogeneous/heterogeneous processing elements, which may either be fully interconnected or connected through shared possibly heterogeneous buses.* We have added a taxonomy in Figure 1.1 and the highlighted boxes represent scope of this thesis.

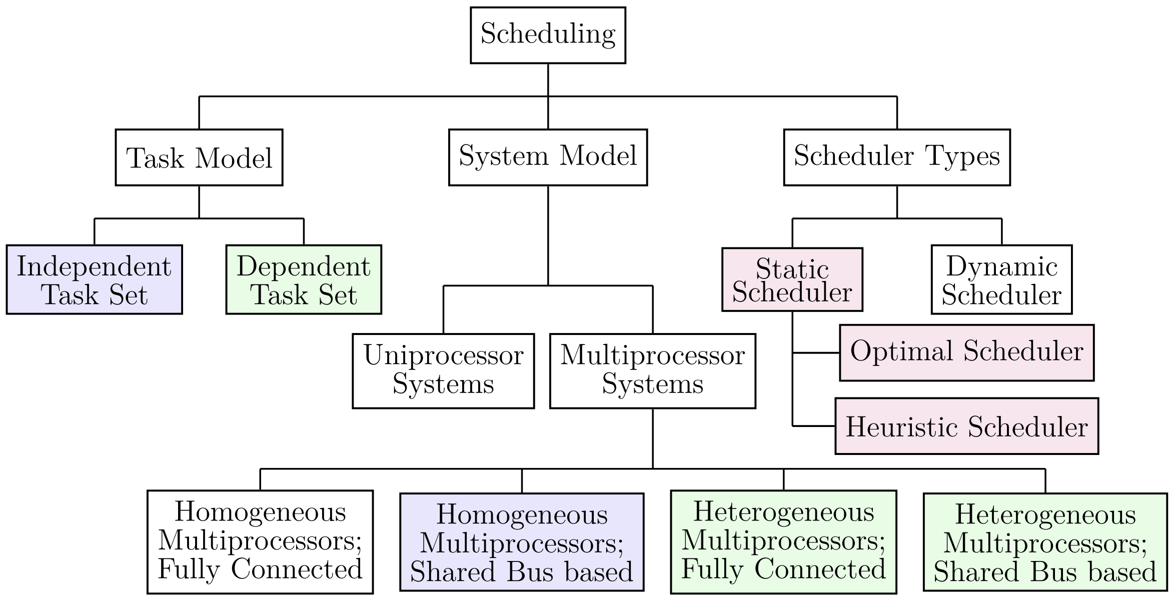


Figure 1.1: *Taxonomy; Highlighted boxes represent scope of the thesis*

Solution approaches to real-time scheduling problems can be broadly classified as *heuristic* and *optimal*. Heuristic schedule construction methodologies are typically based on the satisfaction of a set of sufficiency conditions and cannot take into consideration all necessary schedulability requirements. Consequently, such scheduling schemes become sub-optimal in nature with their results often deviating significantly from their optimal counterparts. On the other hand, optimal solution approaches take all necessary and sufficient conditions into consideration and have the potential to make a fundamental difference in time-critical systems with respect to performance, reliability, and other non-functional metrics like cost, power, space, etc. Optimal schedules can also act as benchmarks allowing accurate comparison and evaluation of heuristic solutions [9]. Several strategies including automata based synthesis, *Constraint Satisfaction Problem* (CSP)/*Satisfiability Modulo Theories* (SMT) based modeling, search based technique, ILP, etc. have been typically used to construct optimal schedulers for CPSs. *This thesis has focused towards the synthesis of optimal scheduler using a state-space search approach or CSP based modeling followed by solution generation using CPLEX [10] a standard industry grade constraint solver.*

Though, the optimal scheduling solutions may potentially deliver significantly better performance compared to sub-optimal heuristic solutions, finding optimal solutions may become prohibitively expensive for large problem sizes. Further, during design space exploration, multiple quick design iterations are needed and/or powerful server systems may not be available at the designer's disposal. In such cases the designer must resort to sub-optimal, yet satisfactorily good polynomial time heuristic solution for the problem at hand. Many heuristic schedulers which are commonly based on variations of the well known *list scheduling strategy* [1, 6, 11–13] are found in literature, particularly for CPS applications represented as task graphs.

This dissertation presents a few novel real-time optimal/heuristic offline task-message co-scheduling strategies for safety-critical CPSs consisting of various types of task and execution platform scenarios. In real-time safety-critical CPSs, where deadline misses may lead to catastrophic consequences, offline scheduling is often preferred as all timing requirements can be guaranteed before putting the system in operation, specially in cases where the task systems are persistent and do not vary dynamically at run time. Additionally, offline scheduling allows time and space complexities involved in solution space exploration to become independent of run-time scheduling overheads.

1.1 Related Work

Traditionally, scheduling of real-time *independent* tasks on multiprocessor systems has been based on either partitioned or global approaches [14–16]. With partitioning, the multiprocessor scheduling problem is transformed into uniprocessor scheduling problem, where a task is assigned to a designated processor and gets executed entirely on that processor. Well known optimal uniprocessor scheduling algorithms include *Rate-Monotonic* (RM) (static priority) and *Earliest Deadline First* (EDF) (dynamic priority), proposed by Liu and Layland [17]. However, a major drawback of partitioning is that, up to half of the system capacity may remain unutilized in order to ensure timing constraints of a given task set [18]. Unlike the fully partitioned approaches, more global schedulers like *Proportional fair* (Pfair) [19], PD^2 [20], *Early-Release fair* (ERfair) [21], *Boundary Fair* (BF) [22], *SA* [23], *Largest Local Remaining Execution time First* (LLREF) [24], *Reduction to UNiprocessor* (RUN) [25], *Deadline Partitioning Fair* (DP-Fair) [26] can achieve very high utilization of the system capacity by allowing migrations of tasks among processors. The Pfair scheduler proposed by Baruah et al. [19] is known to be the first optimal global real-time scheduler on multiprocessor systems for tasks with implicit deadlines. Based on Pfair, Anderson et al. [21] proposed a work-conserving multiprocessor scheduling algorithm called the ERfair scheduler. However both the above schemes attempt to maintain proportional fairness at each time slot and incur unrestricted preemption/migration overheads due to this. Recently, Levin et al. [26] proposed a semi-partitioned approximate proportional fair optimal scheduler called DP-Fair with much lower and bounded context switching overheads.

The problem of scheduling PTGs on multiprocessor systems has also received the attention of researchers over many decades. Various optimal solution approaches, such as linear programming, Best-first search, and other exhaustive enumeration techniques including model-based formal synthesis mechanisms, have been proposed [27,28]. Prasanna et al. [29] devised a control-theoretic optimal scheduling mechanism for task graphs executing on a homogeneous multiprocessor system. Later, they have extended their scheme to include communication overheads between task nodes [30]. Sarad et al. [9] developed a *Mixed-Integer Linear Programming* (MILP) based PTG scheduling strategy for platforms consisting of a set of fully connected *homogeneous* processing nodes. Liu et al. [31] considered the problem of task node assignment for PTGs, among heterogeneous clusters connected through communication links of various transmission capacities. Kanemitsu et al. [32] presented an ILP based optimal task scheduling scheme for fully-connected heterogeneous distributed systems. Hsiu et al. [33] developed an optimal and approximation algorithm for the scheduling of PTGs on heterogeneous distributed platforms with shared buses. However, a drawback of this scheme is the simplistic assumption that mapping of task nodes to processing elements are known a priori.

Although optimal scheduling solutions may potentially deliver significantly better performance, it may be noted that computation of such optimal solutions may often become prohibitively expensive for large problem sizes. Therefore, research in this domain has also focused towards the design of low-overhead heuristics that provide quick and satisfactory schedules. Heuristic scheduling of PTGs on multiprocessor platforms have often been dealt with *list scheduling* based techniques. These scheduling strategies typically maintain an ordered priority list of all tasks in the PTG [1, 6, 11–13] and involves two phases, (i) *task prioritization*: for selecting the highest-priority ready task and (ii) *processor selection*: for selecting a suitable processor that minimizes execution time. Some examples of this class of techniques include the *Modified Critical Path* (MCP) [34], *Highest Level First* (HLF) [35], *Critical Path On a Processor* (CPOP) [6], HEFT [6], PEFT [1] and *Heterogeneous Selection Value* (HSV) [11] algorithms. They attempt to construct a static-schedule for the given PTG to minimize the overall schedule length while satisfying resource and precedence constraints.

1.2 Challenges

Developing efficient scheduling strategies for diverse real-time applications in today's safety-critical CPSs must meet several challenges. We now enumerate a few such important challenges and discuss them [27].

1. **Timing requirements:**

   Real-time systems are characterized by their operations not only being logically correct, but also on the time at which they are performed. The time before which a task should complete its execution for the safety of the system, is called its deadline. *Scheduling schemes for safety-critical real-time systems must be able to guarantee the timing requirements (i.e., deadlines) associated with various types of tasks that co-exist in the system.*

2. **Resource constraints:**

Safety-critical systems are implemented on platforms consisting of a limited number of resources. Providing a lot of redundant hardware is not always possible/feasible as the system's cost increases, and the system's performance may degrade in terms of power/energy dissipation, etc. For example, in cost-sensitive safety-critical systems like cars, a cost differential of even a hundred dollars can make a commercial difference [36–39]. In addition, over the years, the nature of the processing elements used in real-time systems is transformed from uniprocessor to homogeneous multiprocessor platforms to heterogeneous multiprocessor platforms to cater to higher computation demands while adhering to restrictions on power/energy dissipation. *Scheduling schemes for safety-critical real-time systems must be able to effectively utilize available resources of the underlying platform to satisfy the resource constraints associated with the real-time task set.*

3. **Energy minimization:**
   Energy consumption in real-time systems has become an important issue with the increase in the number of processing elements. Effective energy management is important for battery-powered embedded systems, such as those deployed in autonomous mobile robots, wearable devices, industrial controllers, etc. Recharging or replacing batteries in such systems is not always practical or feasible. Hence, effective energy management can enhance the lifetime of the batteries resulting in higher performance and financial advantages. Even for systems directly connected to the power grid, reducing energy consumption provides significant monetary and environmental gains [40]. Scheduling schemes for real-time systems must optimize the energy consumption satisfying other constraints like timing, resource, precedence, etc.

1.3 Objectives

The principal aim of this dissertation has been to investigate the theoretical and practical aspects of co-scheduling strategies in safety-critical CPSs, keeping in view the challenges/hurdles discussed in the previous section. In particular, the objectives of this work may be summarized as follows:

1. Development of co-scheduling strategies for a set of independent periodic tasks executing on a bus-based homogeneous multiprocessor system, with the objective of maximizing system level QoS.

2. Design and implementation of QoS adaptive scheduling mechanisms for real-time systems modeled as PTGs, on fully-connected heterogeneous multiprocessor system.

3. Development of co-scheduling strategies for PTGs executing on a shared-bus based heterogeneous distributed platform.

4. Design of energy-aware processor-bus co-scheduling strategy for the heterogeneous distributed CPS platforms, as mentioned above.

1.4 Summary of work done

As part of this Ph.D research work, we have developed multiple scheduler design schemes for real-time CPSs. The entire thesis work is composed of multiple contributions categorized into four phases, each of which is targeted towards a distinct task/platform scenario.

1. **Task Scheduling on Homogeneous Distributed Systems**
   CPSs, including those in the automotive domain, are often designed by assigning to each task an appropriate criticality-based reward value that is acquired by the system on its successful execution. Additionally, each task may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher rewards for the system.

   This work proposes co-scheduling strategies for a set of independent periodic tasks executing on a bus-based homogeneous multiprocessor system, with the objective of maximizing system level QoS. Each service-level of any task has a distinct computation demand (serviced by one or more processors) and communication demand (serviced by a set of shared buses) with higher service-levels having higher resource demands. Successful execution of a task at a certain service-level is associated with a reward corresponding to that service-level and this reward is proportional to the task's relative importance/criticality. The objective of the task allocation mechanism is to maximize aggregate rewards such that both computation and communication resource demands of all tasks may be feasibly satisfied. The problem is posed as a *Multi-dimensional Multiple-Choice Knapsack formulation* (MMCKP) and present a *Dynamic Programming* (DP) solution (called MMCKP-DP) for the same. Although DP delivers optimal solutions, it suffers from significantly high overheads (in terms of running time and main memory consumption) which steeply increase as the number of tasks, service-levels, processors and buses in the system grows. Even for a system with a moderate task set consisting of 90 tasks, it takes approximately 1 hour 20 minutes and consumes a huge amount of main memory space (approximately 68 GB). Such large time and space overheads are often not affordable, especially when multiple quick design iterations are needed during design space exploration and/or powerful server systems are not available at the designer's disposal.

Therefore, in addition to the optimal solution approach, we propose an efficient but low-overhead heuristic strategy called ALOLA which consumes drastically lower time and space complexities while generating good and acceptable solutions which do not significantly deviate from the optimal solutions. ALOLA is a greedy but balanced heuristic service-level allocation approach that proceeds level by level so that a high aggregate QoS may be acquired by the system at much lower complexity compared to the optimal MMCKP-DP strategy. The mechanism starts by storing all tasks in a max-heap (based on a key $cost_i$) and assigning base service-levels to all tasks. The algorithm then proceeds by repeatedly extracting the task at the root of the heap, incrementing its service-level by 1, updating its *cost* value and reheapifying it, until residual resources are completely exhausted, or all the tasks have been assigned their maximum possible service-levels.

Our simulation based experimental evaluation shows that even on moderately large systems consisting of 90 tasks with 5 service-levels each, 16 processors and 4 buses, while MMCKP-DP incurs a run-time of more than 1 hour 20 minutes and approximately 68 GB main memory, ALOLA takes only about 196 μs (speedup of the order of 10^6 times) and less than 1 MB of memory. Moreover, while being fast, ALOLA is also efficient being able to control performance degradations to at most 13% compared to the optimal results produced by MMCKP-DP. Both the presented solution strategies (MMCKP-DP and ALOLA) assume DP-Fair [26], a well known optimal multiprocessor scheduler, as the underlying scheduling mechanism.

2. **PTG Scheduling on Heterogeneous Distributed Systems**
   Continuous demands for higher performance and reliability within stringent resource budgets is driving a shift from homogeneous to heterogeneous processing platforms for the implementation of today's CPSs. These CPSs are often distributed in nature and typically represented as PTGs due to the complex interactions between their functional components. This work considers the problem of scheduling a real-time system modeled as PTG, where tasks may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher *rewards*/QoS for the system. In this work, we propose *the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for scheduling a real-time PTG executing on a distributed platform consisting of a set of fully-connected heterogeneous processing elements.*

   First, we develop an ILP based optimal solution strategy namely, ILP-SATC, which follows an intuitive design flow and represents all specifications related to resource, timing and dependency, through a systematic set of constraints. However, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In order to improve scalability, a second strategy namely, ILP-SANC has been designed. ILP-SANC is based on the *non-overlapping approach* [9] which sets constraints and variables in such a way that no two tasks executing on the same processor overlap in time. Further, in ILP-SANC the total number of constraints required to compute a schedule for a PTG becomes independent of the deadline of a given PTG, which helps to control complexity of the proposed scheme. For example, given a PTG with seven tasks, each having two service-levels and executes on a distributed system consisting of 2 heterogeneous processors, ILP-SATC generates 7834 constraints and takes ∼2 seconds to find the optimal schedule. On the other hand, ILP-SANC generates only 203 constraints and takes 0.06 seconds to find the same solutions.

Though ILP-SANC shows appreciable improvements in terms of scalability over the ILP-SATC, it still suffers from high computational overheads (in terms of running time) as the number of nodes in a PTG and/or the number of resources, increase. For example, given a PTG with ∼20 tasks, each having three service-levels and executes on a distributed system consisting of 8 heterogeneous processors, ILP-SANC takes ∼4 hours to find the optimal schedule. It may be noted that such large time overheads may often not be affordable, especially when multiple quick design iterations are needed during design space exploration. Therefore, two low-overhead heuristics (i) G-SAQA and (ii) T-SAQA are proposed. Both G-SAQA and T-SAQA internally make use of PEFT [1], a well known PTG scheduling algorithm on heterogeneous multiprocessor systems, to compute a baseline schedule which assumes all task nodes to be at their base service-levels. Since PEFT attempts to minimize schedule length, the resulting schedule length may be marked by unutilized slack time before deadline.

The G-SAQA algorithm starts by using PEFT to compute task-to-processor mappings as well as start and finish times of tasks, based on task execution times associated with their base service-levels. If length of the obtained PEFT schedule violates deadline, then the algorithm terminates as generation of a feasible schedule is not possible. Otherwise, the available *global slack* ($slack_g = Deadline - PEFT\ makespan$) is used to enhance the tasks' assigned service-levels in an endeavour to maximize achievable reward while retaining task-to-processor mappings as provided by PEFT. The enhancement of task service-levels happen in a service-level by service-level manner, starting with all tasks situated at their base service-levels. At each step, the most eligible task is selected (from the task set) for service-level upgradation by one. The selection of this task is based on a prioritization key, which is the ratio between gain in rewards and increase in execution time to upgrade service-level from current to the next one.

Though G-SAQA follows an intuitive design flow, it only considers global slack ($= \textit{Deadline} - \textit{PEFT makespan}$) to upgrade service-levels of tasks in the PTG. However, a closer look at the PEFT schedule reveals that there exists gap within the scheduled nodes of the PTG which could be used along with the global slack to achieve better performance in terms of service-levels and delivered rewards compared to G-SAQA. It may also be possible to consolidate multiple small gaps within the PEFT schedule into larger consolidated slacks, which may be used to further improve performance in terms of achieved rewards. Therefore, the *total slack* available with a task at any given time comprises of the global slack along with the maximum consolidated inter-node gap between the task and its successor on its assigned processor in the PEFT schedule. With the above insights on the total task-level slacks available in a PTG, it proposes another heuristic namely, T-SAQA with the objective of achieving better performance compared to G-SAQA. The basic structure of T-SAQA is same as that of the G-SAQA algorithm except the way it updates the start times of selected task node's (selected for service-level enhancement) descendants and slacks associated with the task nodes in the PTG. In particular, G-SAQA uniformly delays the start times of all descendant nodes of the selected task and reduces the global slack value by the same amount. In this regard, it may be emphasized that T-SAQA works with distinct total slack values associated with the task nodes in the PTG, instead of using a single global slack pool. By harnessing the total slacks available with individual task nodes, T-SAQA updates the start and finish times of only those descendant task nodes of the selected task, whose start times are impacted due to the service-level upgradation of the selected task. Our simulation based experimental results show that both the heuristic schemes (G-SAQA and T-SAQA) are about $\sim 10^6$ times faster on an average than the optimal strategy ILP-SANC, when number of tasks in the PTG is $\sim$15, number of service-levels of each task is 3 and number of heterogeneous processors in the system is 8. It also shows that both T-SAQA and G-SAQA returns at most $\sim$30% and $\sim$45% less rewards than ILP-SANC, respectively. In all cases T-SAQA outperforms G-SAQA in terms of rewards maximization while T-SAQA has more running time than G-SAQA.

3. **PTG Scheduling on Heterogeneous Distributed Shared Bus Systems**
   The PTG scheduling technique considered in the previous section assumed a fully connected heterogeneous platform. Assumption of a fully connected platform helps to avoid the problem of resource contention, as is the case when the system is assumed to be associated with shared data transmission channels. However, it may be appreciated that shared bus networks form a very commonly used communication architecture in CPSs [41, 42]. Therefore, this work extend the problem of scheduling PTGs on fully-connected platforms, to CPS systems where the processors are connected through a limited number of bus based shared communication channels. In this work, we propose *the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for the scheduling of real-time PTGs executing on a distributed platform consisting of a set of heterogeneous processing elements interconnected by heterogeneous shared buses.*

   We first develop an ILP based solution strategy namely, ILP-ETR to produce optimal schedules for real-time PTGs executing on a distributed heterogeneous platform. ILP-ETR follows a comprehensive design approach, which represents all specifications related to resource, timing and dependency, through a systematic set of constraints. Although ILP-ETR follows an intuitive design flow, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In order to improve its scalability, we propose an improved ILP formulation namely, ILP-NC based on the *non-overlapping* *approach* [9] which sets constraints and variables in such a way that no two tasks executing on the same processor overlap in time. Experimental results show that ILP-ETR takes ∼5 hours to compute the schedule of a PTG with ∼20 nodes executing on a system with 4 processor and 2 buses, and the deadline of the PTG is set to its optimal makespan. On the other hand, ILP-NC takes only ∼12 secs to compute schedule for the same. Again, ILP-ETR is unable to find optimal solution within 24 hours when the deadline of the PTG is increased by 25% of its optimal makespan. In this case, ILP-NC takes only ∼12 seconds to find the optimal solution.

In addition to the two optimal ILP based approaches, we have designed a fast and efficient heuristic strategy namely, CC-TMS for the problem at hand. The CC-TMS is based on a *list scheduling* based heuristic approach to co-schedule task and message nodes in a real-time PTG executing on a distributed system consisting of a set of heterogeneous processors interconnected by heterogeneous shared buses. The algorithm assigns priority to all nodes in the PTG according to a parameter called, *upward rank* of each node. It then selects the highest priority task node and computes the task's *Earliest Finish Time* (EFT) values on each processor while temporarily allocating parent message nodes to suitable buses. The task is actually mapped on the processor where it has minimum EFT. Based on this selected task-to-processor mapping the parent message nodes of the task node are assigned to suitable buses. This process repeats until all task nodes are scheduled on processors. To evaluate the performance of the CC-TMS with respect to optimal solutions, we define a metric called *Makespan Ratio* as follows:

$$Makespan\ Ratio = \frac{Optimal\ Makespan}{Heuristic\ Makespan} \times 100 \tag{1.1}$$

Extensive simulation based experimental results show that CC-TMS achieves 97% and 58% (*Makespan Ratio*) in the best and worst case scenarios, respectively.

4. **Energy-aware Scheduling for Systems Consisting of Multiple PTG Applications**

The works done in the second and third phases deal with the co-scheduling of a single task graph on heterogeneous distributed platform. In the current phase, we endeavour towards the design of heterogeneous processor-shared bus co-scheduling strategies for a given set of independent periodic applications, each of which is modelled as a PTG. In particular, we have developed an ILP based optimal and heuristic strategy for the mentioned system model, whose objective is to minimize system level dynamic energy dissipation. Obviously to achieve energy savings, the processors in the system are assumed to be *Dynamic Voltage and Frequency Scaling* (DVFS) enabled and thus, the operating frequencies of these processors can be dynamically reconfigured to a discrete set of alternative voltage/frequency-levels at run-time. However, the ILP based optimal scheme called ILP-ES is associated with very high computational complexity and is not scalable even for small problem sizes. Therefore, we propose an efficient but low-overhead heuristic strategy called SAFLA which consumes drastically lower time and space complexities while generating good and acceptable solutions.

The SAFLA algorithm starts by using an efficient co-scheduling algorithm TMC which actually extend the CC-TMS algorithm (discussed above) to schedule multiple periodic PTGs executing on a shared bus-based heterogeneous distributed platform. This schedule is generated assuming all processor to be running at their highest frequency for the entire duration of the schedule. SAFLA terminates with failure, if the schedule returned by TMC violates deadline. Otherwise, the available slack associated with each task node is used to enhance the tasks' assigned voltage/frequency-levels in an endeavour to minimize energy dissipation, while retaining task-to-processor/message-to-bus mappings as provided by TMC. Experimental results show that SAFLA is an effective scheduling scheme and delivers handsome savings in terms of lower energy consumption in most practical scenarios.

1.5 Organization of the Thesis

The thesis is organized into eight chapters. A summary of the contents in each chapter is as follows:

- **Chapter 1**: *Introduction*
  This chapter is introductory, discussing the motivation of our work.

- **Chapter 2**: *Background on Real-time Systems*
  This chapter presents a background on real-time systems and various co-scheduling strategies of real-time tasks and messages on distributed multiprocessor platforms. In particular, we try to present the vocabulary needed to understand the following chapters.

- **Chapter 3**: *QoS Aware Scheduling of Independent Task Sets on Homogeneous Distributed Systems*
  In the third chapter, we propose strategies for co-scheduling a set of independent periodic tasks with multiple service-levels, executing on a bus-based homogeneous multiprocessor system. The problem is posed as a *Multi-dimensional Multiple-Choice Knapsack formulation* (MMCKP) and present a *Dynamic Programming* (DP) solution (called MMCKP-DP) for the same. Although DP delivers optimal solutions, it suffers from significantly high overheads (in terms of running time and main memory consumption), which steeply increase as the number of tasks, service-levels, processors and buses in the system grows, and severely restricts the scalability of the strategy. Therefore, in addition to the optimal solution approach MMCKP-DP, we propose an efficient but low-overhead heuristic strategy called ALOLA which not only consumes drastically lower time and space complexities but also generate good and acceptable solutions, which do not significantly deviate from the optimal solutions.

- **Chapter 4**: *Optimal Scheduling of PTGs on Heterogeneous Distributed Systems*
  Research conducted in the fourth chapter deals with the optimal scheduling mech-

anism of a real-time system modeled as PTG executing on a fully connected distributed heterogeneous platform. Here, tasks may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher *rewards*/QoS for the system. To solve the problem, an ILP based optimal solution approach namely, ILP-SATC, is proposed. Though the formulation of ILP-SATC follows an intuitive design flow, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In order to improve scalability, a second ILP based strategy namely, ILP-SANC, has been designed. Instead of explicitly relying on task mobility based manipulations as ILP-SATC, ILP-SANC guarantees that the executions of no two tasks in the system overlap in time on the same processor. This modification in the design approach allows the constraint set in ILP-SANC to be independent of the deadline of a given PTG.

- **Chapter 5**: *Heuristic PTG Scheduling Strategies on Heterogeneous Distributed Systems*
  Though ILP-SANC in (Chapter 4) shows appreciable improvements in terms of scalability over the ILP-SATC, it still suffers from high computational overheads (in terms of running time) as the number of nodes in a PTG and/or the number of resources, increase. Therefore in the fifth chapter, two low-overhead heuristic algorithms namely, G-SAQA and T-SAQA, are proposed for the same problem as discussed in the previous (fourth) chapter. The base-line heuristic, G-SAQA, is faster but returns moderately good solutions. T-SAQA extends G-SAQA and deliver significantly better solution, albeit at the cost of slightly higher time complexity.

- **Chapter 6**: *PTG Scheduling on Shared-Bus Based Heterogeneous Platforms*
  In this chapter, we propose the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for the co-scheduling of real-time PTGs executing on a distributed platform, consisting of a set of heterogeneous processing elements interconnected by heterogeneous shared buses. To solve the problem, two

ILP based strategies namely, ILP-ETR and ILP-NC are proposed. Although, both the approaches produce optimal solutions, ILP-NC suffers significantly lower computational overheads compared to ILP-ETR. In addition to the optimal solution approaches, we propose a fast but effective heuristic strategy called CC-TMS which consumes much lower time and space complexities, while producing satisfactory solutions.

- **Chapter 7**: *Scheduling Multiple Independent PTG Applications on Shared-Bus Platform*
  Chapter 7 deals with the energy-aware co-scheduling of multiple periodic PTGs executing on a distributed platform consisting of heterogeneous processing elements and interconnected through a set of heterogeneous shared buses. An ILP based optimal scheduling strategy namely, ILP-ES is proposed to minimize the overall system-level energy dissipation. Further, an efficient, low-overhead heuristic strategy called SAFLA has been proposed for the problem at hand.

- **Chapter 8**: *Conclusion and Future Work*
  The thesis concludes with this chapter. A comparative analysis on the algorithms presented in the different contributory chapters has been carried out. We discuss the possible extensions and future works that can be done in this area.

Chapter 2

Background and Related Work

This dissertation is oriented towards the design of real-time task-message co-scheduling strategies for safety-critical CPSs. The previous chapter provided a background of the overall problem domain and also highlighted several challenges imposed by diversity in the nature of available computing/communication platforms, resource constraints, timeliness, performance requirements, etc. that the developed solution strategies must adhere to.

In this chapter, we present a brief background as well as state-of-the-art related to different types of real-time systems followed by various real-time scheduling strategies for CPSs. We first provide an overview on the structure of real-time systems. Then, the evolution of scheduling algorithms for real-time systems implemented on homogeneous and heterogeneous multiprocessor systems are discussed.

2.1 An Overview of Real-time Systems

Typically, real-time systems are composed of the *Application Layer*, *Real-time Scheduler* and *Hardware Platform* [43].

- **The Application Layer** consists of all applications that should be executed.
- **The Real-time Scheduler** takes scheduling decisions and provides services to the Application Layer.

- **The Hardware Platform** consists of processors, memories, communication networks, etc. on which the applications are executed.

We will now present each of these layers in detail and introduce important theoretical models for enabling the analysis of these systems and allow the design of efficient scheduling strategies.

2.1.1 Application Layer

The application layer is composed of all the applications that the system needs to execute. Applications in real-time systems often consist of a set of *recurrent* tasks. Each such task may represent a piece of code (i.e., program) which is triggered by external events that may happen in their operating environment. Each execution instance of the task is referred to as a *job*. We now discuss the set of definitions related to a real-time task.

2.1.1.1 Real-time Task Model

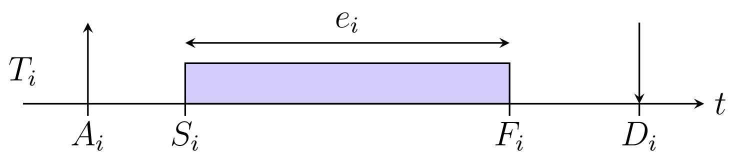


Figure 2.1: *Typical parameters of a real-time task*

Formally, a real-time task (denoted by T_i; shown in Figure 2.1) can be characterized by the following parameters:

1. **Arrival time** (A_i) is the time at which a task becomes ready for execution. It is also referred as *release time* or *request time* of the task.

2. **Start time** (S_i) is the time at which a task starts its execution.

3. **Computation time** or *Execution time* (e_i) is the time taken by the processor to finish computation of the task without interruption.

4. **Finishing time** or *Completion time* (F_i) refers to the time at which a task finishes its execution.

5. **Deadline** (D_i) is the time before which a task should complete its execution without causing any damage to the system. If a deadline is specified with respect to the task arrival time, it is called a *relative deadline*, whereas if it is specified with respect to time zero, it is called an *absolute deadline.*

6. **Worst-case execution time** is the largest computation time of a task among all of its possible executions.

7. **Laxity** or *Slack time* ($slack_i$) is the maximum amount of time by which execution of a task can be delayed after its activation/arrival, to complete within its deadline ($slack_i = D_i - e_i$).

8. **Priority** is the importance given to a task in context of the schedule at hand.

9. **Criticality** is a parameter related to the consequences of missing a deadline (typically, it can be *hard*, *firm*, or *soft*).

A real-time task T_i can be classified as periodic, aperiodic or sporadic based on regularity of its activations [14, 27].

1. **A Periodic** task consists of jobs that arrive strictly periodically, separated by a fixed time interval π_i.

2. **A Sporadic** task consists of jobs that may arrive at any time once a minimum interarrival time has elapsed since the arrival of the previous job of the same task.

3. **An Aperiodic** task consists of jobs where there is no regularity in the interarrival times of consecutive jobs.

There are three levels of constraints with respect to the placement of deadlines relative to the repetition periodicity of periodic and sporadic tasks.

1. *Implicit Deadlines:* All task deadlines are equal to their periods ($D_i = \pi_i$).

2. *Constrained Deadlines:* All task deadlines are less than or equal to their periods ($D_i \leq \pi_i$).

3. *Arbitrary Deadlines:* Task deadlines may be less than, equal to, or greater than their periods.

In this dissertation, we deal with periodic tasks having implicit deadlines. Now, we provide a few other definitions related to tasks to be executed in a real-time system.

- *Utilization*: The utilization of an implicit deadline task T_i is given by $U_i = e_i/\pi_i$. The total utilization of a task set $T = \{T_1, T_2, \ldots, T_n\}$ is defined as, $U = \sum_{i=1}^{n}(e_i/\pi_i)$.
- *Static and Dynamic Task System*: In a static task system, the tasks that will execute on the platform are completely defined before start of the system. In a dynamic task system, tasks may join, leave or get modified during run-time. *In this thesis, we only deal with static task systems.*
- *Hyperperiod ($\mathcal{H}$)*: Given a static task system, $\mathcal{H}$ represents the minimum time interval after which the schedule repeats itself. For a set of periodic tasks, $T = \{T_1, T_2, \ldots, T_n\}$ with periods $\{\pi_1, \pi_2, \ldots, \pi_n\}$, hyperperiod is given by the *Least Common Multiple* (LCM) of the periods ($\mathcal{H} = LCM(\pi_1, \pi_2, \ldots, \pi_n)$).

2.1.2 Real-time Scheduler

A real-time scheduler acts as an interface between applications and the hardware platform. It schedules tasks using a real-time scheduling algorithm. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm.* Given a set of tasks, $T = \{T_1, T_2, \ldots T_n\}$, a *schedule* is an assignment of tasks to the processors in the platform, so that each task may be executed until completion. A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints. A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule. In a *work-conserving* scheduling algorithm, processors are never kept idle while there exists a task waiting for execution.

2.1.3 Hardware Platform

Based on the number of processors, a system can be classified into *uniprocessor* and *multiprocessors* systems. A *processor* is a hardware element (digital circuit) that executes programs or tasks.

1. **Uniprocessor** systems can execute only one task at a time and must switch between tasks.
2. **A Multiprocessor** system may range from several separate uniprocessors tightly coupled using high speed networks to multi-core. It can be classified as follows:
   (a) **Homogeneous:** The processors are identical; hence the rate of execution of all tasks is the same on all processors.
   (b) **Uniform:** The processors are architecturally identical, but may execute at different clock speeds (operation frequencies). Thus the rate of execution of a task depends only on the speed of the processor. A processor running at speed say, 2 GHz, will execute all tasks at exactly twice the rate of a processor executing at speed 1 GHz.
   (c) **Heterogeneous:** The processors are different; hence the rate of execution of a task depends on both the processor and the task. That is, given a task set, execution rates of a task on the different processors of the platform, are completely unrelated to the execution values exhibited by other tasks. Indeed, not all tasks may be able to execute on all processors.

In this dissertation, we have considered homogeneous/heterogeneous multiprocessor systems where processors are either fully-connected or communicate through a shared homogeneous/heterogeneous bus-based network.

1. **Fully-connected multiprocessor system:** In this system, each pair of processors have dedicated communication channel to send/receive data or messages. There is no contention for communication resources to transmit data among processors. Figure 2.2 shows a typical fully-connected multiprocessor system where

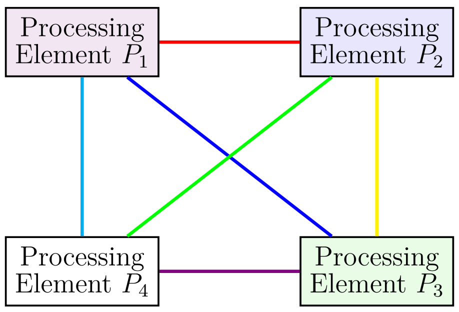


Figure 2.2: *Fully connected multiprocessor system*

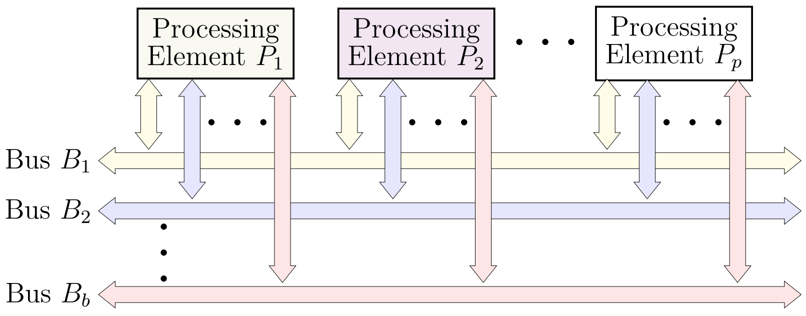


Figure 2.3: *Shared bus-based multiprocessor system*

all processors/processing elements are connected to each other through dedicated links. For example, processing element P_1 has separate communication channels to processing elements P_2, P_3 and P_4.

2. **Shared bus-based multiprocessor system:** In this system, processors are connected through shared bus-based communication network. Here, processors can be connected to all buses or a subset of buses. Further, buses can be homogeneous or heterogeneous in nature. Figure 2.3 shows a shared bus-based multiprocessor system. For example, processing element P_1 is connected to all buses $B_1, B_2, \ldots, B_b$.

We assume, each processor has its own private memory. For any given processor, task execution and communication with other processors can be conducted simultaneously, without any contention. Specifically, we assume that tasks on different processors communicate by transmitting data from the source processor via the underlying communication network, to the local memory of the receiving processor. On the other hand, intra-processor communication is realized through the reading and writing of variables stored in the local memory of the processor.

2.2 Types of Task Constraints:

Typical constraints that can be specified on real-time tasks are of three classes: timing constraints, precedence relations, and constraints on shared resources.

1. **Timing Constraints:** Real-time systems are characterized by computational activities with stringent timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a task is the deadline. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in three categories:

   – **Hard:** A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the system under control.

   – **Firm:** A real-time task is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value.

   – **Soft:** A real-time task is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation.

2. **Precedence Constraints:** In certain applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage. Such precedence relations are usually described through a *Directed Acyclic Graph* (DAG)/*Precedence-constrained Task Graph* (PTG), where tasks are represented by nodes and precedence relations by arrows. A PTG induces a partial order on the task set.

   – The notation $T_i \prec T_j$ specifies that task T_i (T_j) is an ancestor (a descendant) of task T_j (T_i), meaning that the PTG contains a directed path from node T_i to node T_j.

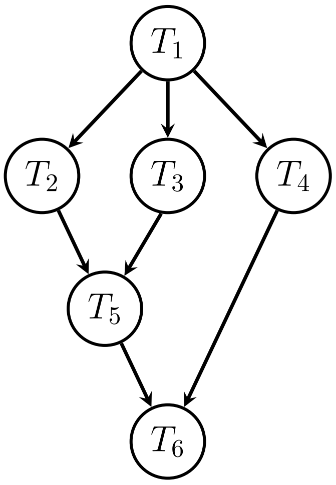


Figure 2.4: *A Precedence-constrained Task Graph (PTG)*

– The notation $T_i \rightarrow T_j$ specifies that task T_i (T_j) is a predecessor (successor) of task T_j (T_i), meaning that the PTG contains an arc directed from node T_i to node T_j.

Figure 2.4 illustrates a PTG that describes the precedence constraints among six tasks. From the figure, it can be observed that task T_1 do not have any predecessor and can immediately start its execution. Tasks with no predecessors are called source/start task nodes. It can also be seen that T_2, T_3 and T_4 can start executing only after the predecessor task node T_1 completes its execution. Tasks with no successors, as T_6 in the above figure, are called sink task node. A PTG can have multiple source or sink nodes.

3. **Resource Constraints:** The hardware platform in a real-time system consists of a limited number of resources which are shared among multiple applications. So, the resources must be used in a mutually exclusive way. For example, multiple tasks can not execute on the same processor at a single time instant, i.e., a processor can execute at most one task at a moment. Similarly, a bus can transmit only a single message at any time instant.

2.3 Classification of Real-Time Scheduling Algorithms

Among the great variety of algorithms proposed for scheduling real-time tasks, the following main classes can be identified:

- Preemptive Vs. Non-preemptive.
  - In preemptive algorithms, a running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy. The unfinished portion of the interrupted task may be re-allocated to the same processor or to a different processor [44].
  - In non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.
- Static vs. Dynamic.
  - Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
  - Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.
- Off-line vs. Online.
  - In *off-line* scheduling, the scheduler has a priori knowledge of the task set and its constraints, such as arrival times, execution times, precedence constraints, etc. The schedule is generated and stored at design time and dispatched later during runtime of the system. Static schedulers are typically off-line in nature.
  - In *online* scheduling, algorithms make their scheduling decisions at runtime based on information about the tasks that have arrived so far. Although they are often flexible and adaptive, they may incur significant overheads because

of runtime processing. However, they are a must in systems which do not have enough information before run-time to execute the scheduler statically.

- Optimal vs. Heuristic.
  - An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it is able to find a feasible schedule, if one exists.
  - An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends towards the optimal schedule, but does not guarantee finding it.

2.4 A Discussion with Motivational Examples

This thesis deals with *Real-Time Cyber-Physical System* (RT-CPS) as its target domain. RT-CPS applications are usually dedicated towards controlling physical plants. The applications execute persistently in infinite loops periodically acquiring data from the plant/environment through sensors, processing the same, and then generating appropriate actuation data. These control applications are often complex and executed on high capacity compute servers at locations which may be geographically distant from the plant. The sensors and actuators on the other hand, are in general co-located with the plant. In this scenario, both the sensory and actuation data must be transmitted as messages in a timely fashion over networks. The overall problem therefore involves two distinct resource management issues — that of managing computation resources for application execution and managing communication resources for message transmission.

Large systems such as today's distributed manufacturing systems may constitute multiple control sub-systems running on distributed processing platforms. Control applications in most of these sub-systems are modeled as real-time *independent tasks* or PTGs, depending on the nature of interactions between their functional components. These tasks often execute in a federated fashion on dedicated (separate) processing units in order to keep the design methodology simple, while also meeting specifications related to timing, reliability, energy, etc. However, it may be noted that such federated execution generally leads to poor utilization of resource capacities due to lack of resource sharing among tasks and/or messages. Poor resource utilization in turn, can result in higher design costs as more resources must be deployed to synthesize a given system, than is otherwise necessary. On the other hand, executing tasks on consolidated architectures can reduce design costs, although it can cause to significantly increased design complexity due to a higher degree of contention for shared resources.

A scheduler allocates resources to the tasks and decides their execution sequence (start times) taking schedulability constraints into consideration. To schedule a set of *independent* tasks, a scheduler needs to satisfy deadline and resource constraints along with precedence relationships associated with sensing and actuation messages. On the other hand to schedule a set of *dependent* tasks (represented as a PTG), a scheduler additionally needs to satisfy the precedence constraints among tasks. A significant amount of work exist in the literature on scheduling of PTGs on distributed systems [1, 6, 9, 11–13, 32]. Most of the existing schemes assume full interconnection between the processing elements in the execution platform. Distributed processing systems formed by local area wireless networks such as ZigBee [45] are practical examples of such execution platforms. Today, there is an increasing trend towards the execution of real-time applications over such distributed local area networks in scenarios such as an Industry 4.0 based smart manufacturing plant. The applications here are typically complex interdependent work flows usually represented as PTGs. However, there is a severe dearth of efficient resource allocation techniques for executing real-time PTGs on the distributed platforms as mentioned above.

Although fully connected platforms find some practical use cases, there exist a vast majority of distributed systems where the processing elements are not fully interconnected by dedicated communication links. A shared bus-based automotive RT-CPS can be considered as a typical example of such a distributed platform. It may be noted that real-time scheduling techniques for this scenario must tackle contention for shared communication channels, in addition to simultaneously handling the contention of tasks attempting to execute on a shared pool of processing elements.

Today, scheduling of tasks and messages in real-time distributed systems is usually carried out in a federated manner in two steps. The first step among them is dedicated to the allocation and scheduling of tasks on processing elements. Given the mapping and execution order of tasks (obtained through the first step), the second step is dedicated to the scheduling of messages between tasks while taking care that all schedulability constraints are satisfied. This separation of concerns between task and message scheduling allows the adaption of simpler design methodologies, and hence has been employed as the usual practice. However, it may be noted that integrated scheduling of tasks and messages have the potential to provide improved optimization, leading to possibly higher resource usage efficiencies and lower deployment costs. With this motivation, *this thesis has attempted towards the design of task-message co-scheduling strategies for both independent tasks as well as PTGs, on fully-connected and shared-bus based distributed platforms.*

Scheduling problems considered in this thesis are generally complex and NP-Hard in nature. We will now consider a series of small test case scenarios of progressively higher complexities and present important scheduling challenges that arrive for each of them. First, let us consider a set of five *independent* tasks T_1, T_2, T_3, T_4 and T_5 to be executed on a *homogeneous* platform consisting of two processors P_1 and P_2. Table 2.1 shows execution times of these five tasks. All tasks have the same deadline of 14. Based

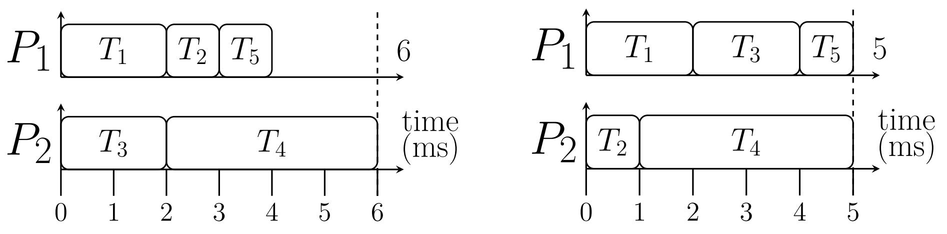
T_1	T_2	T_3	T_4	T_5
2	1	2	4	1

Table 2.1: *Execution times of tasks on homogeneous processors*

on variations in task-to-processor allocation decisions, there can be multiple possible schedules with different makespans. For example, Figures 2.5a and 2.5b show two work-conserving[1] schedules having makespans 6 (slack time $14 - 6 = 8$) and 5 (slack time

[1]In a *work-conserving* scheduling algorithm, processors are never kept idle while there exists a task

$14 - 5 = 9$) respectively, for the given task set.



(a) *Schedule 1* **(b)** *Schedule 2*

Figure 2.5: *Gantt chart: Independent tasks (Table 2.1); Homogeneous processors*

We can observe from the above example that based on variations in task-to-processor allocation decisions, there can be multiple schedules with different makespans. A schedule with lower makespan is always beneficial as it allows higher slack times which may be used to better optimize one or more performance metrics such as expenditure on resources, energy, reliability, security, etc.

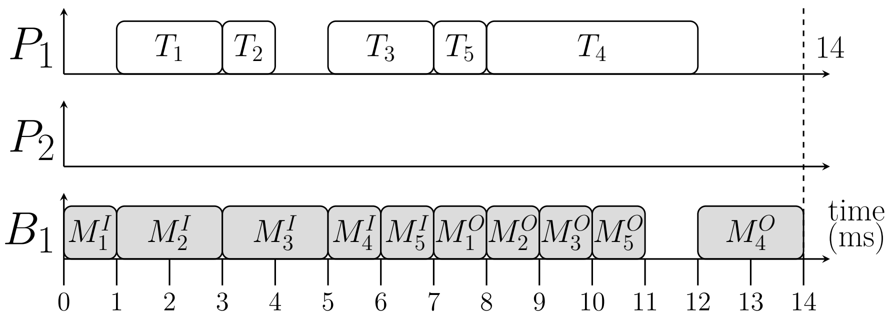
The first test case scenario does not consider message communication. Next, we consider the case when each task takes a sensory input and produces an actuation output. The sensed inputs are transmitted as messages over communication channels, to processors where tasks get executed and the corresponding control outputs are computed. The computed outputs in turn are communicated to actuators as messages via communication channels. Let us consider the same set of five *independent* tasks T_1, T_2, T_3, T_4 and T_5 executing on a *homogeneous* distributed platform consisting of two processors P_1 and P_2. Tables 2.1 and 2.2 show the tasks' execution times and their corresponding input/output messages. Here, M_i^I and M_i^O represent input and output messages of a task T_i. For simplicity, we assume that the bandwidth of the communication channels among processors in this example is of unit capacity. So, the communication time of messages will become same as their sizes. Here also, all tasks have the same deadline 14.

Depending on variations in task-to-processor allocation decisions, there can be multiple possible schedules with different makespans. For example, Figures 2.6a and 2.6b show two schedules having makespans 14 and 13 respectively, for the given task set and

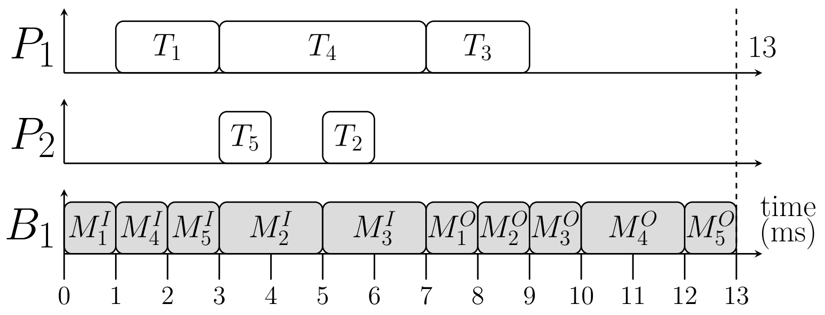
waiting for execution.

Input Messages					Output Messages				
M_1^I	M_2^I	M_3^I	M_4^I	M_5^I	M_1^O	M_2^O	M_3^O	M_4^O	M_5^O
1	2	2	1	1	1	1	1	2	1

Table 2.2: *Transmission times of input and output messages*



(a) *Schedule 1*



(b) *Schedule 2*

Figure 2.6: *Gantt chart: Independent tasks (Table 2.1) with associated messages (Table 2.2); Homogeneous processors*

associated messages. The generated schedules also show that each task starts only after it receives the corresponding input message. Similarly, the output message starts its transmission after the respective task finishes its execution. In Chapter 3 (contribution 1) we present solution strategies for a system model very similar to the second test case scenario.

In the third case scenario, we consider PTG as the application model. Let us consider a set of five *dependent* tasks represented as a PTG in Figure 2.7, to be executed on the same two processors *homogeneous* platform. In the PTG, $T_1 - T_5$ represent tasks and $M_1 - M_6$ represent messages. Tables 2.1 and 2.3 show execution times of tasks and message sizes/transmission times, respectively. End-to-end deadline of the PTG is assumed to be 14. There can be multiple possible schedules with different makespans

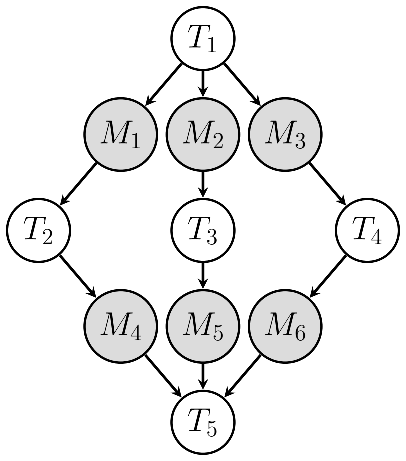
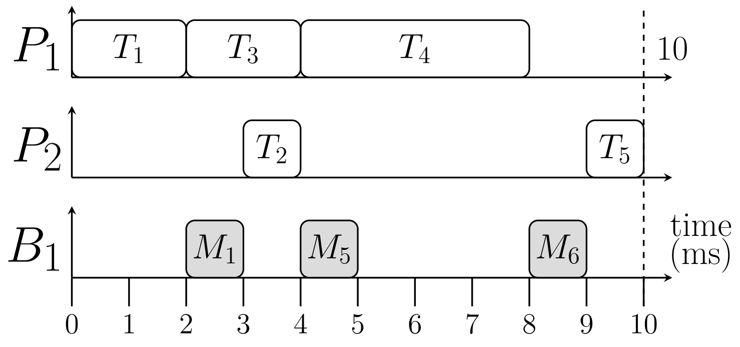


Figure 2.7: *PTG with task and message nodes*

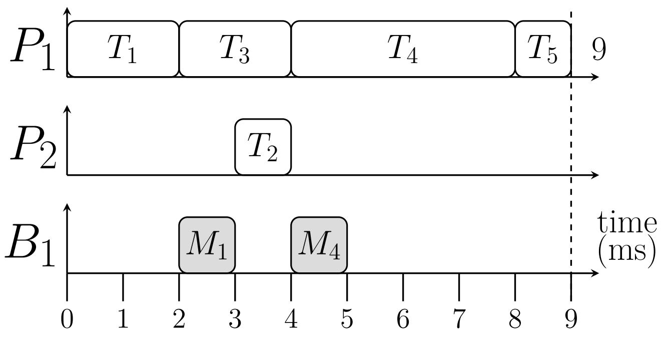
M_1	M_2	M_3	M_4	M_5	M_6
1	2	2	1	1	1

Table 2.3: *Transmission times of message nodes in the PTG shown in Figure 2.7*

based on variations in task-to-processor allocation decisions. For example, Figures 2.8a and 2.8b show two work-conserving schedules having makespans 10 and 9, respectively. If both the parent and child tasks T_i and T_j of a message M_k are scheduled on the same processor, then there is no need to transmit the message M_k over the communication channel. Hence, in this case the transmission time associated with message M_k becomes zero. For example in Figure 2.8a, the message M_2 is not transmitted as both parent and child tasks T_1 and T_3 are scheduled on the same processor P_1. From the same figures, it can also be seen that precedence among task and message nodes in the PTG are preserved. For example (in Figure 2.8a), task T_3 starts after the predecessor task T_1 finishes its execution. Similarly, task T_2 starts after the predecessor message M_1 finishes its transmission.



(a) *Schedule 1*



(b) *Schedule 2*

Figure 2.8: *Gantt chart: PTG (Figure 2.7) with tasks (Table 2.1) and message (Table 2.3) nodes; Homogeneous processors*

Today, continuous demands for higher performance and reliability within stringent resource budgets, is driving a shift from *homogeneous* to *heterogeneous* processing platforms for the implementation of CPSs. Depending on processor types, a task may require same or distinct execution times on different processors, and this must be correctly accounted in the generated schedule. In the fourth test case scenario, we consider the two processors P_1 and P_2 to be *heterogeneous*. Table 2.4 shows the execution times of five task nodes (of the PTG in Figure 2.7) on P_1 and P_2. Similarly, Table 2.3 shows the communication times of message nodes. We consider the deadline of the PTG to be 14.

Figures 2.9a and 2.9b show two work-conserving schedules having makespans 11 and 9 respectively, for the given PTG. From the figures, it can be seen that each task has consumed distinct amounts of time depending on the processor on which it is assigned. For example (in Figure 2.9a), task T_2 has taken 3 time units on processor P_2.

	T_1	T_2	T_3	T_4	T_5
P_1	2	1	2	4	1
P_2	3	3	1	3	2

Table 2.4: *Execution times of tasks on two heterogeneous processors*

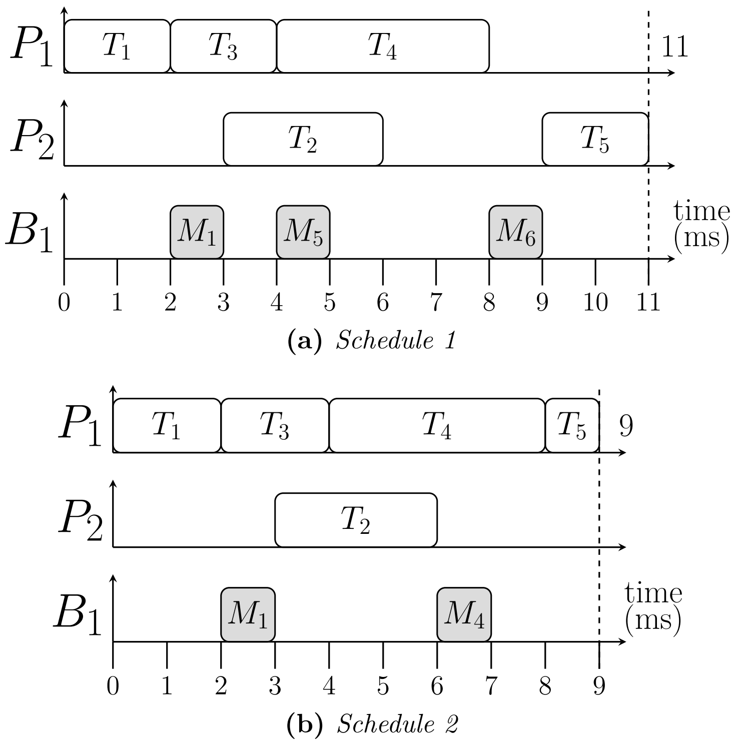


Figure 2.9: *Gantt chart: PTG (Figure 2.7) with tasks (Table 2.4) and message (Table 2.3) nodes; Heterogeneous processors*

A significant amount of work is done on the problem of scheduling PTGs executing on various distributed platforms [1, 6, 9, 11–13, 32]. Most of these works assumed that processors are fully-connected. That is each pair of processors have dedicated communication channel to send/receive data or messages. In the fifth test case scenario, we consider a distributed platform which is fully-connected. Let us consider the same PTG (Figure 2.7) as the previous test case. However, we now consider a fully-connected *heterogeneous* distributed platform consisting of three processors P_1, P_2 and P_3. Tables 2.5 and 2.3 show the execution and transmission times of tasks and messages. Figures 2.10a

	T_1	T_2	T_3	T_4	T_5
P_1	2	1	2	4	1
P_2	3	3	1	3	2
P_3	1	2	3	2	3

Table 2.5: *Execution times of tasks on three heterogeneous processors*

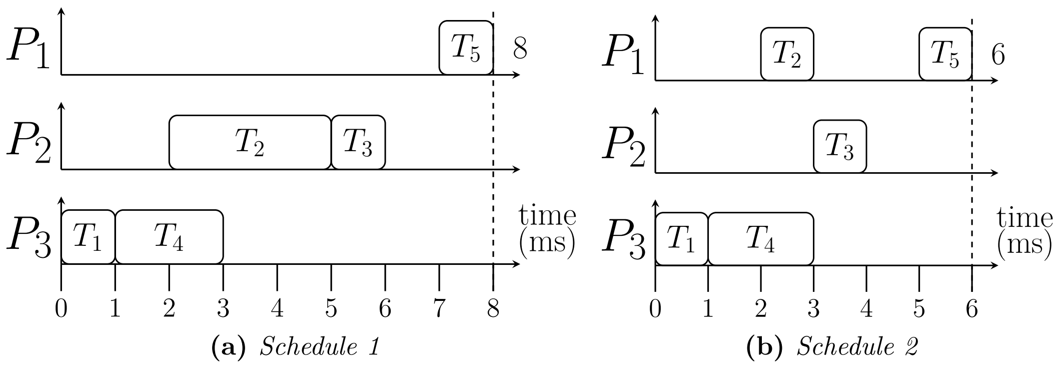


Figure 2.10: *Gantt chart: PTG (Figure 2.7) with tasks (Table 2.5) and message (Table 2.3) nodes; Heterogeneous processors; Fully-connected*

and 2.10b show two work-conserving schedules having makespans 8 and 6 respectively, for the given PTG. From the same figures, it can also be seen that when both parent and child tasks are scheduled on different processors, the child task has to wait for the corresponding message transmission. For example (in Figure 2.10a), task T_2 has to wait for 1 time unit for the transmission of message M_1 through the dedicated communication channel between processors P_3 and P_2. Chapters 4 and 5 address the problem of scheduling a PTG on system model very similar to the fourth and fifth test case scenario.

In the previous test case scenario, we have assumed a distributed platform where processors are fully-connected. However, there exist a vast majority of distributed systems where processors are connected through shared-bus based communication channels [41, 42]. Real-time scheduling techniques for this scenario must tackle contention for shared communication channels, in addition to simultaneously handling the contention of tasks attempting to execute on a shared pool of processing elements. In the sixth test case scenario, we consider a distributed platform where processors are connected through shared-bus based communication medium. Let us consider the same PTG in Figure 2.7 having deadline 14 and executing on a *heterogeneous* distributed platform consists of three processors P_1, P_2 and P_3 connected through a shared bus B_1. Tables 2.5 and 2.3 show the execution times of tasks and transmission times of messages, respectively. Figures 2.11a and 2.11b show two work-conserving schedules having

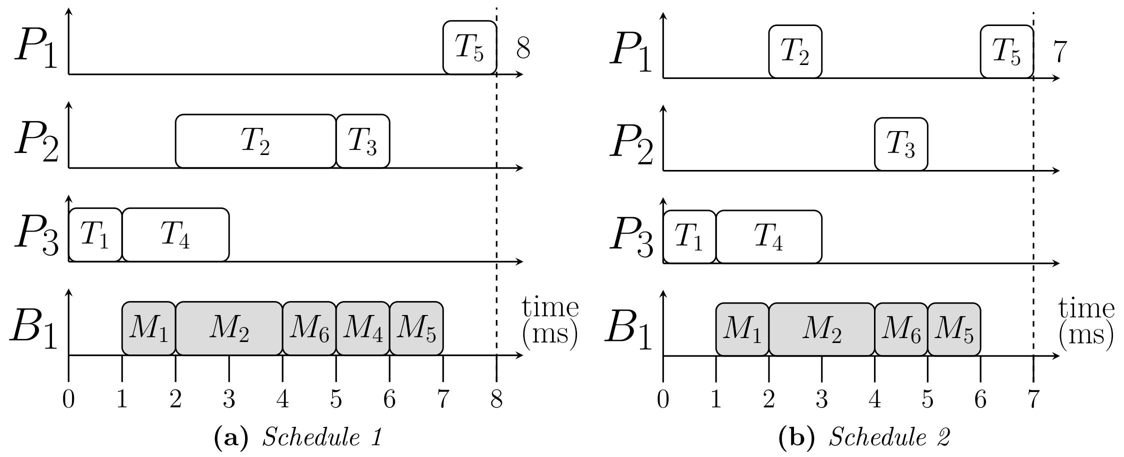


Figure 2.11: *Gantt chart: PTG (Figure 2.7) with tasks (Table 2.5) and message (Table 2.3) nodes; Heterogeneous processors; Shared bus*

makespans 8 and 7 respectively, for the given PTG. As all the three processors are connected through a single shared bus, depending on schedule of messages on the bus, a task may have to wait for the transmission of other messages which are irrelevant to it. For example in Figure 2.11b, task T_3 has to wait for the transmission of message M_1 which is irrelevant to it, in addition to M_2. In the Chapter 6, we consider a similar platform model for PTG scheduling and present solution for the same.

In the above discussions, we have shown the scheduling of a set of independent tasks or a single PTG executing on various types of distributed platforms. However, there can be multiple co-executing control functionalities in the system, with each such functionalities being represented as a PTG. These control functionalities repeat/execute periodically. The seventh (also the final) test case scenario considers the problem of scheduling multiple PTGs on a shared-bus based distributed platform. Let us consider two PTGs G^1 and G^2 (Figure 2.12) to be executed on a shared bus-based *heterogeneous* distributed platform consisting of three processors (P_1, P_2 and P_3) and one bus (B_1). Here, T_i^g represents i^{th} task node and M_k^g represents k^{th} message node of the PTG G^g. Tables 2.6 and 2.7 show execution and communication times of tasks and messages, respectively, in PTGs G^1 and G^2. Both PTGs G^1 (Figure 2.12a) and G^2 (Figure 2.12b) have implicit deadlines $D^1 = 14$ and $D^2 = 7$, respectively. So, hyperperiod for these two co-executing tasks become 14. Within this hyperperiod we have one instance of G^1 and two instances of G^2. We determine a schedule for these two PTGs over one hyperperiod. The determined schedule will repeat every hyperperiod. Figures 2.13a and 2.13b show

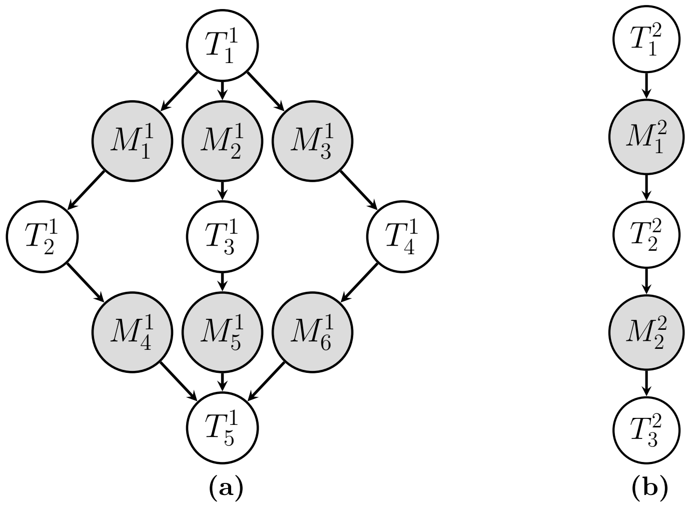


Figure 2.12: *(a) PTG G^1, Deadline $D^1 = 14$; (b) PTG G^2, Deadline $D^2 = 7$*

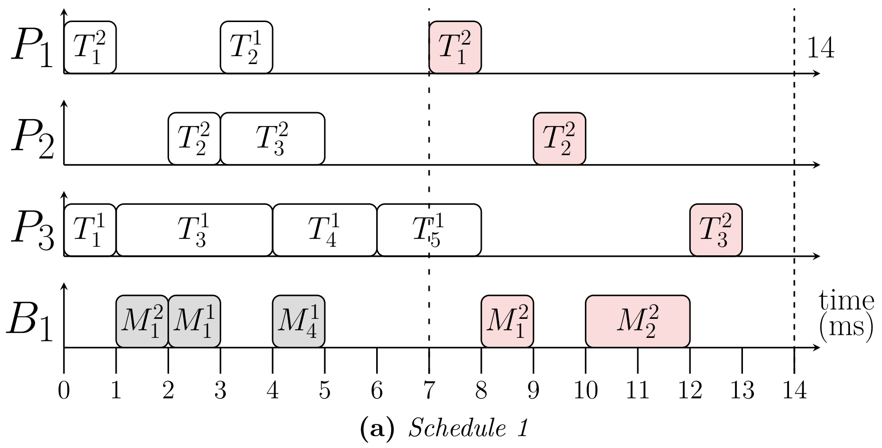
Processor	PTG G^1					PTG G^2		
	T_1^1	T_2^1	T_3^1	T_4^1	T_5^1	T_1^2	T_2^2	T_3^2
P_1	2	1	2	4	1	1	2	3
P_2	3	3	1	3	2	2	1	2
P_3	1	2	3	2	3	3	3	1

Table 2.6: *Execution times of task nodes in PTGs G^1 and G^2*

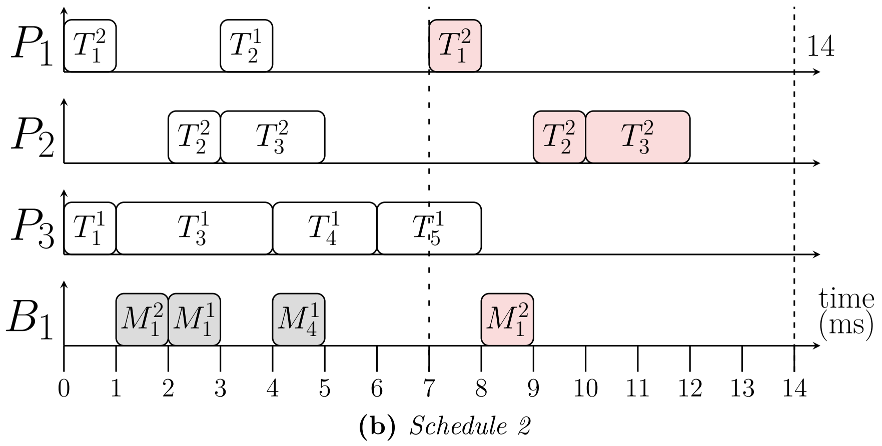
two different work-conserving schedules for PTGs G^1 and G^2 executing on the same distributed platform. We can see that a task of a PTG at different iterations within

Bus	PTG G^1						PTG G^2	
	M_1^1	M_2^1	M_3^1	M_4^1	M_5^1	M_6^1	M_1^2	M_2^2
B_1	1	2	2	1	1	1	1	2

Table 2.7: *Transmission times of message nodes in PTGs G^1 and G^2*



(a) *Schedule 1*



(b) *Schedule 2*

Figure 2.13: *Gantt chart: Multiple PTGs (Figure 2.12; Tables 2.6 & 2.7); Heterogeneous processors; Shared bus*

the hyperperiod may be assigned to different processors and it takes distinct execution times to compute on its assigned processor. For example in Figure 2.13a, the task T_3^2 is assigned on processor P_2 in its first iteration and on P_3 in the second iteration. T_3^2 takes 2 time units on P_2 and 1 time unit on P_3. In Chapter 7, we consider the problem of scheduling multiple independent PTGs on a shared bus-based distributed platform.

2.5 Multiprocessor Scheduling - A Brief Survey

This dissertation deals with the co-scheduling of tasks and messages for: (1) Mutually independent task sets, (2) *Precedence-constrained Task Graphs* (PTGs), (3) Mutually independent applications, each represented as a PTG. Traditionally, scheduling of real-time *independent* tasks on multiprocessor systems has been based on either partitioned or global approaches [14–16]. With partitioning, the multiprocessor scheduling problem is transformed into a set of uniprocessor scheduling problem, where each task is assigned to a designated processor and gets executed entirely on that processor. Well known optimal uniprocessor scheduling algorithms include RM (static priority) and EDF (dynamic priority), proposed by Liu and Layland [17]. RM is a preemptive, static priority scheduling algorithm where all jobs of a task have the same priority. It assigns higher priorities to tasks with shorter periods or higher repetition rates. EDF is a preemptive, dynamic priority scheduling algorithm. Tasks with earlier (absolute) deadlines are executed with higher priorities. A set of periodic tasks is schedulable with EDF if and only if, $U \leq 1$. With partitioning, as each task runs only on a single processor, there is no penalty due to inter-processor task migrations. However, a major drawback of *partitioning* is that, the problem is provably *NP-hard* for both homogeneous and heterogeneous multi-cores [46], and up to half of the system capacity may remain unutilized in order to ensure timing constraints of a given task set [18].

Unlike the fully partitioned approaches, more global schedulers like Pfair [19], PD^2 [20], ERfair [21], BF [22], *SA* [23], LLREF [24], RUN [25], DP-Fair [26], can achieve very high utilization of the system capacity by allowing migrations of tasks among processors. The Pfair scheduler proposed by Baruah et al. [19] is known to be the first optimal global real-time scheduler on multiprocessor systems for tasks with implicit deadlines. Based on Pfair, Anderson et al. [21] proposed a work-conserving multiprocessor scheduling algorithm called the ERfair scheduler. Given a set of n implicit-deadline periodic tasks to be executed on p *homogeneous* processors, ERfair ensures that the minimum rate of progress for each task is proportional to a parameter called the task's weight wt_i, which is defined as the ratio of it's execution requirement e_i and period π_i. That is, in order to satisfy ERfairness, each task T_i should complete at least $(\sum_{i=1}^{n}(e_i/\pi_i) \times t)$ part of it's total execution requirement e_i, at any time instant t, subsequent to the start of the task at time S_i. Following the above criterion, ERfair is able to deliver optimal/full resource utilization; hence, any task set is guaranteed to be schedulable if Equation 2.1 is satisfied.

$$\sum_{i=1}^{n}(e_i/\pi_i) \leq p \tag{2.1}$$

However both the above schemes attempt to maintain proportional fairness at each time slot and incur unrestricted preemption/migration overheads due to this. Recently, Levin et al. [26] proposed a semi-partitioned approximate proportional fair optimal scheduler called DP-Fair with much lower and bounded context switching overheads. DP-Fair is an optimal two level hierarchical homogeneous multi-resource scheduler. At the outer level, time is partitioned into *slices*, demarcated by the periods/deadlines of all jobs in the system. At the inner level, within a given *time slice* of length t time slots (say), each task T_i is allocated a workload equal to its proportional fair share, $shr_i = (e_i/\pi_i) * t$ and assigned to one or more resources (processor/bus resource) for scheduling. DP-Fair is an optimal algorithm which ensures full resource utilization. Given a set of n tasks to be scheduled on p *homogeneous* resources, DP-Fair guarantees a feasible schedule if Equation 2.1 is satisfied. Using a combination of techniques including *DP-Wrap*, *mirroring* and *fairness-oblivious intra-slice execution*, DP-Fair is able to ensure atmost $n-1$ context switches and $p-1$ migrations per time slice [26]. These facets make DP-Fair a lucrative resource allocation technique which can efficiently combine the twin benefits of *high resource utilization* and *low scheduling related overheads*. However, the above schemes can not handle the combined scheduling of tasks and messages as required for *Real-Time Cyber-Physical Systems* (RT-CPSs) running on distributed systems.

The problem of scheduling PTGs on multiprocessor systems has also received the attention of researchers over many decades. Various optimal solution approaches, such as linear programming, Best-first search, and other exhaustive enumeration techniques including model-based formal synthesis mechanisms, have been proposed [27,28]. Prasanna et al. [29] devised a control-theoretic optimal scheduling mechanism for task graphs executing on a homogeneous multiprocessor system. Later, they have extended their scheme to include communication overheads between task nodes [30]. Sarad et al. [9] developed a MILP based PTG scheduling strategy for platforms consisting of a set of fully connected *homogeneous* processing nodes. Kanemitsu et al. [32] presented an ILP based optimal task scheduling scheme for fully-connected heterogeneous distributed systems. Hsiu et al. [33] developed optimal and approximation algorithms for the scheduling of PTGs on heterogeneous distributed platforms with shared buses. However, these schemes do not consider the combined real-time scheduling of tasks and messages as necessary in fully connected or shared-bus based network system.

Although optimal scheduling solutions may potentially deliver significantly better performance, it may be noted that computation of such optimal solutions may often become prohibitively expensive for large problem sizes. Therefore, research in this domain has also focused towards the design of low-overhead heuristics that provide quick and satisfactory schedules. Heuristic scheduling of PTGs on multiprocessor platforms have often been dealt with *list scheduling* based techniques. These scheduling strategies typically maintain an ordered priority list of all tasks in the PTG [1, 6, 11–13] and involves two phases, (i) *task prioritization*: for selecting the highest-priority ready task and (ii) *processor selection*: for selecting a suitable processor that minimizes execution time. Some examples of this class of techniques include the MCP [34], HLF [35], CPOP [6], HEFT [6], PEFT [1] and HSV [11] algorithms. They attempt to construct a static-schedule for the given PTG to minimize the overall schedule length (makespan) while satisfying resource and precedence constraints. However, none of them are applicable on real-time PTGs, and assume the underlying communication platform to be fully-connected.

Designing energy-efficient schedulers for deadline-constrained PTGs executing on distributed heterogeneous platforms is a computationally hard problem. This is because each task in the PTG consumes distinct amounts of execution time and energy on different processors [8, 47]. In [48], the authors have presented an algorithm called *Downward Energy Consumption Minimization* (DECM) which utilizes the slack available between the completion instant of the sink node and the deadline of a PTG, in order to minimize processor operating frequencies. Further, they extended DECM to a new strategy named *Downward and Upward Energy Consumption Minimization* (DUECM). DUECM improves on DECM by utilizing the slack between adjacent task nodes assigned onto the same processor, while still meeting deadline of the PTG. Huang et al. investigated scheduling approaches on processors which are enabled/not-enabled with DVFS, for PTGs with hard real-time constraints running on fully-connected heterogeneous systems [49]. Jian et al. proposed a two-stage algorithm for the energy-aware scheduling of multiple workflows on DVFS enabled heterogeneous computing systems [8]. In the first stage, all tasks in the workflows are scheduled using HEFT by setting all processors at their highest operating voltage. In the next phase, processor operating voltages have been tuned for each task, to save energy. However, this scheme cannot be employed for scheduling workflows with real-time requirements. In [50], the authors have explored the scheduling of multiple deadline-constrained workflows in a cyber-physical cloud environment with the objective of minimizing energy dissipation of the workflows. However, this work [50] is applicable to only aperiodic/sporadic workloads as only one instance of each workflow has been considered in the schedule. It may be noted that none of the above discussed schemes can be employed to minimize the energy dissipation of multiple *periodic* real-time PTGs executing on a shared bus based heterogeneous distributed platform. We have added Table 2.8 to summarize important related works.

2.5.1 An Overview of HEFT & PEFT

Important concepts related to the heuristic PTG scheduling strategies discussed in Chapters 5, 6 and 7 of this thesis are founded on the principles of two classical *list based* PTG scheduling algorithms, HEFT and PEFT. We now provide overviews of these two algorithms before concluding this chapter.

Heterogeneous Earliest Finish Time (HEFT) [6] is a list-based heuristic scheduling algorithm for executing PTGs on a fully connected distributed heterogeneous multiprocessor platform, with the objective of minimizing the overall *makespan*. It has two major phases, (1) *Task Prioritization Phase:* In this phase, priorities of tasks are set with the

Existing work	**Solution type**	**Task type**	**Computation resource type**	**Data communication architecture**	**QoS Adaptive**	**Energy Aware**	**Real-time**
[19,21,26]	Optimal	Independent task set	Homogeneous	N/A	No	No	Yes
[29]	Optimal	PTG	Homogeneous	Data transmission overhead ignored	No	No	No
[9]	Optimal	PTG	Homogeneous	Fully connected, Routing unaware	No	No	No
[32]	Optimal	PTG	Heterogeneous	Fully connected, Routing unaware	No	No	No
[33]	Optimal, Heuristic	PTG	Heterogeneous	Shared bus based, Routing aware	No	No	Yes
[1,6,11]	Heuristic	PTG	Heterogeneous	Fully connected, Routing unaware	No	No	No
[48]	Heuristic	PTG	Heterogeneous	Fully connected, Routing unaware	No	Yes	Yes
[8]	Heuristic	Multiple PTGs	Heterogeneous	Fully connected, Routing unaware	No	Yes	No
[50]	Heuristic	Multiple PTGs	Heterogeneous	Fully connected, Routing unaware	No	Yes	Yes

Table 2.8: *Summary of related works*

upward rank value, $rank_u$, which is based on mean computation and communication costs. The task list is generated by sorting the tasks in decreasing order of $rank_u$. The highest priority task is then selected to schedule on the processor. (2) *Processor Selection Phase:* In this phase, the highest priority task is selected and assigned to the "best" processor which minimizes the task's finish time.

Predict Earliest Finish Time (PEFT) [1] is currently one of the most popular state-of-the-art list-based heuristic algorithms for scheduling PTGs on a fully connected distributed heterogeneous multiprocessor platform. The algorithm is critically pivoted on a function called $OCT()$ which is used to construct a matrix called Optimistic Cost Table (OCT), containing values corresponding to each task-processor pair. This OCT matrix has two important functions: (1) Determination of a rank value for each task based on which a sorted task list is generated during *task prioritization phase.* This list governs the order in which the tasks are considered for processor assignment, and (2) Determination of the most suitable processor for a task in terms of minimizing the overall *makespan* of the schedule.

2.6 Summary

This chapter started with a brief overview about the basic terms and definitions of real-time systems, followed by literature survey of various scheduling algorithms for real-time multiprocessor systems. These concepts and definitions will be either referred or reproduced appropriately later in this thesis, to enhance readability. In the next chapter, we present strategies for co-scheduling a set of independent periodic tasks with multiple service-levels, executing on a bus-based homogeneous multiprocessor system.

Chapter 3

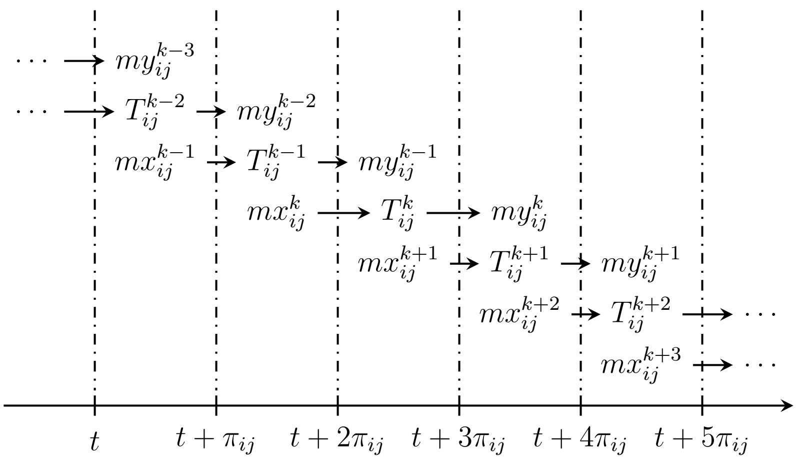
QoS Aware Scheduling of Independent Task Sets on Homogeneous Distributed Systems

In the last chapter, we discussed various scheduling algorithms for real-time multiprocessor systems with the consideration of different design parameters. In this chapter, we propose strategies for co-scheduling a set of independent periodic tasks with multiple service-levels, executing on a bus-based homogeneous multiprocessor system. Here, tasks may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher *rewards/Quality of Service* (QoS) for the system. Thus, successful execution of task at a certain service-level delivers a distinct QoS. The problem is posed as a *Multi-dimensional Multiple-Choice Knapsack formulation* (MMCKP) and present a *Dynamic Programming* (DP) solution (called MMCKP-DP) for the same. Although DP delivers optimal solutions, it suffers from significantly high overheads (in terms of running time and main memory consumption) which steeply increase as the number of tasks, service-levels, processors and buses in the system grows, and severely restricts the scalability of the strategy. Therefore, in addition to the optimal solution approach MMCKP-DP, we propose an efficient but low-overhead heuristic strategy called *Accurate Low Overhead Level Allocator* (ALOLA) which not only consumes drastically lower time and space complexities but also generate good and acceptable solutions, and do not significantly deviate from the optimal solutions.

In this chapter, we first present a detailed description of the problem considered in this work followed by the MMCKP based formulation for adaptive resource allocation and the heuristic solution approach (ALOLA). Subsequently, we discuss experimental results to evaluate the ALOLA and present a case study based on a real-world application. Finally, we concluded the chapter.

3.1 Problem Description

This work considers the problem of off-line scheduling of a set of n persistent real-time periodic tasks with multiple service-levels, on a system consisting of p homogeneous processors and b homogeneous buses. The task deadlines are implicit, that is, deadlines of all tasks are same as their periods. All processors and buses are synchronized in time. Time is represented on a discrete scale and is measured as an integral number of time slots. Each task T_i has $|SL_i|$ alternative service-levels $SL_i = \{sl_{i1}, sl_{i2}, \ldots, sl_{i|SL_i|}\}$, where each service-level sl_{ij} has an associated computation requirement e_{ij} to be completed within a period π_{ij}. Associated with each task, there is also an input message which must be received from a designated sensor prior to the commencement of its computation, via bus. All tasks produce an output message at the completion of their computations which is transmitted over a bus to one or more designated actuators. Thus, corresponding to each service-level sl_{ij}, a task T_i has input and output message transmission demands of mx_{ij} and my_{ij} time slots (over bus(es)) respectively, within every period π_{ij}. We assume both tasks and messages to be preemptive. The computation and total communication resource demands of task T_i at service-level j are denoted by wt_{ij} ($wt_{ij} = \frac{e_{ij}}{\pi_{ij}}$) and wm_{ij} ($wm_{ij} = \frac{mx_{ij}}{\pi_{ij}} + \frac{my_{ij}}{\pi_{ij}}$), respectively. Here, wt_{ij} and wm_{ij} are referred to as the task weight and message weight of T_i for the j^{th} service-level. Without loss of generality, we assume that the period of a task is equal to the sampling period of its input sensor and inverse of the frequency of actuation of its output. Let, T^k_{ij} denote the k^{th} instance of task T_i at service-level sl_{ij}, with mx^k_{ij} being its input message and my^k_{ij} the output message. The designed algorithm must guarantee that mx^k_{ij} (from a sensor) is transmitted and received during the previous period so that the message is available as input to the task at the beginning of the current period. Similarly, the output message my^k_{ij} produced by T_{ij}^k in the current period should be transmitted over the bus to designated actuators during the next period. This pipelined design criterion ensures the completion of one task execution every period, as illustrated in Figure 3.1.



$\cdots \rightarrow my_{ij}^{k-3}$
$\cdots \rightarrow T_{ij}^{k-2} \rightarrow my_{ij}^{k-2}$
$mx_{ij}^{k-1} \rightarrow T_{ij}^{k-1} \rightarrow my_{ij}^{k-1}$
$mx_{ij}^{k} \rightarrow T_{ij}^{k} \rightarrow my_{ij}^{k}$
$mx_{ij}^{k+1} \rightarrow T_{ij}^{k+1} \rightarrow my_{ij}^{k+1}$
$mx_{ij}^{k+2} \rightarrow T_{ij}^{k+2} \rightarrow \cdots$
$mx_{ij}^{k+3} \rightarrow \cdots$
t
$t+\pi_{ij}$
$t+2\pi_{ij}$
$t+3\pi_{ij}$
$t+4\pi_{ij}$
$t+5\pi_{ij}$

Figure 3.1: *Pipelined message transmission and execution over a synchronized system of homogeneous processors and buses*

Successful execution of task T_i at a certain service-level sl_{ij} delivers a distinct *Quality of Service* (QoS) which is measured in terms of a numeric reward[1] value QoS_{ij}. Higher the service-level of execution, higher the QoS achieved by a task, but higher also becomes its resource demands in terms of task and message weights. Thus, computational requirements have a monotonically increasing relationship with service-levels. The actual numeric values of reward assigned to the different service-levels of a task depend on the system wide criticality/importance attributed to the task's different service-level.

Prior to schedule generation, an appropriate service-level is selected for each task such that the aggregate QoS obtained by the system is maximized while guaranteeing feasible schedules for both the tasks' computation demands as well as messages (Given the appropriate task service-levels, two different DP-Fair schedulers have been employed in parallel, one of which allocates processor resources to meet computation demands of the tasks while other allocates bus resources to schedule task messages). For any task T_i, we define a binary decision variable x_{ij} corresponding to each of its service-levels. x_{ij}

[1]The terms *QoS* and *reward* mean the same and has been used interchangeably in the thesis.

is set to 1, if T_i is executed at sl_{ij}. From Equation 2.1, it is clear that, given a selected service-level say, sl_{ij} for each task, feasible execution may be guaranteed if the following constraints hold:

$$\sum_{i=1}^{n}\sum_{j=1}^{|SL_i|} x_{ij} * wt_{ij} \leq p \tag{3.1}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{|SL_i|} x_{ij} * wm_{ij} \leq b \tag{3.2}$$

$$\sum_{j=1}^{|SL_i|} x_{ij} = 1, \quad \text{for } i = 1, 2, \ldots, n \tag{3.3}$$

The objective of the problem discussed above may be symbolically expressed as:

$$max \sum_{i=1}^{n}\sum_{j=1}^{|SL_i|} x_{ij} * QoS_{ij} \tag{3.4}$$

3.1.1 An Optimal Solution Approach (MMCKP-DP)

The objective function in Equation 3.4 along with the constraints in Equations 3.1, 3.2 and 3.3 together constitute a *Multi-Dimensional Multiple-Choice Knapsack Problem* (MMCKP). It may be observed that this MMCKP formulation has an optimal substructure and hence a solution approach based on *Dynamic Programming* (DP) becomes a natural choice. Thus, the above MMCKP may be equivalently represented as a recursive DP formulation as depicted in Equation 3.5 below:

$$f(k,\beta,\gamma) = \max_{j=1}^{|SL_k|} \begin{cases} f(k-1, \beta - wt_{kj}, \gamma - wm_{kj}) + QoS_{kj}, \\ \qquad [\text{If } wt_{kj} \leq \beta, wm_{kj} \leq \gamma \text{ and} \\ \qquad f(k-1, \beta - wt_{kj}, \gamma - wm_{kj}) > 0] \\ 0, \quad [\text{Otherwise}] \end{cases} \tag{3.5}$$

Here, the function $f(k, \beta, \gamma)$ recursively returns the maximum achievable aggregate QoS/reward considering $1, 2, \ldots, k$ tasks, while limiting computation and communication resource consumption below β ($0 < \beta \leq p$) and γ ($0 < \gamma \leq b$), respectively. The constraint: $f(k-1, \beta - wt_{kj}, \gamma - wm_{kj}) > 0$, in Equation 3.5, ensures that while determining the service-level for the k^{th} task, all tasks from $1, 2, \ldots, k-1$ have at least been assigned at their minimum service-levels. It may be noted that β and γ are free to take any arbitrary values within the continuous ranges (0, p] and (0, b], respectively. Therefore, to make β and γ applicable in the DP formulation, the above continuous ranges have been discretized in terms of a unit called *tics*. Thus in Equation 3.5, both β and γ have been measured as integral number of *tics* $((1 \leq \beta \leq \frac{p}{tics})$ and $(1 \leq \gamma \leq \frac{b}{tics}))$. The necessary base condition corresponding to the above recursion is presented in Equation 3.6.

$$f(1, \beta, \gamma) = \max_{j=1}^{|SL_1|} \begin{cases} QoS_{1j}, & [\text{If } wt_{1j} \leq \beta, wm_{1j} \leq \gamma] \\ 0, & [\text{Otherwise}] \end{cases} \tag{3.6}$$

Time Complexity of MMCKP-DP: An implementation of the above DP formulation requires the generation of partial solutions $f(k, \beta, \gamma)$ for different values of the number of tasks (k), as well as all distinct values (β and γ) considered for processor and communication capacities. For any given value of $\langle k, \beta, \gamma \rangle$, the partial optimal solution is provided by choosing that service-level for the k^{th} task which delivers the maximum QoS. Therefore the overall time complexity becomes,

$$O\left(\sum_{i=1}^{n} |SL_i| \times \frac{p}{tics} \times \frac{b}{tics}\right) \tag{3.7}$$

where, $|SL_i|$ denotes the number of service-levels corresponding to task T_i.

Space Complexity of MMCKP-DP: At any intermediate stage, an efficient implementation of MMCKP-DP necessitates memorization of all partial solutions corresponding to the $(k-1)^{th}$ task when partial solutions $f(k, \beta, \gamma)$ for the k^{th} task is being derived. As there are $\frac{p}{tics} \times \frac{b}{tics}$ partial solutions for the k^{th} task and each such solution is an enumeration of the service-levels assigned to tasks $T_1, T_2, \ldots, T_{k-1}$, the overall space complexity is given by,

$$O\left(n \times \frac{p}{tics} \times \frac{b}{tics}\right) \tag{3.8}$$

where, n denotes the number of tasks.

The MMCKP-DP strategy suffers from significantly high overheads which steeply increase as the number of tasks, service-levels, processors and buses in the system becomes larger. The simulation based experimental evaluation shows that even on moderately large systems consisting of 90 tasks with 5 service-levels each, 16 processors and 4 buses, the MMCKP-DP incurs a run-time of more than 1 hour 20 minutes and approximately 68 GB main memory when $tics = 0.001$. Hence, we propose a fast yet efficient heuristic algorithm called ALOLA, which attempts to achieve the same objective of maximizing aggregate rewards such that both computation and communication resource demands are satisfied.

3.1.2 Accurate Low Overhead Level Allocator (ALOLA)

ALOLA is a greedy but balanced heuristic service-level allocation approach that proceeds level by level so that a high aggregate QoS may be acquired by the system at much lower complexity compared to the optimal MMCKP-DP strategy. The mechanism starts by storing all tasks in a max-heap (based on a key $cost_i$) and assigning base service-levels to all tasks. The algorithm then proceeds by repeatedly extracting the task at the root of the heap, incrementing its service-level by 1, updating its *cost* value and reheapifying it, until residual resources are completely exhausted, or all the tasks have been assigned their maximum possible service-levels. Algorithm 1 depicts a step-wise description of ALOLA.

The effectiveness of the solution hinges on the design of the prioritization key, $cost_i$. In order to obtain good and acceptable solutions, ALOLA must not only consider individual QoS gains ($\Delta QoS_i^{j,l}$) during service-level enhancements (from level j to l), but also the amount of incremental consolidated resource ($\Delta CR_i^{j,l}$) required during such an enhancement process. Therefore, the optimization objective gets transformed from QoS as in MMCKP-DP to ($\Delta QoS/\Delta CR$) in ALOLA. ΔCR produces a measure of the incremental consolidated resource consumption combining both computation and communication resource demands and is defined as follows:

$$\Delta CR_i^{j,l} = (1-\alpha) * \Delta wt_i^{j,l} + \alpha * \Delta wm_i^{j,l} \tag{3.9}$$

Here, $\Delta wt_i^{j,l}$ and $\Delta wm_i^{j,l}$ are respectively the extra computation and communication resource demands of task T_i corresponding to service-level enhancement from level j to

l. The term α is a constant and is defined as the ratio of the relative mean approximate bus utilization (ABU) of any task with respect to the mean approximate total resource utilization combining processors (APU) and buses (ABU). Hence, α is symbolically represented as follows:

$$\alpha = \frac{ABU}{APU + ABU} \tag{3.10}$$

where, $[APU = (\sum_{i=1}^{n} \frac{1}{|SL_i|}(\sum_{j=1}^{|SL_i|} wt_{ij}))/M]$ and $[ABU = (\sum_{i=1}^{n} \frac{1}{|SL_i|}(\sum_{j=1}^{|SL_i|} wm_{ij}))/B]$. It may be observed that the parameters α and $(1-\alpha)$ provides a measure of the relative approximate bus utilization and processor utilization for any task considering all possible service-level assignments over all tasks. This makes α independent of specific service-levels assigned to the tasks at any time and thus, its value remains unchanged over the entire execution of the algorithm. Such a static definition of α helps us to control the overall complexity of the ALOLA algorithm. The key $cost_i$ which determines the urgency of each task T_i (towards service-level upgrade) within the heap, is as follows:

$$cost_i = max \left\{ \frac{\Delta QoS_i^{j,j+1}}{\Delta CR_i^{j,j+1}}, \frac{\Delta QoS_i^{j,|SL_i|}}{\Delta CR_i^{j,|SL_i|}} \right\} \tag{3.11}$$

Here, $[\Delta QoS_i^{j,j+1}/\Delta CR_i^{j,j+1}]$ denotes the additional QoS received by the system per unit consolidated resource consumption, if T_i is upgraded from current service-level j to level $(j+1)$ and $[\Delta QoS_i^{j,|SL_i|}/\Delta CR_i^{j,|SL_i|}]$ represents the QoS gain per unit additional resource consumption, if T_i is enhanced from level j to its highest service-level $|SL_i|$.

It was observed that $cost_i$ (Equation 3.11) is able to provide an appropriate balance between the immediate gain obtained through a service-level enhancement from j to $(j+1)$ and the overall obtainable gain for task T_i is $(\Delta QoS_i^{j,|SL_i|}/\Delta CR_i^{j,|SL_i|})$. A situation where the consideration of such overall gains may be useful is as follows: Let us consider the relative urgency of two tasks T_i and $T_{i'}$ currently allocated service-level j and j' respectively (say), at an arbitrary intermediate stage of the resource allocation process using ALOLA. Assume, $[\Delta QoS_i^{j,j+1}/\Delta CR_i^{j,j+1}]$ is lower than $[\Delta QoS_{i'}^{j',j'+1}/\Delta CR_{i'}^{j',j'+1}]$. However,

$$\frac{\Delta QoS_i^{j,|SL_i|}}{\Delta CR_i^{j,|SL_i|}} >> \frac{\Delta QoS_{i'}^{j',|SL_{i'}|}}{\Delta CR_{i'}^{j',|SL_{i'}|}}.$$

ALGORITHM 1: ALOLA

```
Input: A set of $n$ tasks, $p$ homogeneous processors and $b$ homogeneous buses
Output: Assignment of service-levels to tasks
1  Assign minimum service-level to all tasks
2  forall tasks $T_i$ do
3      Compute $cost_i$ using Equation 3.11
4  Create max-heap based on $cost_i$
5  Insert all tasks into max-heap
6  while max-heap is non-empty do
7      Select root node $T_i$
8      if Extra resource demands of $T_i \leq$ available resources then
9          Upgrade service-level of $T_i$ to the next one
10         if current service-level $< |SL_i|$ then
11             Recompute $cost_i$ using Equation 3.11
12         else
13             remove $T_i$ from max-heap
14     else
15         remove $T_i$ from max-heap
16     Heapify max-heap
```

In such a situation, if overall gain in QoS/reward is not considered as part of the *key*, $T_{i'}$ will be selected for up-gradation by one level over T_i even if its overall gain $(\Delta QoS_i^{j,|SL_i|}/\Delta CR_i^{j,|SL_i|})$ is much greater than

$$max\left\{\frac{\Delta QoS_{i'}^{j',j'+1}}{\Delta CR_{i'}^{j',j'+1}}, \frac{\Delta QoS_{i'}^{j',|SL_{i'}|}}{\Delta CR_{i'}^{j',|SL_{i'}|}}\right\}.$$

A more severe case is that, if T_i's immediate gain $[\Delta QoS_i^{j,j+1}/\Delta CR_i^{j,j+1}]$ is relatively very low, T_i may be indefinitely starved in spite of potentially handsome overall gains. Defining the key $cost_i$ as,

$$max\left\{\frac{\Delta QoS_i^{j,j+1}}{\Delta CR_i^{j,j+1}}, \frac{\Delta QoS_i^{j,|SL_i|}}{\Delta CR_i^{j,|SL_i|}}\right\},$$

appropriately handles the situation.

Time Complexity of ALOLA: The complexity related to the assignment of the minimum service-level to all tasks and computation of their $cost_i$ values in lines 1-3, is $O(n)$. Building the max-heap in line 4 also takes $O(n)$ time. The *while* loop in lines 6-16 iterates at most $|SL_i|$ times for each task T_i and during each iteration, the heapify operation (line 16) incurs $O(\log n)$ time. So, the overall time complexity of the ALOLA algorithm becomes $O(\sum_1^n |SL_i| \times \log n)$.

Space Complexity of ALOLA: The space complexity of ALOLA is upper bounded by the memory required to store *task weights*, *message weights* and *QoS* values corresponding to all service-levels for each task. So, the overall space complexity of the ALOLA is $O(n \times \sum_1^n |SL_i|)$.

3.1.3 Example: Service-level Assignment

Consider $(n = 3)$ tasks T_1, T_2, and T_3 with $p = 2$ processors and $b = 1$ bus. The $\left\langle wt_{ij}, wm_{ij}, QoS_{ij} \right\rangle$ values corresponding to different service-levels of each task are given in Table 3.1. For these values, $APU = 0.95$, $ABU = 0.933$ and $\alpha = 0.496$.

Task	SL_i	wt_{ij}	wm_{ij}	QoS_{ij}
T_1	sl_{11}	0.3	0.1	2
	sl_{12}	0.6	0.2	4
	sl_{13}	0.7	0.3	5
T_2	sl_{21}	0.6	0.3	4
	sl_{22}	0.7	0.4	7
	sl_{23}	0.8	0.4	8
T_3	sl_{31}	0.5	0.2	2
	sl_{32}	0.7	0.3	6
	sl_{33}	0.8	0.6	7

Table 3.1: *Tasks parameters*

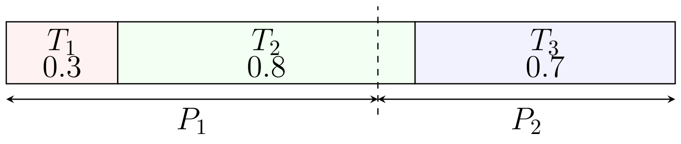
The overall residual computation and communication capacities after allocating the minimum service-level sl_{i1} to each task T_i become: $(p - \sum_{i=1}^n wt_{i1} =)$ 0.6 and $(b - \sum_{i=1}^n wm_{i1} =)$ 0.4, respectively. The initial key values corresponding to the three tasks are $cost_1 = 9.96$, $cost_2 = 30$, $cost_3 = 26.59$. A Max Heap (H) is built using these key values. T_2 with the highest key value (currently, at the root of the max-heap) is extracted from the heap, its service-level is enhanced from sl_{21} to sl_{22}, $cost_2$ is updated ($cost_2$ becomes 19.82) and then T_2 is re-heapifyed. Now, T_3 with the currently highest key value is extracted from the heap, its service-level is upgraded to sl_{32}, the key value is recomputed ($cost_3$ becomes 5.02) and T_3 is re-heapifyed. After this, T_2 again becomes the task with the highest key value and hence, it is upgraded to sl_{23} subsequent to its extraction from the heap. T_2, which has reached its highest enhancement level, is removed from further consideration and so, it is not reinserted into the heap. Now, T_1 which is the task with the highest key, is extracted from the heap but its service-level cannot be enhanced further to sl_{12} as the additional resources necessary for this enhancement is more than the currently remaining residual resources. T_1 is not considered further. Due to the same reason, T_3 also cannot be considered for further enhancement. The heap becomes empty and the algorithm terminates. The final service-levels for T_1, T_2 and T_3 are sl_{11}, sl_{23} and sl_{32}, respectively and the aggregate QoS fetched by the system is 16.

3.1.4 Offline Schedule Generation

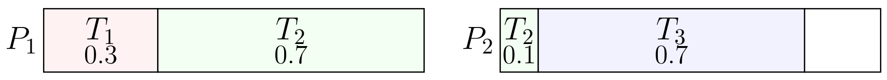
The ALOLA algorithm provides the final service-levels at which each task should be executed. Subsequent to this, the DP-Fair scheduling technique is employed to allocate tasks on processors and messages on buses, for the entire duration $\mathcal{H}$ corresponding to the system's hyperperiod. These offline schedules are then used for online execution and repeat every hyperperiod.

We explain the generation of DP-Fair schedules by continuing with the illustrative example presented in Section 3.1.3. For the service-levels selected by ALOLA, the task weights of T_1, T_2 and T_3 are 0.3, 0.8 and 0.7 respectively, while the corresponding message weights are 0.1, 0.4 and 0.3 respectively. The tasks are first partitioned into the available processors and buses by: (i) lining them up over two separate scales based on task and message weights, as shown in Figures 3.2a and 3.3a respectively, and (ii) extracting unit length chunks from designated scales for allocation on to distinct processors and buses, as depicted by Figures 3.2b and 3.3b. It may be observed from Figure 3.2b that after partitioning, task T_2 becomes a migrating task with one part (0.7-out-of-0.8) allocated to processor P_1 and the other part (0.1-out-of-0.8) allocated to P_2.

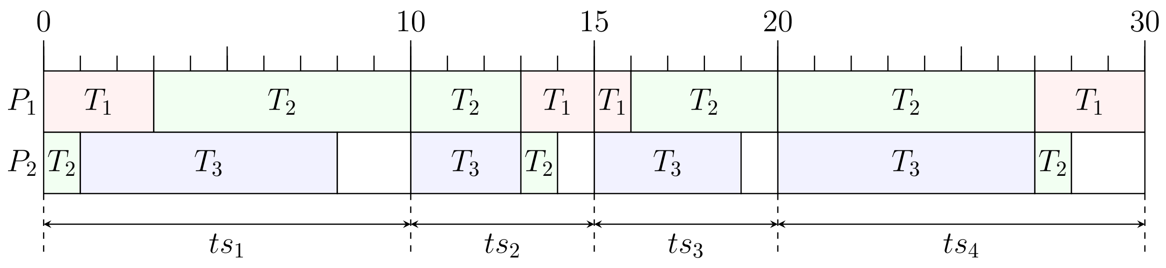
Let the execution times, message transmission times and periods ($\left\langle e_{ij}, m_{ij}, \pi_{ij} \right\rangle$) for the tasks (at their selected service-levels) be: T_1 $\langle 9, 3, 30 \rangle$, T_2 $\langle 12, 6, 15 \rangle$, T_3 $\langle 7, 3, 10 \rangle$. To construct a DP-Fair schedule, the hyperperiod of duration $\mathcal{H} = 30$ (LCM(30, 15, 10), where LCM is the *least common multiple*), is divided into slices $ts_1 = 10, ts_2 = 5, ts_3 = 5$ and $ts_4 = 10$, demarcated by the periods/deadlines of all jobs in $\mathcal{H}$. The tasks and messages are then allocated resource shares in each time slice in proportion to their allocated weights on appropriate processors and buses, as designated by the partitioning process. For example, based on the task weight partition obtained in Figure 3.2b, T_1 and T_3 are allocated execution shares 3 and 7 on P_1 and P_2 respectively, in time slice ts_1. Also T_2, the migrating task, receives shares 7 and 1 on P_1 and P_2 in ts_1. Figure 3.2c depicts the schedule of task execution over all the four time slices in $\mathcal{H}$. It may be observed from the figure that the scheduling sequences in consecutive time slices are mirrored in order to avoid task migrations at time slice boundaries. The system is able to limit context switches to 2 and number of migrations to 1 per time slice on our example three-task-two-processor system. The message schedule is obtained in a way very similar to the generation of the task schedule using the DP-Fair algorithm. Figure 3.3c depicts the obtained message schedule for the entire hyperperiod $\mathcal{H}$.



(a) *Tasks lined up on a scale based on task weights. The dotted line partitions the chunks of at most unit length*

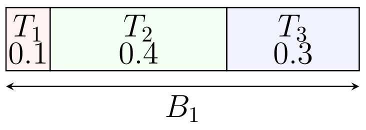


(b) *Tasks partitioned on to processors P_1 and P_2. The white colored block in P_2 represents unused capacity*

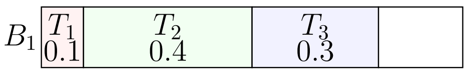


(c) *Task schedule for hyperperiod $\mathcal{H}$*

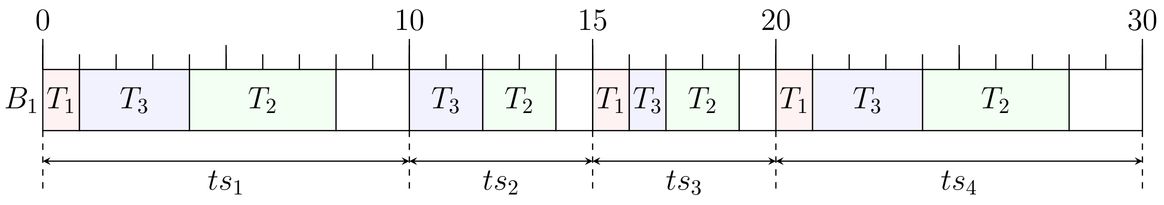
Figure 3.2: *Partitioning and Scheduling of Tasks using DP-Fair*



(a) *Message weight partitioning*



(b) *Allocated message weights to bus*



(c) *Message schedule for hyperperiod $\mathcal{H}$*

Figure 3.3: *Partitioning and Scheduling of Messages using DP-Fair*

3.2 Experiments and Results

In this section, we experimentally evaluate the performance of the ALOLA algorithm and compare it against the optimal MMCKP-DP strategy in terms of both performance (total QoS acquired) and execution time required. The proposed work has been evaluated using simulation based experiments.

3.2.1 Data Generation Framework

The experimentation framework is used as follows: The data sets consist of randomly generated hypothetical periodic tasks whose periods (π_{i1}), task weights (e_{i1}/π_{i1}) and message weights (m_{i1}/π_{i1}) for lowest/base service-level have been generated from normal distributions with mean $\langle mean(\mu),\ standard\ deviation(\sigma)\rangle$ values being $\langle 2000, 400\rangle$, $\langle 0.2, 0.1\rangle$ and $\langle 0.2, 0.1\rangle$, respectively. The reward values (QoS_{i1}) of the tasks are generated from a uniform distribution within the range from 20 to 200. Given n tasks, p processors and b buses, the processor utilization PU and bus utilization BU corresponding to a data set generated using the above mentioned distributions are given by:

$$PU = \frac{1}{p}\sum_{i=1}^{n} wt_{i1} \qquad and, \qquad BU = \frac{1}{b}\sum_{i=1}^{n} wm_{i1} \tag{3.12}$$

Experiments have been conducted with data sets having different PU and BU values varying within the range from 0.6 to 1. To obtain a desired and fixed value of PU and BU corresponding to a given data set, the generated weights wt_{i1} and wm_{i1} are appropriately scaled. All tasks are assumed to have 5 service-levels. The weights (wt_{ij} or wm_{ij}) for non-base service-levels (starting from level 2) of the tasks are assigned uniform random values bounded between 110% and 120% of the weights ($wt_{i(j-1)}$ or $wm_{i(j-1)}$) corresponding to their immediately lower service-levels. The rewards (QoS_{ij}) for any task T_i increase monotonically as service-levels become higher. The values of the rewards have been chosen randomly from the range 20 to 200, while ensuring that the random reward value for a task at a given service-level is higher than the reward values at lower service-levels.

We have conducted experiments with data sets having 15, 30, 45 and 60 tasks, #processors varying between 2 to 8 and #buses varying between 1 to 4. All data points in the plots presented below are obtained by taking the average of the results from 50 different runs of a given experiment, executed with distinct data sets for a fixed set of parameters. All experiments have been carried out on a 2.5 GHz core, with 128 GB of physical memory.

3.2.2 QoS Measurements

The performance of ALOLA and MMCKP-DP have been evaluated and compared using a metric called *Normalized Reward* (NR) and is defined as,

$$NR \text{ (in \%)} = \frac{R_{ACT}}{R_{MAX}} \times 100 \tag{3.13}$$

where, R_{ACT} is the total reward acquired through a given service-level assignment and R_{MAX} is the maximum achievable reward (when the highest service-level is always assigned). Two categories of experiments have been conducted. The first category conducts performance analysis in scenarios where bus resources are unlimited, that is, the bus capacity remains underloaded even when all tasks are assigned highest service-levels. For the second category, we assume unlimited processor resources.

Results (Category 1): Figure 3.4 presents the NR plots obtained for both ALOLA and MMCKP-DP in systems consisting of 15, 30, 45 or 60 tasks, 8 processors and processor utilization with respect to minimum service-levels PU (refer Equation 3.12) varying between 0.6 and 1. It may be observed from the figure that as is obvious, QoS based rewards decrease monotonically as PU increases. This is because the probability of attaining higher service-levels is reduced with the increase in PU. The optimal algorithm is seen to significantly outperform the heuristic strategy ALOLA (by about 5% to 10%) for values of PU between $\sim$ 0.6 and $\sim$ 0.85. However, as system load increases further and PU approaches $\sim$ 1, both the optimal and heuristic strategies select only the lowest service-level for each task. This is indicated in Figure 3.4, which shows that both MMCKP-DP and ALOLA deliver the same NR values for the same number of tasks, when $PU = 1$. In Table 3.2, we present the average NR values acquired by both the algorithms for a system consisting of 45 tasks, five distinct values of PU (0.6, 0.7, 0.8, 0.9 and 1) and four distinct number of processors (2, 4, 6 and 8). The table shows that the performance of both algorithms degrade as system load increases and that MMCKP-DP outperforms ALOLA in all cases except fully loaded systems ($PU = 1$), conforming the observations made for Figure 3.4. However, the NR values are not seen to significantly vary with increase in the number of processors for any given value of PU.
Results (Category 2): Experiments for category 2 have been conducted for similar scenarios as category 1, although here, we vary bus utilization BU (refer Equation 3.12) from 0.6 to 1, instead of processor utilization PU. As depicted in Figure 3.5 and Table 3.3, the trends of the results corresponding to this category are almost identical to those obtained for category 1.

3.2.3 Time Measurements: Results

The performance of the designed algorithms with respect to incurred running times have been evaluated by measuring their actual average run-times over various data sets (here, run-time denotes only service-level selection overheads and does not include running

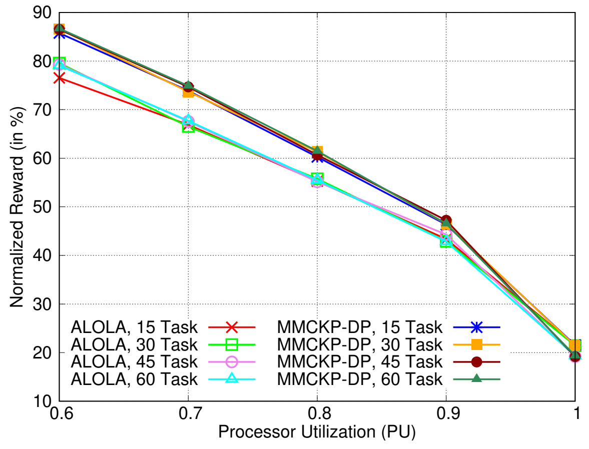


Figure 3.4: *Processor Utilization (PU) Vs. NR*

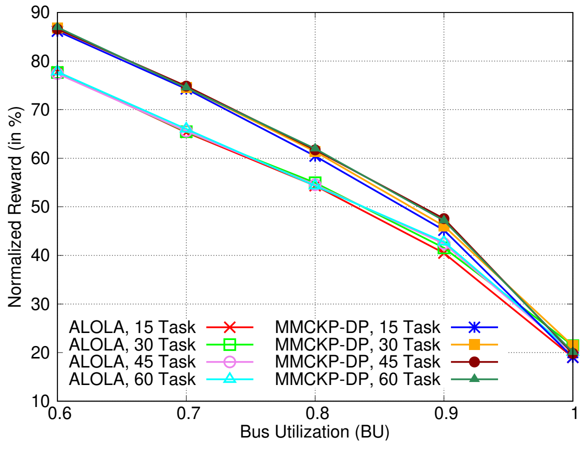


Figure 3.5: *Bus Utilization (BU) Vs. NR*

PU	Algorithm	#Processors			
		2	4	6	8
0.6	MMCKP-DP	86.94	86.88	86.2	86.51
	ALOLA	78.25	79.11	78.84	79.29
0.7	MMCKP-DP	74.5	74.24	73.91	74.68
	ALOLA	66.93	66.71	67.11	67.57
0.8	MMCKP-DP	61.93	60.64	61.84	60.68
	ALOLA	54.8	54.88	55.91	55.14
0.9	MMCKP-DP	47.29	46.44	46.58	47.22
	ALOLA	43.03	42.84	43.52	44.3
1	MMCKP-DP	21.16	19.55	19.82	19.19
	ALOLA	21.16	19.55	19.82	19.19

Table 3.2: *QoS (NR) of MMCKP-DP and ALOLA for varying Processor Utilization (PU) and #Processors*

BU	Algorithm	#Buses			
		1	2	3	4
0.6	MMCKP-DP	87.38	87.02	87.13	86.55
	ALOLA	76.06	77.49	77.76	77.4
0.7	MMCKP-DP	75.25	74.96	75.16	74.83
	ALOLA	65.51	65.55	65.6	65.58
0.8	MMCKP-DP	62.69	62.23	61.63	61.71
	ALOLA	53.71	53.83	54.37	54.54
0.9	MMCKP-DP	46.72	47.39	47.41	47.57
	ALOLA	40.39	41.35	42.16	42.43
1	MMCKP-DP	22.7	21.02	19.99	19.82
	ALOLA	22.7	21.02	19.99	19.82

Table 3.3: *QoS (NR) of MMCKP-DP and ALOLA for varying Bus Utilization (BU) and #Buses*

time of the scheduled tasks on the processors/buses). Then, we have determined the *speedup* achieved by ALOLA over MMCKP-DP. That is,

$$speedup = \frac{\text{Running time of MMCKP-DP}}{\text{Running time of ALOLA}} \tag{3.14}$$

Speedups have been measured for both the categories of experiments for which performance evaluation in terms of QoS was conducted.

Results: Figures 3.6 and 3.7 depict results for the running times incurred by ALOLA for category 1 (which considers 15 to 60 tasks, 4 and 8 processors, 2 buses, with PU being varied between 0.6 and 1) and category 2 (which considers 15 to 60 tasks, 8 processors, 2 and 4 buses, with BU being varied between 0.6 and 1) experiments, respectively. From both figures, it may be seen that run-time of ALOLA decreases as PU (category 1: Figure 3.6) or BU (category 2: Figure 3.7) increases. This is obvious because with the increase in utilization, residual capacities reduce and exhaust more quickly during the execution of ALOLA, thus giving the strategy less opportunity to select higher service-levels. For any given utilization, running times increase with the number of tasks. This is also expected because the time complexity of ALOLA is directly proportional to the number of tasks and service-levels. However, for any given utilization and number of tasks, the run-times remain unaffected by the change in number of processors (category 1: Figure 3.6) or number of buses (category 2: Figure 3.7). In Table 3.4, we present the speedups achieved by ALOLA over MMCKP-DP for 15 to 90 tasks, number of processors varying from 2 to 16 (with #buses $b = 2$, $PU = 0.7$; underloaded bus capacity) and number of buses varying between 1 to 4 (with #processors $p = 16$, $BU = 0.7$; underloaded processor capacity). The actual speedups are 10^6 times the values (say, x) shown in the table (that is, actual speedup = $x \times 10^6$ times). It may seen that speedups increase with the number of tasks, processors and/or buses. The reason may be attributed to the complexity of MMCKP-DP which is highly sensitive to the number of tasks, their service-levels, as well as the number of processors and buses (refer Equation 3.7). In comparison, the complexity of ALOLA exhibits significantly lower sensitivity to the number of tasks and service-levels (refer Section 3.1.2). ALOLA's running time is also dependent on the residual processor and bus capacities. Therefore, run-time of ALOLA does not significantly vary with changes in the number of processors and/or buses, being only indirectly sensitive to them.

3.3 Case Study: Flight Management System

To illustrate the generic applicability of our proposed strategy to real world designs, we present a case study using an automated flight control system employed in modern

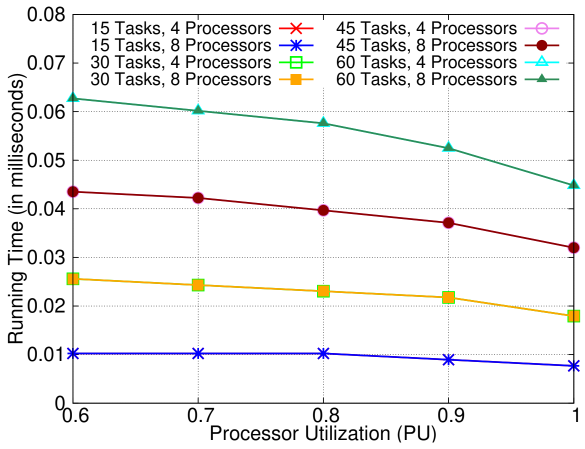


Figure 3.6: *ALOLA: Processor Utilization (PU) Vs. Running Time*

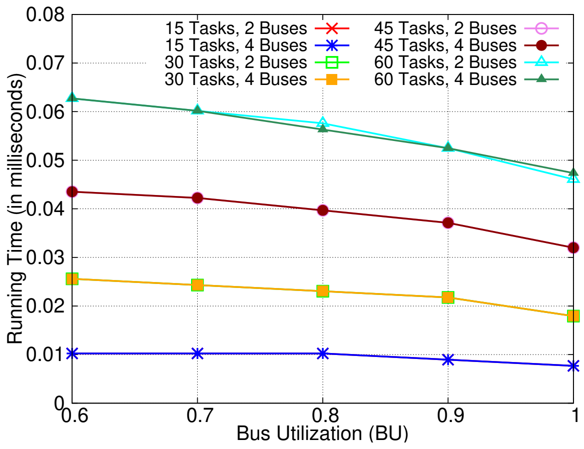


Figure 3.7: *ALOLA: Bus Utilization (BU) Vs. Running Time*

Number of Tasks	Speedup ($x \times 10^6$)								
	#Processors					#Buses			
	2	4	6	8	16	1	2	3	4
15	0.12	0.7	1.13	1.66	2.89	1.46	2.9	4.52	6.33
30	0.62	1	1.55	2.14	4.06	2.52	5.92	8.11	9.25
45	0.75	1.51	2.36	3.02	6.03	2.88	7.19	10.53	13.58
60	0.89	1.97	2.81	3.78	8.08	3.54	8.52	10.95	14.71
75	1.11	2.26	3.12	4.45	8.14	5.98	9.79	14.34	18.32
90	1.21	2.58	4.19	4.7	10.07	5.9	11.48	17.4	24.23

Table 3.4: *Speedup of ALOLA with respect to MMCKP-DP*

avionic systems. The FMS in an aircraft performs several flight control functions like flight planning, navigation, guidance and control [51]. FMS employs four separate tasks, namely, *Guidance*, *Control*, *Slow Navigation* and *Fast Navigation*, to control the aircraft during flight. The *Guidance* task (say T_1) sets the reference trajectory of the aircraft with respect to altitude and heading. The *Control* task (say T_2) executes closed loop control functions that compute actuator commands based on reference trajectory and navigation sensor values. Actuator commands for numerous components including elevator, ailerons, rudder and throttle are executed by the control task in order to achieve desired reference altitude and heading for the aircraft. The *Slow Navigation* task (say

T_3) reads data from sensors at low frequencies which are used by the less critical *Guidance* task to determine high-level altitude and heading commands. The *Fast Navigation* task (say T_4) on the other hand, is used to read data from sensors at higher frequencies which are used by the higher critical *Control* task. In addition to these basic tasks (Guidance, Control, Slow Navigation and Fast Navigation), a fighter aircraft like F-16 performs an additional task, *Missile Control* (say T_5), which monitors the aircraft radar to detect enemy targets and if a target has been detected, it fires a missile to destroy the target.

We have used the task set corresponding to FMS simulation of F-16 fighter aircraft, from [7]. Table 3.5 shows the execution times (e_i), message transmission times (m_i), periods (π_i), computation demands (wt_i), communication demands (wm_i) and QoS (QoS_i) for the different service-levels (SL_i) of all the five tasks mentioned above. Both algorithms MMCKP-DP and ALOLA run successfully on the task set in Table 3.5. In this particular case, the heuristic algorithm ALOLA assigns the same service-levels to the tasks as the optimal algorithm MMCKP-DP. The assigned service-levels to tasks T_1, T_2, T_3, T_4, and T_5 by both ALOLA and MMCKP-DP are $sl_{13}, sl_{23}, sl_{32}, sl_{43}$, and sl_{52}, respectively. The aggregate QoS fetched by the system due to this assignment is 464.

3.4 Applicability Considerations

ALOLA is designed keeping in mind of the communication protocols that allow messages to be preempted/migrated. However, to use ALOLA with communication protocols that do not allow message preemption/migration, the largest size a message can have, must be restricted to the size of the smallest time slice within the hyper-period. This is because, time slots, time slices and task periods, have the same length and are always synchronized across all processors and buses. For a major class of real-time control systems, where message sizes (in terms of transmission time) are typically very small compared to the sizes of control tasks, this restriction may not be impractical. For example CAN, a common bus protocol used for control data transmission in automotive applications, allow message (frame) payloads to be at most 64 bytes with data rates being 5 Mbps

Tasks	SL_i	e_i [ms]	m_i [ms]	π_i [sec]	wt_i	wm_i	QoS_i
T_1 (Guidance)	sl_{11}	100	80	10	0.01	0.008	10
	sl_{12}	100	80	5	0.02	0.016	15
	sl_{13}	100	80	1	0.1	0.08	20
T_2 (Controller)	sl_{21}	80	100	5	0.016	0.02	1
	sl_{22}	60	80	1	0.06	0.08	100
	sl_{23}	80	100	1	0.08	0.1	104
	sl_{24}	60	80	0.2	0.3	0.4	120
	sl_{25}	80	100	0.2	0.4	0.5	124
T_3 (Slow Navigation)	sl_{31}	100	120	10	0.01	0.012	10
	sl_{32}	100	120	5	0.02	0.024	20
	sl_{33}	100	120	1	0.1	0.12	25
T_4 (Fast Navigation)	sl_{41}	60	70	5	0.012	0.014	1
	sl_{42}	60	70	1	0.06	0.07	100
	sl_{43}	60	70	0.2	0.3	0.35	120
T_5 (Missile Control)	sl_{51}	500	200	10	0.05	0.02	1
	sl_{52}	500	200	1	0.5	0.2	200

Table 3.5: *Task set of FMS [7]*

(for the CAN FD standard). So, a CAN message can be very comfortably transmitted within 1 millisecond, which is the typical size of a time slot (and hence, the minimum time slice length) as considered in this work. Thus, ALOLA can be seamlessly used with protocols like CAN, completely avoiding message preemption/migration.

In addition, task deadlines/periods are often adjustable within bounded limits [52–55] in many real-time systems. This flexibility can be used with ALOLA to obtain lower bounds on time slice lengths which are significantly higher than the size of a single time slot (1 millisecond). With this approach, ALOLA can also be used with communication protocols which support considerably higher message sizes compared to CAN, but do not allow message preemption/migration.

More recently, there is a proliferation in the use of distributed real-time systems in a variety of application domains such as smart industries, smart-grids, medical systems, automotive and avionic systems etc., many of which may need to support even larger message sizes than those in more traditional control systems. Keeping in view the emergence of applications which need to support large message sizes, newer research works and protocols are evolving. These strategies allow large messages to be split into smaller chunks which may possibly be transmitted in an independent fashion. For example, Glabbeek et al. in [56] proposed the *Fragmentation and Reassembly* protocol running on top of the CAN bus, to split large messages into multiple fixed size frames at the transmission end, and to reassemble the fragmented messages at the receiver end. In the presence of multiple CAN buses, different frames of the same message can possibly be transmitted over distinct buses. As another example, in the futuristic Ethernet based IEEE 802.1Qbv Time Sensitive Networking (TSN) [57–59], each message of a message stream/flow, can be split into multiple Ethernet frames, and each Ethernet frame can follow different paths. Thus, the two protocols discussed above effectively allow both preemption and migration of messages.

Although, the proposed scheme ALOLA is a simple generic processor-bus co-scheduling strategy which has not been designed with any particular protocol in mind, we purview that it is possible to extend ALOLA in order to support any of the above protocols, with moderate effort. This work assumes that all buses are connected to every device in the system, and the assumption is true when the intended system under design is small-scale, containing a small number of devices and geographically spread over a small area. However in the case of larger systems where devices cannot be connected to all buses, ALOLA cannot be directly employed and must be appropriately adapted. There may be a need to adopt a more partitioned approach where a subset of devices (consisting of processors, sensors and actuators) is connected to a disjoint sub-group of buses, and a device is always connected to all buses within a sub-group. Among devices, a single sensor/actuator may possibly be connected to multiple bus sub-groups. Now, a separate ALOLA scheduler can be deployed for each distinct sub-group of buses and the processors connected to this bus sub-group.

3.5 Summary

Many cyber-physical systems including those in the automotive domain often execute applications which necessitate combined scheduling on both processors and buses. In this chapter, we consider independent periodic tasks with multiple service-levels, where each instance of a task also needs to receive data from sensor(s) and produce data to actuator(s) through one or more system buses. The scheduling problem is modeled as a *Multi-dimensional Multiple-Choice Knapsack Problem formulation* and shown that conventional *Dynamic Programming* based solution approaches are very expensive. Therefore, we have proposed a heuristic strategy called ALOLA, which is very fast producing speedups of the order of 10^6 times with respect to the optimal *Dynamic Programming* solution and also efficient, being able to control performance degradations to at most 13% compared to the optimal solutions. While this chapter considers the scheduling of independent tasks, the remaining chapters of this thesis deals with *Precedence-constrained Task Graphs* (PTGs). Applications represented as PTGs are increasingly becoming important in CPSs ranging from automotive and avionics domains, smart grids, nuclear plants, etc. Many of these CPSs are quickly moving from homogeneous to heterogeneous platforms in order to extract higher application specific performance, better thermal and power characteristics, etc. Hence, in the next and subsequent chapters, we delve with computation-communication co-scheduling strategies for PTGs on heterogeneous distributed systems.

Chapter 4

Optimal Scheduling of PTGs on Heterogeneous Distributed Systems

In the previous chapter, we have considered the problem of scheduling real-time independent task sets running on homogeneous multiprocessor systems. However, Continuous demands for higher performance and reliability within stringent resource budgets is driving a shift from homogeneous to heterogeneous processing platforms for the implementation of today's CPSs. These CPSs are often distributed in nature and typically represented as PTGs due to the complex interactions between their functional components. This chapter discusses two optimal scheduling mechanism for a real-time system modelled as a PTG to be executed on a fully connected distributed heterogeneous platform. Here, tasks may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher *rewards*/QoS for the system. To solve the problem, an ILP based optimal solution approach namely, *ILP - Service-level Allocation with Timed Constraints* (ILP-SATC), is proposed. Though the formulation of ILP-SATC follows an intuitive design flow, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In order to improve scalability, a second ILP based strategy namely, *ILP - Service-level Allocation with Non-overlapping Constraints* (ILP-SANC), has been designed. Instead of explicitly relying on task mobility based manipulations as ILP-SATC, ILP-SANC guarantees that the executions of no two tasks in the system overlap in time on the same processor. This modification in the design approach allows the constraint set in ILP-SANC to be independent of the deadline of a given PTG.

In this chapter, we first present a detailed description of the problem considered in this work, followed by the formulation of ILP-SATC and ILP-SANC. Then, both the ILP formulations have been comprehensively evaluated through an extensive set of experiments over benchmark PTGs. Performance of the design strategies have also been compared against a state-of-the-art approach [1]. Finally, we demonstrate the practical applicability of the ILP based formulation using a real-world case study on an *Adaptive Cruise Controller* (ACC) application and conclude the chapter.

4.1 The Task and Platform Models

This work considers a periodic application represented as a PTG $G = (T, \mathcal{E})$ to be executed on a fully-connected *heterogeneous multiprocessor platform* consisting of a set of processors $P = \{P_1, P_2, \ldots, P_p\}$. Figure 4.1 shows the pictorial representation of the fully-connected *heterogeneous multiprocessor platform* as considered here. Each processor has its own private memory. For any given processor, task execution and communication with other processors can be conducted simultaneously, without any contention. Specifically, we assume that tasks on different processors communicate by transmitting data from the source processor via the fully-connected network to the local memory of the receiving processor. On the other hand, intra-processor communication is realized through the reading and writing of variables stored in the local memory of the processor.

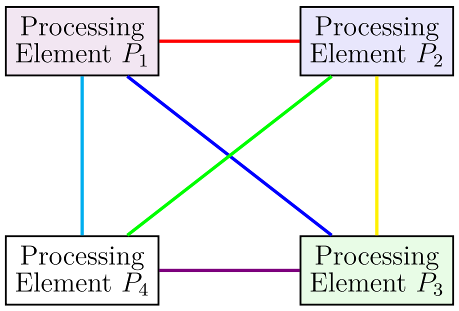


Figure 4.1: *Fully connected multiprocessor system*

- $T = \{T_1, T_2, \ldots, T_n\}$ represents n task nodes.

- $\mathcal{E} \subseteq T \times T$ denotes the edge-set that describes *precedence-constraints* between pairs of tasks in T. The symbol m_{ij} is used to denote the communication cost associated with the edge $T_i \rightarrow T_j$; $m_{ij} = 0$, when T_i and T_j are assigned to the same processor. All tasks/messages in PTG G execute/transmit non-preemptively.

- Each task T_i has $SL_i = \{sl_{i1}, sl_{i2}, \ldots, sl_{i|SL_i|}\}$ alternative service-levels. Associated with a given service-level sl_{il} of task T_i, there exists p possibly distinct worst-case execution times e_{ilr} ($i \in [1, n]$, $l \in [1, |SL_i|]$, $r \in [1, p]$) corresponding to the p heterogeneous processors. There is a *reward* QoS_{il} which is obtained on successful completion of every instance of T_i.

- Given any two service-levels sl_{il} and $sl_{il'}$ of task T_i such that $l < l'$, the execution times on any processor P_r are related as $e_{ilr} \leq e_{il'r}$. However, the processors being heterogeneous, the execution times of a task on two different processors are completely unrelated.

- The execution times of a task T_i on processor P_r for all service-levels is set to ∞, to model the scenario in which the execution of T_i is infeasible on P_r.

- Each instance of application G has an associated *deadline* D. Two consecutive instances of G are separated by a *period* π. We assume the deadline to be *implicit*, i.e., $D = \pi$.

To denote in-degree and out-degree of the task node T_i, we use the notations $indeg(T_i)$ and $outdeg(T_i)$, respectively. Given a pair of task nodes $\langle T_i,\, T_j \rangle$, T_i (T_j) is said to be the predecessor (successor) of T_j (T_i) if there is an edge ($T_i \rightarrow T_j$) form T_i to T_j. To denote predecessor and successor of the task node T_i, we use the notations $pred(T_i)$ and $succ(T_i)$, respectively. Similarly, given two tasks T_i and T_j, T_i (T_j) is said to be the *ancestor* (*descendant*) of T_j (T_i), if $T_i \prec T_j$ i.e., there exists a path from T_i to T_j, in the PTG.

Example: An example of a PTG G and its task parameters are shown in Figure 4.2 and Table 4.1, respectively. PTG G consists of 6 tasks, each task having 2 service-levels. Each task T_i may have distinct execution times e_{ilr} on each processor P_r for any given service-level sl_{il} along with associated reward QoS_{il}. For example, at service-level sl_{11}, task T_1 has execution times $e_{111} = 3$ and $e_{112} = 1$ on processors P_1 and P_2, respectively, and an associated reward $QoS_{11} = 4$. An edge between T_1 and T_2 has an associated communication cost of 2.

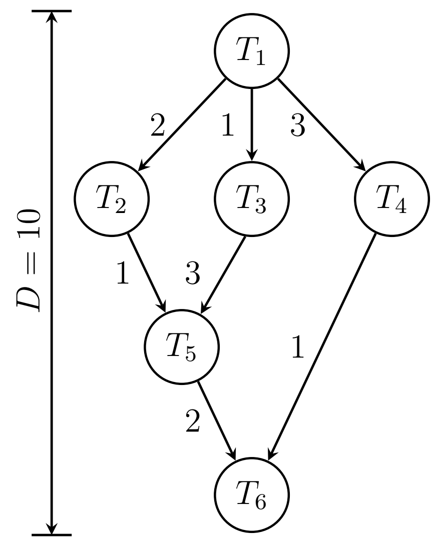


Figure 4.2: *Example of a PTG G*

Task	SL_i	e_{ilr}		QoS_{il}
		P_1	P_2	
T_1	sl_{11}	3	1	4
	sl_{12}	4	2	7
T_2	sl_{21}	1	2	5
	sl_{22}	2	3	9
T_3	sl_{31}	2	3	2
	sl_{32}	3	4	6
T_4	sl_{41}	4	1	3
	sl_{42}	6	3	9
T_5	sl_{51}	1	4	1
	sl_{52}	3	7	8
T_6	sl_{61}	3	2	2
	sl_{62}	4	3	4

Table 4.1: *Values of tasks in Figure 4.2*

Problem Statement: Generate a real-time schedule consisting of a feasible processor assignment, service-level and start time for each task node of a given PTG having a stipulated end-to-end deadline, such that the total QoS acquired by the system is maximized while ensuring that deadline, precedence and resource constraints are not violated on a fully-connected heterogeneous multiprocessor platform.

4.2 Earliest/Latest Start Times for PTG Nodes

Let, t_i^s and t_i^l be the *earliest* and *latest* time steps at which task T_i may start its execution. It may be noted that tasks may be scheduled on any processor and at any service-level. In addition, communication overheads are ignored when a task node T_i and its successor T_j

are scheduled on the same processing node. Hence, in order to get all possible valid ranges of start times, (i) we ignore communication costs associated with edges i.e., $m_{ij} = 0$, and (ii) we consider execution times of tasks corresponding to their lowest service-levels. The t_i^s (ASAP time) and t_i^l (ALAP time) values for task nodes are computed as follows.

ASAP time computation procedure:

1. The in-degree of task node T_i is represented as $indeg(T_i)$.
2. $\forall T_i | indeg(T_i) = 0$, set ASAP time of T_i as, $t_i^s = 1$.
3. ASAP times for the remaining task nodes (except T_i, where $indeg(T_i) = 0$) are recursively determined (downward) as follows:

$$t_i^s = \max_{T_j \in pred(T_i)} (t_j^s + \min_{r \in [1,p]} e_{j1r})$$

   where, $pred(T_i)$ is the set of predecessors of task node T_i.

ALAP time computation procedure:

1. The out-degree of task node T_i is represented as $outdeg(T_i)$.
2. $\forall T_i | outdeg(T_i) = 0$, set ALAP time of T_i as,

$$t_i^l = D - \min_{r \in [1,p]} e_{i1r} + 1$$

3. ALAP times for the remaining task nodes (except T_i, where $outdeg(T_i) = 0$) are recursively determined (upward) as follows:

$$t_i^l = \min_{T_j \in succ(T_i)} (t_j^l - \min_{r \in [1,p]} e_{i1r})$$

   where, $succ(T_i)$ is the set of successors of task node T_i.

Example (contd.): Let us assume the deadline D of PTG G (in Figure 4.2) to be 10 time units. Table 4.2 shows the ASAP and ALAP times corresponding to each task node in G, obtained through the above discussed procedure. For example, ASAP and ALAP times of task node T_1 are 1 and 5, respectively. $\square$

	T_1	T_2	T_3	T_4	T_5	T_6
ASAP	1	2	2	2	4	5
ALAP	5	7	6	8	8	9

Table 4.2: *ASAP & ALAP times of nodes in G (Figure 4.2)*

4.3 ILP Formulation: ILP-SATC

In this section, we present an ILP based solution to the PTG scheduling problem. First, let us consider a set of binary decision variables: $X = \{X_{ilrt} : i = 1, 2, \ldots, n;\ l = 1, 2, \ldots, |SL_i|;\ r = 1, 2, \ldots, p;\ t = 1, 2, \ldots, D\}$. X_{ilrt} is set to 1, if task T_i is executed at service-level sl_{il} on processor P_r and starts its execution at time step t; $X_{ilrt} = 0$, otherwise. We now present the required constraints on the binary variables X to model the scheduling problem.

4.3.1 Unique Start Time Constraint

The start time of each *task* should be unique. That is, each task node T_i must start its execution only at one service-level sl_{il} at a unique time step t on a distinct processor P_r.

$$\forall i \in [1, n] \quad \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} X_{ilrt} = 1 \tag{4.1}$$

4.3.2 Resource Constraint

Resource bounds for processors must be satisfied at each time step. Any processor r can execute at most one task at a given time. In this regard, it may be noted that a task node T_i can only be executing on processor P_r at time t, if it has started at most $t - e_{ilr} + 1$ time steps earlier.

$$\forall t \in [1, D] \text{ and } \forall r \in [1, p] \quad \sum_{i=1}^{n} \sum_{l=1}^{|SL_i|} \sum_{t'=\psi}^{t} X_{ilrt'} \leqslant 1 \tag{4.2}$$

where, $\psi = t - e_{ilr} + 1$.

4.3.3 Dependency Constraint

Precedence-constraints between task pairs must be satisfied. For each directed edge ($T_i \xrightarrow{m_{ij}} T_j$), the execution of T_i along with the transmission of its output message must complete before T_j starts. The message communication cost is neglected if both T_i and T_j are executed on the same processor.

$\forall T_i, T_j \in \mathcal{E} \mid T_i \in pred(T_j)$ and $T_j \in succ(T_i)$

$$\sum_{l=1}^{|SL_i|}\sum_{r=1}^{p}\sum_{t=t_i^s}^{t_i^l}(t + e_{ilr} + m_{ij}) * X_{ilrt} - m_{ij} * Y_{ij} \leqslant \sum_{l=1}^{|SL_j|}\sum_{r=1}^{p}\sum_{t=t_j^s}^{t_j^l} t * X_{jlrt} \tag{4.3}$$

where,

$$Y_{ij} = \sum_{r=1}^{p}\sum_{l_1=1}^{|SL_i|}\sum_{l_2=1}^{|SL_j|}\sum_{t_1=t_i^s}^{t_i^l}\sum_{t_2=t_j^s}^{t_j^l} X_{il_1rt_1} * X_{jl_2rt_2}$$

It may be noted that in the above equation, $Y_{ij} = 1$ when both the predecessor (T_i) and successor (T_j) task nodes are assigned to the same processor P_r. Otherwise, $Y_{ij} = 0$.

4.3.4 Linearization of Non-linear Term

As, $X_{il_1rt_1}$ and $X_{jl_2rt_2}$ are binary decision variables, we linearize their multiplication by introducing another binary decision variable $U_{ijl_1l_2rt_1t_2}$ ($= X_{il_1rt_1} * X_{jl_2rt_2}$) as shown below:

$$Y_{ij} = \sum_{r=1}^{p}\sum_{l_1=1}^{|SL_i|}\sum_{l_2=1}^{|SL_j|}\sum_{t_1=t_i^s}^{t_i^l}\sum_{t_2=t_j^s}^{t_j^l} U_{ijl_1l_2rt_1t_2} \tag{4.4}$$

Now, the non-linear variables $U_{ijl_1l_2rt_1t_2}$ can be linearized using the following four inequalities.

$$X_{il_1rt_1} \geqslant U_{ijl_1l_2rt_1t_2} \tag{4.5}$$

$$X_{jl_2rt_2} \geqslant U_{ijl_1l_2rt_1t_2} \tag{4.6}$$

$$U_{ijl_1l_2rt_1t_2} \geqslant X_{il_1rt_1} + X_{jl_2rt_2} - 1 \tag{4.7}$$

$$U_{ijl_1l_2rt_1t_2} \in \{0,1\} \tag{4.8}$$

4.3.5 Deadline Constraint

All task nodes in the PTG have to complete its execution within the deadline D. It can be noted that, this constraint can be satisfied, if all task nodes with out-degree zero, finishes before deadline of the PTG. The constraint can be written as:

$\forall T_i \mid outdeg(T_i) = 0$

$$\sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} (t + e_{ilr} - 1) * X_{ilrt} \leqslant D \tag{4.9}$$

4.3.6 Objective Function

Our objective is to maximize the overall system reward (QoS) while satisfying the schedulability constraints. The objective function can be written as:

$$Maximize \sum_{i=1}^{n} \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} X_{ilrt} * QoS_{il} \tag{4.10}$$

subject to constraints presented in Equations 4.1 - 4.9.

4.3.7 Complexity Analysis

The complexity of the proposed formulation ILP-SATC can be analyzed in terms of the total number of constraints and the number of variables per constraint. Such an analysis for ILP-SATC is presented in Table 4.3. The total complexity of ILP-SATC (in terms of number of constraints) can be obtained as $O(n + Dp + |\mathcal{E}| + |\mathcal{E}| \times |SL_i|^2 \times p \times D^2)$. Considering $|\mathcal{E}| >> n$, the total complexity becomes $O(|\mathcal{E}| \times |SL_i|^2 \times p \times D^2)$.

Example (contd.): Applying the ILP-SATC on our example PTG G (Figure 4.2), we obtain the schedule represented through the gantt chart depicted in Figure 4.3. This problem generates 6962 constraints and takes 2.56 seconds to solve using the CPLEX optimizer [10]. It may be noted that the schedule assigns unique start times to all tasks

Constraint Type	Equation No.	#Constraints	#Variables Per Constraint
Unique Start Time	4.1	$O(n)$	$O(\|SL_i\| \times p \times D)$
Resource	4.2	$O(D \times p)$	$O(n \times \|SL_i\| \times D)$
Dependency	4.3	$O(\|\mathcal{E}\|)$	$O(\|SL_i\|^2 \times p \times D^2)$
Deadline	4.9	$O(n)$	$O(\|SL_i\| \times p \times D)$
Linearization	4.5	$O(\|\mathcal{E}\| \times \|SL_i\|^2 \times p \times D^2)$	$O(1)$
	4.6	$O(\|\mathcal{E}\| \times \|SL_i\|^2 \times p \times D^2)$	$O(1)$
	4.7	$O(\|\mathcal{E}\| \times \|SL_i\|^2 \times p \times D^2)$	$O(1)$

Table 4.3: *Complexity of ILP-SATC*

and satisfies resource bounds, dependency constraints and deadline. For example, task T_1 has been assigned with a service-level sl_{11}, start time 0 and processor P_2. Since, tasks T_1 and T_2 have been scheduled on the same processor P_2, their communication cost is discarded. On the other hand, tasks T_1 and T_3 have been scheduled on different processors. Subsequently, the execution of T_3 starts after the output from T_1 is communicated to P_1. This communication incurs an overhead of 1 time unit. The total reward obtained for the given PTG G is 31.

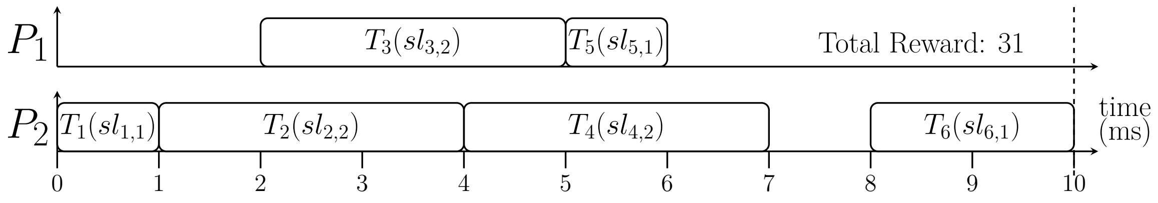


Figure 4.3: *ILP-SATC: Schedule for G (in Figure 4.2) depicted as a gantt chart*

4.4 ILP Formulation: ILP-SANC

It may be observed that the complexity of ILP-SATC presented in the earlier section (Subsection 4.3.7) depends on the number of processors, the deadline and the number of edges associated with a given PTG. In order to improve its scalability, we propose an improved ILP formulation based on the *non-overlapping approach* [9] which sets constraints and variables in such a way that no two tasks executing on the same processor overlap in time. Further, the total number of constraints required to compute a schedule for a PTG becomes independent of the deadline of a given PTG, which helps control complexity of the proposed scheme.

Before presenting ILP-SANC, we first introduce the set of decision variables. Start time of each task T_i is captured by an integer decision variable $S_i \in \mathbb{Z}^+$. The formulation also uses three sets of binary decision variables namely, X_{ir}, Y_{il}, and α_{ij}. Here, variables X_{ir} are used to capture task-to-processor mappings, variables Y_{il} indicate task-to-service-level mappings and variables α_{ij} are used to determine execution precedence order among mutually independent task pairs. Variable $X_{ir} = 1$, if task T_i is assigned onto processor P_r; $X_{ir} = 0$, otherwise. Variable $Y_{il} = 1$, if task T_i executes at service-level sl_{il}; $Y_{il} = 0$, otherwise. Finally, variable $\alpha_{ij} = 1$, if task T_i starts before task T_j; $\alpha_{ij} = 0$, otherwise. Now, we present the set of constraints on the decision variables as required to generate feasible schedules for the given problem at hand.

4.4.1 Unique Resource Assignment:

Each task T_i must execute on a unique processor P_r.

$$\forall i \in [1, n] \quad \sum_{r=1}^{p} X_{ir} = 1 \tag{4.11}$$

4.4.2 Unique Quality-level Assignment:

A task T_i must be assigned to a distinct service-level sl_{il}.

$$\forall i \in [1, n] \quad \sum_{l=1}^{|SL_i|} Y_{il} = 1 \tag{4.12}$$

4.4.3 Dependency Constraint:

Precedence-constraints between task pairs must be satisfied. For each directed edge $(T_i \xrightarrow{m_{ij}} T_j)$, the execution of T_i along with the transmission of its output message must complete before T_j starts. The message communication cost is neglected if both T_i and T_j are executed on the same processor.

$\forall T_i, T_j \in \mathcal{E} \mid T_i \in pred(T_j)$ and $T_j \in succ(T_i)$

$$S_i + \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} U_{ilr} * e_{ilr} + (1 - \sum_{r=1}^{p} Z_{ijr}) * m_{ij} \leq S_j \tag{4.13}$$

where, $U_{ilr} = Y_{il} * X_{ir}$ and $Z_{ijr} = X_{ir} * X_{jr}$. It can be seen that when both tasks T_i and T_j executes on the same processor P_r, Z_{ijr} becomes 1, causing the term containing m_{ij} to vanish. The term $(S_i + \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} U_{ilr} * e_{ilr})$ captures the absolute finish time of T_i at service-level sl_{il} when assigned on P_r.

4.4.4 Linearization of Non-linear Term

The non-linear terms $U_{ilr} = Y_{il} * X_{ir}$ and $Z_{ijr} = X_{ir} * X_{jr}$ in Equation 4.13 can be linearized using constraints 4.14, 4.15, 4.16, 4.17 and constraints 4.18, 4.19, 4.20, 4.21, respectively.

The non-linear term $U_{ilr} = Y_{il} * X_{ir}$ in Equation 4.13 can be linearized using the following four constraints,

$$U_{ilr} \leq Y_{il} \tag{4.14}$$

$$U_{ilr} \leq X_{ir} \tag{4.15}$$

$$U_{ilr} \geq Y_{il} + X_{ir} - 1 \tag{4.16}$$

$$U_{ilr} \in \{0, 1\} \tag{4.17}$$

Similarly, the non-linear term $Z_{ijr} = X_{ir} * X_{jr}$ in Equation 4.13 can be linearized as follows,

$$Z_{ijr} \leq X_{ir} \tag{4.18}$$

$$Z_{ijr} \leq X_{jr} \tag{4.19}$$

$$Z_{ijr} \geq X_{ir} + X_{jr} - 1 \tag{4.20}$$

$$Z_{ijr} \in \{0,1\} \tag{4.21}$$

4.4.5 Non-overlapping Constraints:

No two tasks T_i and T_j can be allocated onto the same processor such that their execution overlap in time. Pairs of tasks which are connected by edges in the PTG share explicit precedence relationships, and the above mentioned dependency constraint naturally enforces the non-overlapping property for them. For the rest of the mutually independent task pairs, the non-overlapping constraint may be presented as:

$\forall T_i, T_j$ which do not share ancestor-descendant relationship ($i = 1, \dots, n-1$; $j = i+1, \dots, n$),

$$S_i + \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} U_{ilr} * e_{ilr} - C * (1 - \alpha_{ij}) - C * (1 - \sum_{r=1}^{p} Z_{ijr}) \le S_j \tag{4.22}$$

$$S_j + \sum_{l=1}^{|SL_j|} \sum_{r=1}^{p} U_{jlr} * e_{jlr} - C * \alpha_{ij} - C * (1 - \sum_{r=1}^{p} Z_{ijr}) \le S_i \tag{4.23}$$

where, $U_{ilr} = Y_{il} * X_{ir}$; $U_{jlr} = Y_{jl} * X_{jr}$; $Z_{ijr} = X_{ir} * X_{jr}$ and C is a large constant.

Constraint 4.22 guarantees that task T_i always finishes before T_j starts ensuring non-overlap. Constraint 4.23 enables this property for the case when T_i starts after T_j. It may also be observed that when both tasks T_i and T_j are assigned to the same processor P_r, the last term in the *left hand side* (LHS) of constraints 4.22 and 4.23 vanish. Otherwise, the constraint may be observed to be trivially satisfied due to the presence of the large constant C. In the LHS of Equation 4.22, the second last term (i.e., $C * (1 - \alpha_{ij})$) vanishes for the case when $\alpha_{ij} = 1$. Otherwise, the constraint gets trivially satisfied due to the constant C. Similarly, when T_i starts after T_j, the term $(C * \alpha_{ij})$ in Equation 4.23 vanishes, trivially satisfying the constraint. When T_j starts after T_i on the same processor P_r (that is, $Z_{ijr} = \alpha_{ij} = 1$), Constraint 4.22 enforces completion of the execution of T_i before the commencement of T_j. We may consider similar arguments for Constraint 4.23 as well.

4.4.6 Deadline Constraint:

All tasks must complete their execution on or before deadline D. This can be satisfied by restricting the finish times of all sink nodes to be at most the deadline D.

$\forall T_i \mid outdeg(T_i) = 0$

$$S_i + \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} (U_{ilr} * e_{ilr}) - 1 \leq D \tag{4.24}$$

where, $U_{ilr} = Y_{il} * X_{ir}$.

4.4.7 Objective Function

The goal of the formulation is to maximize the aggregate QoS acquired by the system while ensuring that none of the above constraints are violated. The objective function is as follows:

$$Maximize \sum_{i=1}^{n} \sum_{l=1}^{|SL_i|} (Y_{il} * QoS_{il}) \tag{4.25}$$

subject to the constraints discussed above (in Equations 4.11 - 4.24).

4.4.8 Complexity Analysis

The complexity of the proposed formulation ILP-SANC can be analyzed in terms of the total number of constraints and the number of variables per constraint. Such an analysis for ILP-SANC is presented in Table 4.4. The total complexity of ILP-SANC (in terms of number of constraints) can be obtained as $O(n^2 \times |SL_i| \times p)$. It may be noted that the complexity of ILP-SANC is independent of the deadline of a PTG.

Example (contd.): Applying ILP-SANC on G in Figure 4.2, the schedule shown in Figure 4.4 is obtained. The reward produced (31) is same as that obtained using ILP-SATC. However, it is worth noting that ILP-SANC generates only 166 constraints and takes 0.82 secs to produce the solution. On the contrary, ILP-SATC produced 6962 constraints and taken 2.56 seconds.

Constraint Type	Equation No.	#Constraints	#Variables Per Constraint
Unique Resource	4.11	$O(n)$	$O(p)$
Unique Quality-level	4.12	$O(n)$	$O(\lvert SL_i \rvert)$
Dependency	4.13	$O(\lvert \mathcal{E} \rvert)$	$O(\lvert SL_i \rvert \times p)$
Non-overlapping	4.22	$O(n^2)$	$O(\lvert SL_i \rvert \times p)$
	4.23	$O(n^2)$	$O(\lvert SL_i \rvert \times p)$
Deadline	4.24	$O(n)$	$O(\lvert SL_i \rvert \times p)$
Linearization	4.14	$O(n^2 \times \lvert SL_i \rvert \times p)$	$O(1)$
	4.15	$O(n^2 \times \lvert SL_i \rvert \times p)$	$O(1)$
	4.16	$O(n^2 \times \lvert SL_i \rvert \times p)$	$O(1)$
	4.18	$O(n^2 \times p)$	$O(1)$
	4.19	$O(n^2 \times p)$	$O(1)$
	4.20	$O(n^2 \times p)$	$O(1)$

Table 4.4: *Complexity of ILP-SANC*

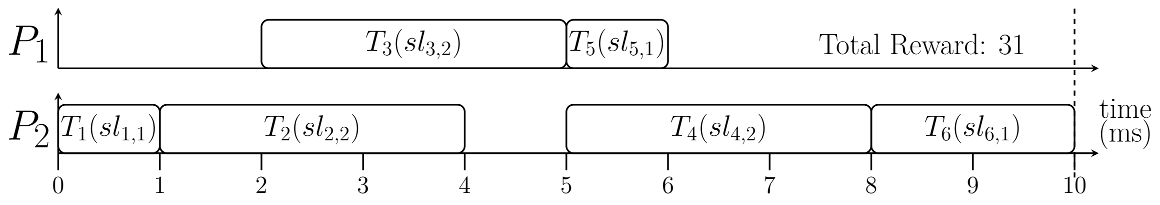


Figure 4.4: *ILP-SANC: Schedule for G (in Figure 4.2) depicted as a gantt chart*

4.5 Experimental Evaluation

In this section, we evaluate the performance of the proposed ILP formulation presented in the earlier section.

Experimental Setup: The experiments have been conducted using benchmark PTGs adopted from [1, 2, 60]. In particular, we considered two real-world applications namely, *Gaussian Elimination* and *Epigenomics.* The PTG representation of *Gaussian Elimination* and *Epigenomics* are shown in Figures 4.5a and 4.5b, respectively. For the scheduling of these two PTGs on a heterogeneous distributed platform, we have varied the following parameters: (1) *Number of processors* $p = \{2, 4, 6, 8\}$, (2) *Com-*

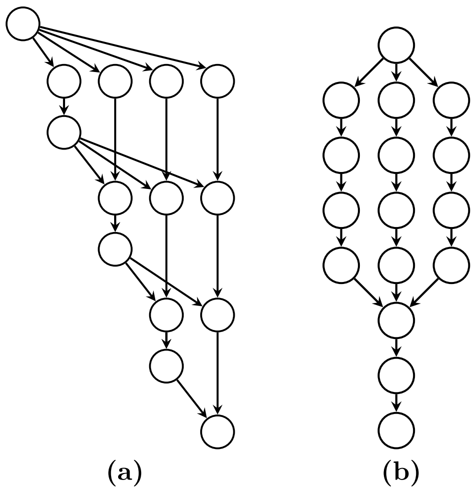
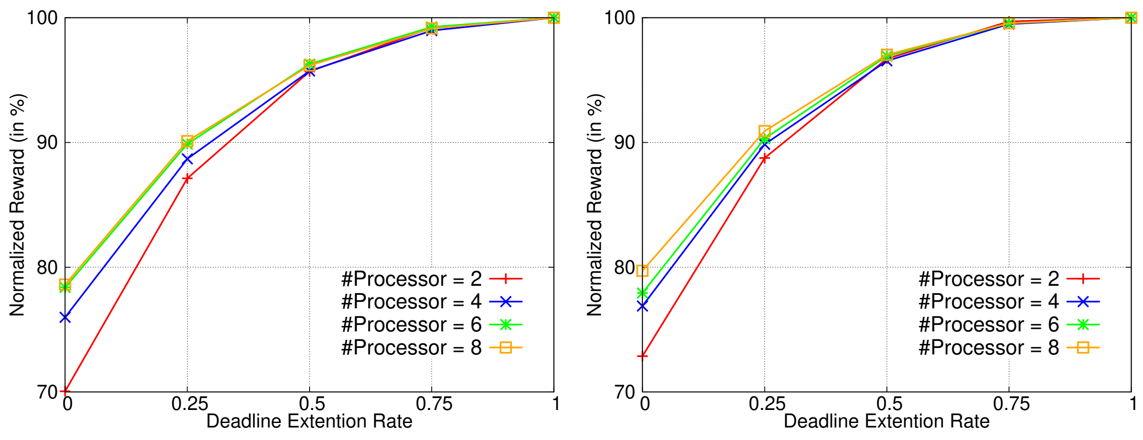


Figure 4.5: *(a) Gaussian Elimination [1], (b) Epigenomics [2]*

munication to Computation Ratio $CCR = \{0.25, 0.5, 0.75, 1, 2\}$ (CCR is the ratio of the average communication cost to the average computation cost. That is, $CCR = \frac{1}{|\mathcal{E}|}\Sigma m_{ij} \Big/ (\frac{1}{n \times p}[\Sigma_{i=1}^{n}\Sigma_{r=1}^{p} e_{i1r}])$). (3) *Number of service-levels* of each task T_i is taken as, $|SL_i| = 3$, (4) *Execution time* e_{i1r} of each task node T_i at its base service-level for processor P_r, is taken randomly from a uniform distribution within the range 10 ms to 30 ms. The execution time (e_{ilr}) for non-base service-levels (starting from level 2) of the tasks are assigned uniform random values bounded between 110% and 130% of the execution times ($e_{i(l-1)r}$) corresponding to their immediately lower service-levels, (5) The rewards (QoS_{il}) for any task T_i, increase monotonically as service-levels become higher. The values of the rewards have been chosen randomly from the range 1 to 200, while ensuring that the random reward value for a task at a given service-level is higher than the reward values at lower service-levels. (6) *Communication time* m_{ij} corresponding to each edge in the PTG has been generated from a uniform random distribution within the range 10 ms to 30 ms. The obtained communication times are then appropriately scaled to maintain desired CCR, (7) *Deadline* for a PTG is obtained from the makespan outputs computed by applying the *list scheduling* based heuristic scheme PEFT [1] on the given PTG. In particular, we compute two makespan D_L and D_H by setting all task nodes at their base and highest service-levels, respectively. Finally, the actual deadline D for the

PTG is randomly selected from a uniform distribution in the range $[D_L, D_H]$. All experiments are carried-out using the CPLEX optimizer [10] version 12.6.2.0, executing on a system having Intel(R) Xeon(R) CPU running Linux Kernel 3.10.0-693.21.1.el7.x86_64.

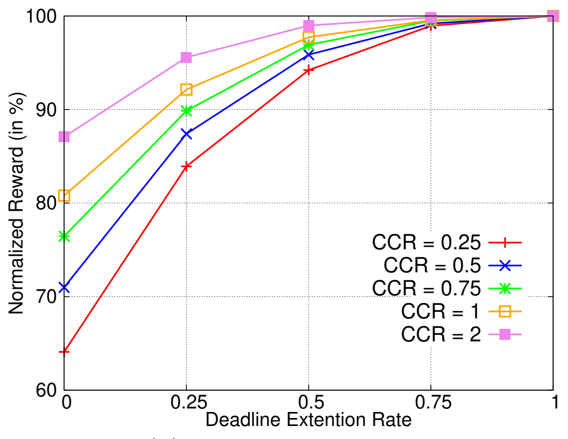
Performance Metrics: Four metrics have been used for evaluating the designed ILP based scheduling strategies: (1) *Normalized Reward*: NR (in %) $= \frac{R_{ACT}}{R_{MAX}} \times 100$, where, R_{ACT} is the actually obtained reward and R_{MAX} is the maximum possible reward for the PTG. (2) *Deadline extension Rate* (DR) determines the *extended deadline* of a given PTG as: $D = D_L + ((D_H - D_L) \times DR)$, where $DR \in \{0, 0.25, 0.5, 0.75, 1.0\}$. For example, the different extended deadlines corresponding to various values of DR for a PTG with $D_L = 20$ and $D_H = 40$ are 20, 25, 30, 35, 40. (3) *Running time* (in seconds): Total time taken to compute the solution for a given PTG. (4) *Percentage of tasks upgraded*: Given a PTG and an input x ($\in \{0, 25, 50, 75, 100\}$), we have selected $x\%$ of task nodes in the PTG and upgraded them to their highest service-levels. Specifically, $x\%$ of the tasks which have the highest *reward per unit execution time (RPE)* values have been chosen for service-level enhancement. Here, RPE of a task T_i is defined as the ratio of the difference in reward to the difference in execution time, when T_i is upgraded to the highest service-level from its base level.



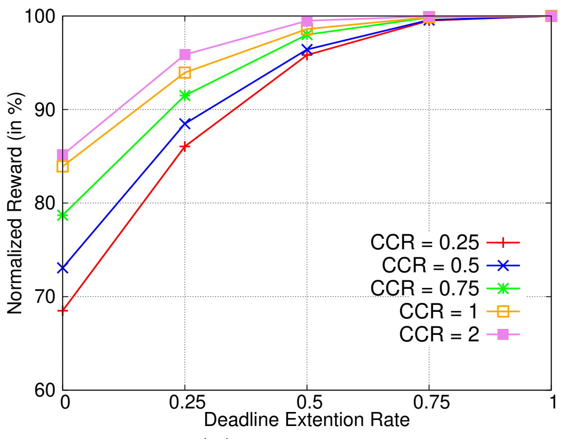
(a) *Gaussian Elimination* **(b)** *Epigenomics*

Figure 4.6: *Effect of varying processors.*

Experiment-1: Varying the number of processors: In this experiment, we vary the number of processors (p) from 2 to 8, while fixing CCR to 0.5. Figure 4.6 depicts the results for this experiment. It may be noted that for any given deadline, the normalized reward NR *increases as the number of processors becomes higher*. This happens because the residual system capacity increases with increasing #processors and this capacity is utilized by the system in order to enhance task service-levels, resulting in higher NR values. For example, in Gaussian Elimination PTG with $DR = 0.25$ (Figure 4.6a), NR values for $p = 2$ and $p = 6$ are $\sim 87\%$ and $\sim 90\%$, respectively.



(a) *Gaussian Elimination*



(b) *Epigenomics*

Figure 4.7: *Effect of varying CCR*

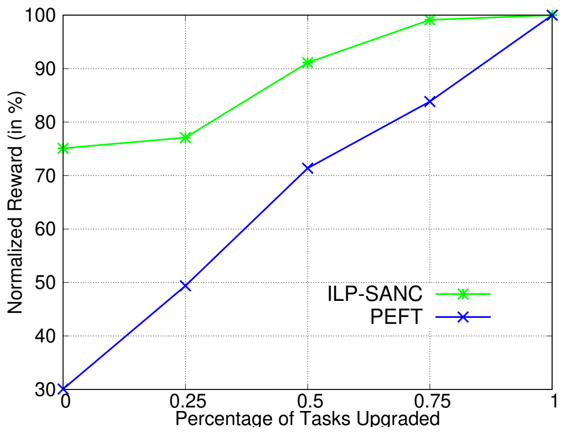
Experiment-2: Varying *CCR*: We vary CCR from 0.25 to 2, while fixing p to 2. Figure 4.7 depicts the results for this experiment. For fixed values of p, n and D, higher values of CCR imply lower computation demands of the task nodes on processor resources at any service-level. Such lower computation demands in turn, naturally enhances the possibility of task service-level upgradation. Consequently, this leads to an increase in the obtained rewards, NR. For example, in Epigenomics PTG with $DR = 0.25$ (Figure 4.7b), the normalized rewards obtained for $CCR = 0.25$ and $CCR = 2$ are $\sim 86\%$ and $\sim 96\%$, respectively.

Experiment-3: Comparing ILP-SATC and ILP-SANC: We set p to 2 and CCR to 0.5. The total amount of time taken by both ILP-SATC and ILP-SANC, when the deadline is varied from D_L to D_H, is shown in the Table 4.5. Observing the

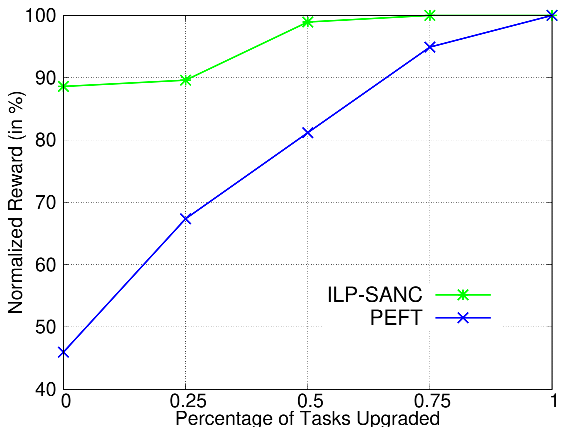
PTG	ILP Version	Deadline Extension Rate				
		0	0.25	0.5	0.75	1
Gaussian Elimination	ILP-SATC	16m:30s	2h:46m:43s	@	@	@
	ILP-SANC	2.6s	2.7s	1.6s	0.6s	0.4s
Epigenomics	ILP-SATC	@	@	@	@	@
	ILP-SANC	45.7s	55s	23.1s	2.3s	0.4s

Table 4.5: *Running time of ILP-SATC and ILP-SANC. The symbol @ represents running times greater than 24 hours*

results obtained for the Gaussian Elimination application, it may be clearly seen that run-times for ILP-SATC significantly increases with larger deadlines. This indicates ILP-SATC's strong dependence on the value of the deadline considered. On the other hand ILP-SANC exhibits drastically lower run-times for all cases.



(a) *Gaussian Elimination*



(b) *Epigenomics*

Figure 4.8: *Comparison with PEFT [1]*

Experiment-4: Comparison with the state-of-the-art: This experiment compares ILP-SANC against the *list based* heuristic scheduling algorithm PEFT [1]. The essential objective of PEFT is to minimize makespan corresponding to a task graph in which all tasks have only a single service-level. Therefore, as PEFT is task service-level oblivious, in order to apply PEFT within our framework, service-levels of all tasks must be fixed before its application. After assigning selected service-levels to task nodes, PEFT is run on the PTG and the resulting normalized reward NR and makespan values are noted. ILP-SANC is then executed on the same PTG with the makespan value delivered by PEFT, as deadline. To improve normalized reward values for PEFT, we have selectively chosen higher service-levels for those tasks which deliver higher reward gains with respect to additional execution time consumed; that is, tasks with larger RPE values have greater priorities towards higher service-level assignment. The experimental results are depicted in Figure 4.8. It may be observed that our proposed ILP based scheme is able to achieve higher normalized rewards compared to PEFT for any given deadline bound, unless the deadline is so relaxed that PEFT is also able to assign highest service-levels to all tasks.

4.6 Case Study: Adaptive Cruise Controller

To illustrate the generic applicability of our proposed strategy to real world designs, we present a case study using an *Adaptive Cruise Controller* (ACC) application present in automotive systems. ACC automatically maintains a safe distance between two cars [5]. Figure 4.9a shows the block diagram of ACC adopted from [3] and Figure 4.9b depicts its corresponding PTG representation. The PTG consists of 20 task nodes $\{T_1, T_2, \ldots, T_{20}\}$. We assume that this PTG is to be scheduled on a distributed platform consisting of three heterogeneous processors $\{P_1, P_2, P_3\}$. The computation times of the task nodes for the given heterogeneous platform are listed in Table 4.6. We assume the deadline to be 150 ms.

We have employed ILP-SANC to compute a schedule that maximizes overall achievable reward for the PTG in Figure 4.9b. Implementation of ILP-SANC using the CPLEX optimizer takes $\sim$ 13.42 secs to produce the solution. It may be noted that ILP-SATC implemented using CPLEX is unable to produce a solution for the same PTG within our stipulated time cap of 24 hours. A gantt chart representation of the resulting schedule is shown in Figure 4.10. The observations may be summarized as follows:

- *Overall reward* for the final schedule is 1954.

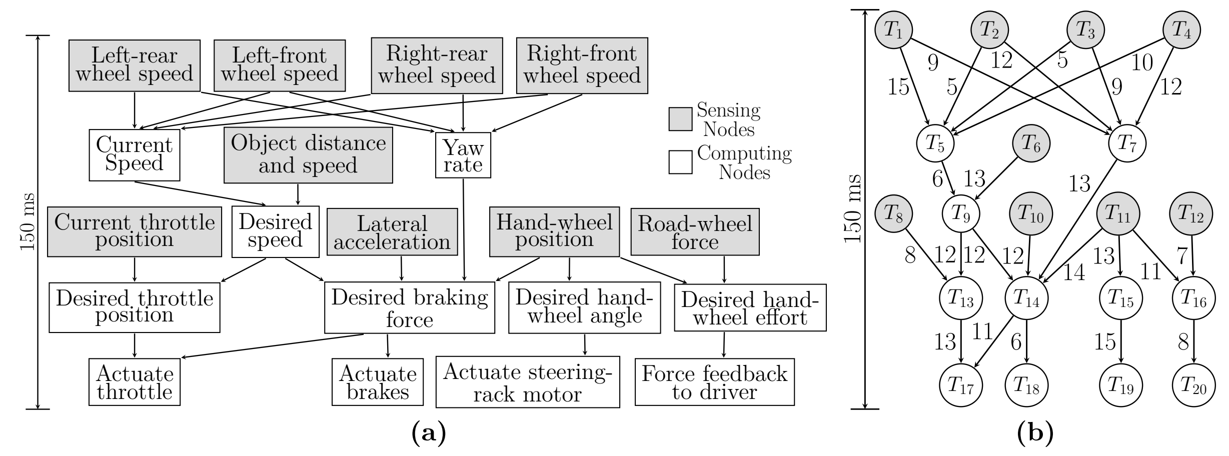


Figure 4.9: *ACC application's (a) Block Diagram [3], (b) PTG representation*

		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}	T_{18}	T_{19}	T_{20}
sl_{i1}	P_1	25	16	19	10	27	30	15	29	12	28	21	24	21	10	20	25	18	19	18	10
	P_2	23	14	30	25	16	13	30	30	22	17	12	25	26	14	21	24	26	17	19	18
	P_3	25	21	23	29	13	28	22	12	30	20	16	21	30	10	25	16	30	29	30	14
	QoS_{i1}	10	35	13	10	14	6	67	58	17	15	35	7	43	72	23	6	25	45	51	4
sl_{i2}	P_1	-	-	-	-	33	-	16	-	13	-	-	-	24	12	23	28	20	23	19	12
	P_2	-	-	-	-	20	-	33	-	24	-	-	-	30	17	24	27	29	21	21	23
	P_3	-	-	-	-	16	-	24	-	33	-	-	-	35	12	29	18	34	36	33	18
	QoS_{i2}	-	-	-	-	85	-	114	-	174	-	-	-	82	120	98	90	53	86	135	95
sl_{i3}	P_1	-	-	-	-	39	-	20	-	15	-	-	-	28	13	29	31	22	25	23	13
	P_2	-	-	-	-	24	-	42	-	29	-	-	-	36	19	30	29	32	23	25	25
	P_3	-	-	-	-	19	-	30	-	40	-	-	-	42	13	37	19	37	39	40	20
	QoS_{i3}	-	-	-	-	146	-	181	-	186	-	-	-	158	195	182	175	196	102	144	100

Table 4.6: *Computation time (in ms) of task nodes*

- *Avoidance of message transmission time*: Messages are not transmitted when a task node is scheduled on the same processor as its successor task node. For example, it may be observed that the message transmission time of 8 *ms* between T_{16} and T_{20} is avoided since both of them are assigned on processor P_3.

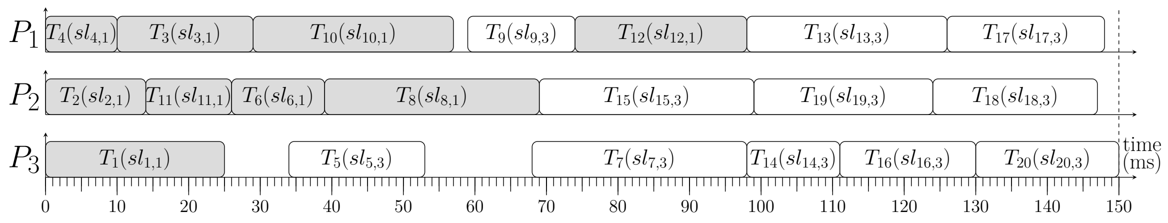


Figure 4.10: *Gantt chart representation of the schedule for the PTG (in Figure 4.9b)*

- *Correct satisfaction of all constraints*: It can be seen from the schedule that, (i) all nodes have a unique start time, (ii) each node is assigned with a distinct service-level, (iii) resource bounds are satisfied at all time instants, and (iv) dependency constraints among nodes have been respected. For example, task T_7 starts its execution at time instant 68 *ms* with service-level 3, on processor P_3. While executing T_7, P_3 does not allow the execution of any other task on it, thus respecting resource constraints. On the completion of T_7, its successor task node T_{14} is also scheduled on P_3. Since, T_7 and T_{14} are scheduled on the same processor, the communication overhead of 13 *ms* has been avoided.

- *Modeling Heterogeneity*: Each node in the PTG has consumed distinct amounts of time depending on the resource on which it is assigned. For example, task T_2 has taken 14 *ms* because it has been assigned on P_2.

4.7 Summary

In this chapter, we have considered the problem of computing optimal schedules for PTGs with multiple service-levels, executing on fully-connected distributed systems consisting of heterogeneous processors. The scheduler construction methodology uses *Integer Linear Programming* as the underlying solution technique. Two distinct solution strategies ILP-SATC and ILP-SANC have been proposed. ILP-SATC is based on the explicit manipulation of the mobilities of task nodes between their earliest and latest start times. On the other hand, the solution strategy proposed in ILP-SANC ensures that the executions of no two tasks overlap in time on the same processor, instead of explicitly relying on task mobility based manipulations. We have shown that the strategy followed in ILP-SANC is able to significantly reduce the number of constraints and variables that are required by the formulation and this results in a drastic improvement in scalability. An extensive set of simulation based experiments have been conducted to evaluate the performance of the two ILPs. The solution strategies have also been compared with a state-of-the-art scheme [1]. Finally, a case study on a *Adaptive Cruise Controller* (ACC) application has been presented. In the next chapter, we present heuristic scheduling mechanisms for the same problem as discussed in this chapter.

Chapter 5

Heuristic PTG Scheduling Strategies on Heterogeneous Distributed Systems

The previous chapter deals with the optimal scheduling mechanism of a real-time system modeled as PTG executing on a fully connected distributed heterogeneous platform. Two distinct ILP based solution strategies ILP-SATC and ILP-SANC have been proposed. Though ILP-SANC shows appreciable improvements in terms of scalability over the ILP-SATC, it still suffers from high computational overheads (in terms of running time) as the number of nodes in a PTG and/or the number of resources, increase. Therefore in this chapter, two low-overhead heuristic algorithms namely, G-SAQA and T-SAQA, are proposed for the same problem as discussed in the previous chapter (Chapter 4). The base-line heuristic, G-SAQA, is faster but returns moderately good solutions. T-SAQA extends G-SAQA and deliver significantly better solution, albeit at the cost of slightly higher time complexity. Then, both the heuristic schemes have been comprehensively evaluated through an extensive set of experiments over benchmark PTGs as well as using randomly generated PTGs. Finally, we demonstrate the practical applicability of the G-SAQA and T-SAQA using a real-world case study on a *Traction Controller* (TC) application and conclude the chapter.

5.1 The Task and Platform Models

The Task and Platform Models discussed in Section 4.1 of Chapter 4 are re-introduced here in Section 5.1 for better readability.

This work considers a periodic application represented as a PTG $G = (T, \mathcal{E})$ to be executed on a fully-connected *heterogeneous multiprocessor platform* consisting of a set of processors $P = \{P_1, P_2, \ldots, P_p\}$. Figure 5.1 shows the pictorial representation of the fully-connected *heterogeneous multiprocessor platform* as considered here. Each processor has its own private memory. For any given processor, task execution and communication with other processors can be conducted simultaneously, without any contention. Specifically, we assume that tasks on different processors communicate by transmitting data from the source processor via the fully-connected network to the local memory of the receiving processor. On the other hand, intra-processor communication is realized through the reading and writing of variables stored in the local memory of the processor.

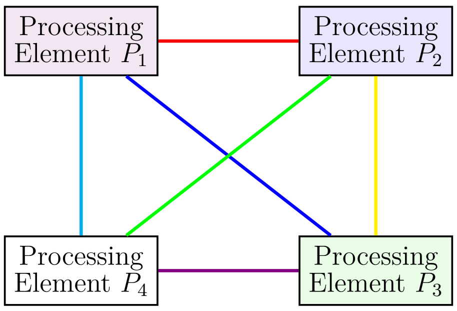


Figure 5.1: *Fully connected multiprocessor system*

- $T = \{T_1, T_2, \ldots, T_n\}$ represents n task nodes.
- $\mathcal{E} \subseteq T \times T$ denotes the edge-set that describes *precedence-constraints* between pairs of tasks in T. The symbol m_{ij} is used to denote the communication cost associated with the edge $T_i \rightarrow T_j$; $m_{ij} = 0$, when T_i and T_j are assigned to the same processor. All tasks/messages in PTG G execute/transmit non-preemptively.
- Each task T_i has $SL_i = \{sl_{i1}, sl_{i2}, \ldots, sl_{i|SL_i|}\}$ alternative service-levels. Associated with a given service-level sl_{il} of task T_i, there exists p possibly distinct worst-case execution times e_{ilr} ($i \in [1, n]$, $l \in [1, |SL_i|]$, $r \in [1, p]$) corresponding to the p heterogeneous processors. There is a *reward* QoS_{il} which is obtained on successful completion of every instance of T_i.

- Given any two service-levels sl_{il} and $sl_{il'}$ of task T_i such that $l < l'$, the execution times on any processor P_r are related as $e_{ilr} \leq e_{il'r}$. However, the processors being heterogeneous, the execution times of a task on two different processors are completely unrelated.

- The execution times of a task T_i on processor P_r for all service-levels is set to ∞, to model the scenario in which the execution of T_i is infeasible on P_r.

- Each instance of application G has an associated *deadline* D. Two consecutive instances of G are separated by a *period* π. We assume the deadline to be *implicit*, i.e., $D = \pi$.

To denote in-degree and out-degree of the task node T_i, we use the notations $indeg(T_i)$ and $outdeg(T_i)$, respectively. Given a pair of task nodes $\langle T_i, T_j \rangle$, T_i (T_j) is said to be the predecessor (successor) of T_j (T_i) if there is an edge ($T_i \rightarrow T_j$) form T_i to T_j. To denote predecessor and successor of the task node T_i, we use the notations $pred(T_i)$ and $succ(T_i)$, respectively. Similarly, given two tasks T_i and T_j, T_i (T_j) is said to be the *ancestor* (*descendant*) of T_j (T_i), if $T_i \prec T_j$ i.e., there exists a path from T_i to T_j, in the PTG.

Example: An example of a PTG G and its task parameters are shown in Figure 5.2 and Table 5.1, respectively. PTG G consists of 7 tasks, each task having 2 service-levels. Each task T_i may have distinct execution times e_{ilr} on each processor P_r for any given service-level sl_{il} along with associated reward QoS_{il}. For example, at service-level sl_{11}, task T_1 has execution times $e_{111} = 3$ and $e_{112} = 1$ on processors P_1 and P_2, respectively, and an associated $QoS_{11} = 2$. An edge between T_1 and T_2 has an associated communication cost of 1.

Problem Statement: Generate a real-time schedule consisting of a feasible processor assignment, service-level and start time for each task node of a given PTG having a stipulated end-to-end deadline, such that the total QoS acquired by the system is maximized while ensuring that deadline, precedence and resource constraints are not violated on a fully-connected heterogeneous multiprocessor platform.

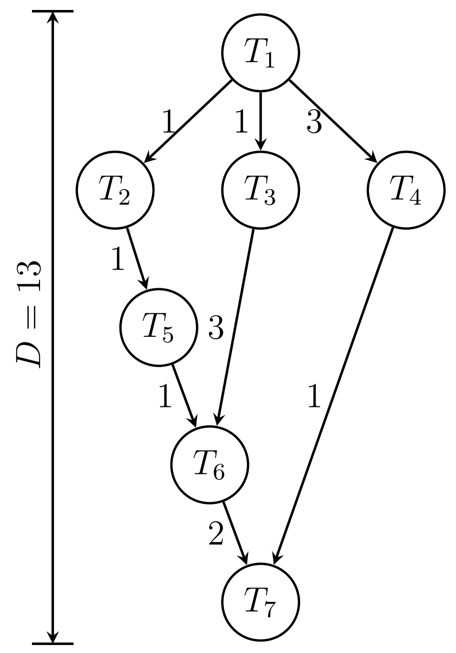


Figure 5.2: *Example of a PTG G*

Task	SL_i	e_{ilr}		QoS_{il}
		P_1	P_2	
T_1	sl_{11}	3	1	2
	sl_{12}	4	2	4
T_2	sl_{21}	1	2	1
	sl_{22}	4	3	9
T_3	sl_{31}	4	3	2
	sl_{32}	5	5	5
T_4	sl_{41}	4	3	3
	sl_{42}	6	4	7
T_5	sl_{51}	3	4	1
	sl_{52}	5	6	4
T_6	sl_{61}	1	4	1
	sl_{62}	3	7	4
T_7	sl_{71}	3	2	2
	sl_{72}	4	3	3

Table 5.1: *Values of tasks in Figure 5.2*

5.2 Heuristic Algorithms

Typically, *list scheduling* based heuristic techniques have been employed to compute feasible schedules for PTGs executing on heterogeneous platforms. Some examples of this class of techniques include the HEFT [6], PEFT [1] and HSV [11] algorithms. They attempt to construct a static-schedule for the given PTG with the objective of minimizing the overall schedule length, while satisfying resource and precedence constraints. On the contrary, our work deals with the problem of scheduling PTGs consisting of task nodes with multiple service-levels, with the objective of maximizing overall system level *reward*, while satisfying the deadline constraint associated with a given application. For this purpose, we devise two distinct types of heuristic algorithms namely, (i) G-SAQA and (ii) T-SAQA. While the solution qualities delivered by T-SAQA is at least as good as G-SAQA, the computational overheads associated with T-SAQA is significantly higher than G-SAQA. Both G-SAQA and T-SAQA internally make use of PEFT, a recent state-of-the-art algorithm, to compute a baseline schedule by setting all task nodes at their base service-level. Since PEFT attempts to minimize schedule length, the resulting schedule length may be marked by unutilized slack time before deadline. This slack may then be used to upgrade service-levels of task nodes. In the next subsection, we discuss the details of G-SAQA.

5.2.1 Global Slack Aware Quality-level Allocator (G-SAQA)

Algorithm 2 depicts a step-wise description of G-SAQA. The G-SAQA algorithm starts by using PEFT to compute task-to-processor mappings as well as start and finish times of tasks, based on task execution times associated with their base service-levels (line nos. 1 to 3). If length of the obtained PEFT schedule violates deadline, then the algorithm terminates as generation of a feasible schedule is not possible (line nos. 4 to 6). Otherwise, the available global slack ($slack_g$ = Deadline − PEFT makespan) is used to enhance the tasks' assigned service-levels in an endeavour to maximize achievable reward while retaining task-to-processor mappings as provided by PEFT. For this purpose, we first construct an *assignment PTG* G' corresponding to G. First, G' has the same node set as G and also retains all edges in G. In addition, each node $T_i \in G'$ is labeled with information related to its processor assignment, start time and finish time, as provided by the PEFT schedule (line no. 7). Further, if two mutually independent tasks T_i and T_j are scheduled on the same processor P_r, and T_j is scheduled immediately after T_i on P_r, then an edge $T_i \xrightarrow{m_{ij}=0} T_j$ is introduced in the assignment PTG G' (line nos. 8 to 10). This edge has been added to ensure that the execution of tasks assigned on P_r do not get overlapped as a side-effect of service-level upgradation.

Now, the available global slack ($slack_g$) needs to be distributed among task nodes in the PTG to upgrade their service-levels, such that the overall reward is maximized. This upgradation happens in a service-level by service-level manner, starting with all tasks situated at their base service-levels. At each step, the most eligible task is selected (from the task set) for service-level upgradation by one. The selection of this task is based on a prioritization key called, $cost_i$ which is defined as follows (line nos. 12 to 13):

$$cost_i = \frac{QoS_{i(l+1)} - QoS_{il}}{e_{i(l+1)r} - e_{ilr}} \tag{5.1}$$

Here for any given task T_i, $cost_i$ is defined as the additional reward received by the system per unit additional consumed resource (w.r.t. service-level upgradation from sl_{il} to $sl_{i(l+1)}$) while adhering to the mapping provided by the PEFT schedule. To select a node that gives maximum reward (by utilizing the available slack) among all nodes in the PTG, G-SAQA creates a max-heap of tasks using $cost_i$ as the key (line no. 13). Then, G-SAQA proceeds by repeatedly extracting the task (say, T_i) at the root of the heap. After extraction, G-SAQA checks whether it is possible to upgrade T_i's service-level by utilizing the available global slack $slack_g$. If the check results in success, T_i's service-level is incremented by 1 and the available slack time is decremented by the additional resource consumed by T_i (line nos. 18 to 20). To adjust for the additional execution time required at the enhanced service-level of T_i, G-SAQA recursively updates the start times of all successor task nodes of T_i. If task T_i has not yet reached its maximum service-level, then its key $cost_i$ is updated and reinserted into the max-heap. This process continues either until all tasks have reached their maximum service-levels, or the available global slack is completely exhausted (line nos. 15 to 23).

Time Complexity of G-SAQA: Assignment of base service-levels to tasks takes $O(n)$ iterations (line nos. 1 to 2). Next, the complexity of computing a PEFT schedule is $O(n^2 \times p)$ (line no. 3) [1]. Construction of the assignment PTG G' from G and the PEFT schedule takes $O(n+|\mathcal{E}|)$ time (line no. 7). The overhead of adding dummy edges between mutually independent task pairs (T_i, T_j) in G' is $O(n^2)$ (line nos. 8 to 10). Computation overheads associated with the calculation of $slack_g$ (line no. 11) and $cost_i$ for all tasks (line nos. 12 to 13) are $O(1)$ and $O(n)$, respectively. Formation of the initial max-heap takes $O(n)$. The service-level upgradation process takes $O(\sum_{i=1}^{n} |SL_i| \times n \times log(n))$ (line nos. 15 to 23) and this includes $O(n)$ required for updating the start and finish times of all descendant nodes of T_i (line no. 21). Hence, the time complexity of the G-SAQA algorithm is $O(\max\{(\sum_{i=1}^{n} |SL_i| \times n \times log(n)), (n^2 \times p)\})$.

Example (contd.): The PEFT schedule of the PTG G in Figure 5.2 is shown in Table 5.2. The makespan of the PEFT schedule is 11 units ($\leq$ 13 units; $slack_g = 2$ units). From the given PEFT schedule, it can be seen that mutually independent tasks

ALGORITHM 2: G-SAQA

```
Input: A PTG consisting of $n$ tasks, $p$ fully-connected heterogeneous processors
Output: A feasible task schedule that maximizes system level reward
1  forall tasks $T_i$ do
2      Assign minimum service-level $sl_{i1}$ to $T_i$
3  Compute a PEFT schedule to determine start time, finish time and processor
     mapping of each task
4  if PEFT makespan violates the deadline constraint $D$ then
5      Declare "generation of a feasible schedule is not possible"
6      return
7  Construct assignment PTG $G'$ from the original PTG $G$ using PEFT schedule
8  forall mutually independent task pairs ($T_i$, $T_j$) do
9      if $T_i$ and $T_j$ are scheduled on the same processor $P_r$ then
10         Add an edge $T_i \xrightarrow{m_{ij}=0} T_j$ in $G'$, if $T_j$ is scheduled immediately after $T_i$
             on $P_r$ and vice versa
11 Compute global slack, $slack_g$ = Deadline − PEFT makespan
12 forall tasks $T_i$ do
13     Compute $cost_i$ using Equation 5.1
14 Make max-heap of tasks using $cost_i$ as key
15 while max-heap is non-empty do
16     Remove root node $T_i$ to possibly upgrade its service-level
17     Compute additional computation demand: $\Delta e_i \leftarrow e_{i(l+1)r} - e_{ilr}$
18     if $\Delta e_i \leq slack_g$ then
19         Upgrade service-level of $T_i$ from $sl_{il}$ to $sl_{i(l+1)}$
20         Update the available global slack: $slack_g \leftarrow slack_g - \Delta e_i$
21         Update start and finish times of all descendant nodes of $T_i$ by $\Delta e_i$
22         if current service-level < $|SL_i|$ then
23             Compute $cost_i$ (Equation 5.1) and reinsert $T_i$ into the max-heap
```

T_3 and T_4 are mapped onto the same processor P_2 and the execution of T_4 immediately succeeds T_3. Hence, a zero weighted edge from T_3 to T_4 is added in G' (Figure 5.3). The initial values of $cost_i$ corresponding to the seven tasks are as follows: $cost_1 = 2, cost_2 = 2.67, cost_3 = 1.5, cost_4 = 4, cost_5 = 1.5, cost_6 = 1.5$ and $cost_7 = 1$. A max-heap is built using these key values. The task T_4 with the highest key value (currently, at the root of the max-heap) is extracted from the heap and its service-level is enhanced from sl_{41} to sl_{42} as the additional computation requirement, $\Delta e_4 = 4 - 3 = 1$, can be satisfied by the available global slack, $slack_g = 2$. The global slack time is decremented to $slack_g = slack_g - \Delta e_4 = 2 - 1 = 1$, and the start time of the successor task

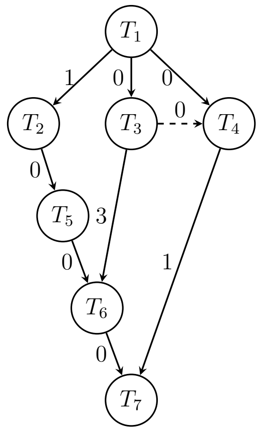


Figure 5.3: *Assignment PTG G' obtained from PTG G and the PEFT schedule*

Task (T_i)	Start time (S_i)	Finish time (F_i)	T_i to P_r
T_1	0	1	P_2
T_2	2	3	P_1
T_3	1	4	P_2
T_4	4	7	P_2
T_5	3	6	P_1
T_6	7	8	P_1
T_7	8	11	P_1

Table 5.2: *PEFT schedule of the PTG in Figure 5.2*

node T_7 is increased by Δe_4 i.e., $S_7 = S_7 + \Delta C_4 = 8 + 1 = 9$. Since T_4 has reached to its highest service-level, it is not reinserted back into the heap.

Next T_2, the task with the highest key, is extracted from the heap but its service-level is not enhanced as, the available global slack, $slack_g = 1$ cannot fulfill the additional computation requirement, $\Delta e_2 = 4 - 1 = 3$ and T_2 is not reinserted into the heap. Next T_1, the task with the highest key, is extracted from the heap and its service-level is enhanced from sl_{11} to sl_{12}, since $\Delta e_1 = 2 - 1 = 1$ is less than equal $slack_g = 1$. Subsequently, the global slack is updated as, $slack_g = slack_g - \Delta e_1 = 1 - 1 = 0$ and the start times of all successor nodes T_2 to T_7 are increased by Δe_1. Thus, the updated start times become, $S_2 = 3, S_3 = 2, S_4 = 5, S_5 = 4, S_6 = 8$ and $S_7 = 10$. It may be observed that as a result of T_1's quality enhancement, the global slack gets exhausted to 0 and hence, the algorithm terminates. The gantt chart representation of the final schedule is given in Figure 5.4. The aggregate reward returned by G-SAQA is 18. □

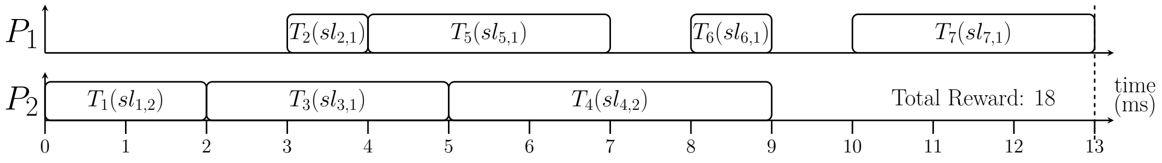


Figure 5.4: *G-SAQA: Gantt chart representation of the schedule for G*

Though G-SAQA follows an intuitive design flow, it only considers global slack (=

deadline − PEFT makespan) to upgrade service-levels of tasks in the PTG. However, a closer look at the PEFT schedule reveals that there exists gap within the scheduled nodes of the PTG which could be used along with the global slack to achieve better performance in terms of service-levels and delivered rewards compared to G-SAQA. It may also be possible to consolidate multiple small gaps within the PEFT schedule into larger consolidated slack which may be used to further improve performance in terms of achieved rewards. Therefore, the *total slack* available with a task at any given time comprises of the global slack along with the maximum consolidated inter-node gap between the task and its successor on its assigned processor in the PEFT schedule. This *total slack* associated with a task T_i in PTG G at any given time, can be computed by finding the difference between the earliest (ASAP) and latest (ALAP) time instants at which T_i's execution can be started while adhering to the resource-mapping provided by PEFT. In Section 4.2 of Chapter 4, we have shown the ASAP/ALAP computation procedure for tasks. Here we present an extension of this procedure. Unlike Chapter 4 where task-to-processor mapping was unknown, in this section we conduct resource-aware ASAP/ALAP computation. The detailed steps required for the computation of the total slack $slack_i$ associated with each task T_i is as follows:
Resource aware ASAP time computation procedure:

1. For source nodes T_i $(\forall T_i | indeg(T_i) = 0)$, set ASAP time as, $t_i^s = 0$.

2. ASAP times for the remaining task nodes T_i are recursively determined as follows:

$$t_i^s = \max_{T_j \in pred(T_i)} (t_j^s + e_{jlr} + m_{ji})$$

   where, $pred(T_i)$ is the set of predecessors of task node T_i and e_{jlr} is the execution time of task T_j at service-level sl_{jl} on processor P_r (T_j is assigned on processor P_r by PEFT).

Resource aware ALAP time computation procedure:

1. For sink nodes T_i $(\forall T_i | outdeg(T_i) = 0)$, set ALAP time of T_i as, $t_i^l = D - e_{ilr}$

2. ALAP times for the remaining task nodes T_i are recursively determined as follows:

$$t_i^l = \min_{T_j \in succ(T_i)} (t_j^l - e_{ilr} - m_{ij})$$

   where, $succ(T_i)$ is the set of successors of task node T_i and e_{ilr} is the execution time of task T_i at service-level sl_{il} on processor P_r (T_i is assigned on processor P_r by PEFT).

Slack time computation: The total slack of task T_i (while adhering to the mapping provided by the PEFT schedule) is computed as,

$$slack_i = t_i^l - t_i^s \tag{5.2}$$

With the above insights on the total task-level slacks available in a PTG, we propose another heuristic namely, T-SAQA with the objective of achieving better performance compared to G-SAQA. The details of T-SAQA is presented in the following subsection.

5.2.2 Total Slack Aware Quality-level Allocator (T-SAQA)

Algorithm 3 depicts a step-wise description of T-SAQA. The basic structure of T-SAQA is the same as that of the G-SAQA algorithm. Specifically similar to G-SAQA, T-SAQA (line nos. 1 to 18) also performs the steps, (i) Computation of the PEFT schedule for PTG G, (ii) construction of assignment PTG G', (iii) formation of max-heap using $cost_i$ as the key, and (iv) service-level upgradation of task node T_i. However, T-SAQA differs from G-SAQA in the way it updates the start times of T_i's descendants and slacks associated with the task nodes in G'. In particular, G-SAQA uniformly delays the start times of all descendant nodes of T_i and reduces the global slack value by the same amount. In this regard, it may be emphasized that T-SAQA works with distinct total slack values ($slack_i$) associated with the task nodes (T_i) in G', instead of using a single global slack pool. By harnessing the total slacks available with individual task nodes, T-SAQA updates the start and finish times of only those descendant task nodes of T_i, whose start times are impacted due to the service-level upgradation of T_i. If task T_i has not reached its maximum service-level, then its key $cost_i$ is updated and

T_i reinserted into the max-heap. Finally, the total slack $slack_i$ associated with each task gets updated (refer Equation 5.2). This process continues either until all tasks have reached their maximum service-levels, or there exists no task whose total slack is sufficient to effect a service-level enhancement (line nos. 14 to 32). Algorithm 3 depicts a step-wise description of T-SAQA.

Time Complexity of T-SAQA: The complexity of T-SAQA differs from G-SAQA due to differences in the service-level upgradation process. Specifically, the upgradation process takes $O(\sum_{i=1}^{n}|SL_i| \times log(n))$ in G-SAQA whereas T-SAQA takes $O(\sum_{i=1}^{n}|SL_i| \times log(n) \times (n + |\mathcal{E}|))$, since ASAP and ALAP times need to be recomputed for all task nodes in G' after each upgradation (line nos. 14 to 32). Hence, the time complexity of the T-SAQA algorithm is $O(\max\{(\sum_{i=1}^{n}|SL_i| \times log(n) \times (n + |\mathcal{E}|)), (n^2 \times p)\})$.

Example (contd.): As we have discussed earlier, the PEFT schedule (Table 5.2) and assignment PTG G' (Figure 5.3) generation steps for T-SAQA is same as that of G-SAQA. The initial values of $slack_i$ and $cost_i$ corresponding to the seven tasks are as follows: $slack_1 = slack_3 = slack_4 = slack_6 = slack_7 = 2$, $slack_2 = slack_5 = 3$ and $cost_1 = 2, cost_2 = 2.67$, $cost_3 = cost_5 = cost_6 = 1.5$, $cost_4 = 4$, $cost_7 = 1$. Similar to G-SAQA, using these $cost_i$ values as keys, a max-heap is built. The task T_4 with the highest key value (currently, at the root of the max-heap) is extracted from the heap and the service-level is enhanced from sl_{41} to sl_{42}, as the additional computation requirement $\Delta e_4 = 4 - 3 = 1$, can be satisfied by the available slack, $slack_4 = 2$. The finish time of T_4 becomes, $S_4 + e_{422} = 4 + 4 = 8$. Now, with respect to the finish time of T_4, the start time of the successor task T_7 becomes $F_4 + m_{47} = 8 + 1 = 9$. As the newly computed start time of T_7 is larger than the current start time of T_7, S_7 is modified to $S_7 = 9$. Since T_4 has reached its highest service-level, it is not reinserted into the heap. The slack times of all tasks are recomputed using Equation 5.2 ($slack_1 = slack_3 = slack_7 = 1$, $slack_2 = slack_5 = 3$, $slack_6 = 2$).

Next T_2, the task with the highest key, is extracted from the heap and the service-level is upgraded from sl_{21} to sl_{22}. Consequently, F_2 becomes 6 and start times of all successor tasks are updated to, $S_5 = 6, S_6 = 9, S_7 = 10$. As task T_2 has reached its

ALGORITHM 3: T-SAQA

```
Input: A PTG consisting of $n$ tasks, $p$ fully-connected heterogeneous processors
Output: A feasible task schedule that maximizes system level reward
1  forall tasks $T_i$ do
2      Assign minimum service-level $sl_{i1}$ to $T_i$
3  Compute a PEFT schedule to determine start time, finish time and processor
     mapping of each task
4  if PEFT makespan violates the deadline constraint $D$ then
5      Declare "generation of a feasible schedule is not possible"
6      return
7  Construct assignment PTG $G'$ from the original PTG $G$ using PEFT schedule
8  forall mutually independent task pairs ($T_i$, $T_j$) do
9      if $T_i$ and $T_j$ are scheduled on the same processor $P_r$ then
10         Add an edge $T_i \xrightarrow{m_{ij}=0} T_j$ in $G'$, if $T_j$ is scheduled immediately after $T_i$
             on $P_r$ and vice versa
11 forall tasks $T_i$ do
12     Compute $cost_i$ and $slack_i$ using Equations 5.1 and 5.2, respectively
13 Make max-heap of tasks using $cost_i$ as key
14 while max-heap is non-empty do
15     Remove root node $T_i$ to upgrade its service-level
16     Compute additional computation demand: $\Delta e_i \leftarrow e_{i(l+1)r} - e_{ilr}$
17     if $\Delta e_i \leq slack_i$ then
18         Upgrade service-level of $T_i$ from $sl_{il}$ to $sl_{i(l+1)}$
19         Update finish time of $T_i$: $F_i \leftarrow S_i + e_{i(l+1)r}$
20         Create an empty successor task list, SuccList and add $T_i$ to it
21         while SuccList is non-empty do
22             Remove the front task node (say, $T_{i'}$) from SuccList
23             forall children $T_j$ of $T_{i'}$ do
24                 if $S_j < F_{i'} + m_{i'j}$ then
25                     Update start time of $T_j$: $S_j \leftarrow F_{i'} + m_{i'j}$
26                     Update finish time of $T_j$: $F_j \leftarrow S_j + e_{jlr}$
27                     Add $T_j$ to SuccList
28         if current service-level < $|SL_i|$ then
29             Compute $cost_i$ (Equation 5.1) and reinsert $T_i$ into the max-heap
30         Compute ASAP, ALAP of all task nodes
31         forall tasks $T_i$ in max-heap do
32             Compute $slack_i$ using Equation 5.2
```

highest service-level, it is not reinserted into the heap. The slack times of remaining tasks are recomputed and they become, $slack_1 = slack_5 = slack_6 = slack_7 = 0$, $slack_3 = 1$. It may be observed that the available slacks of none of the remaining tasks are not sufficient

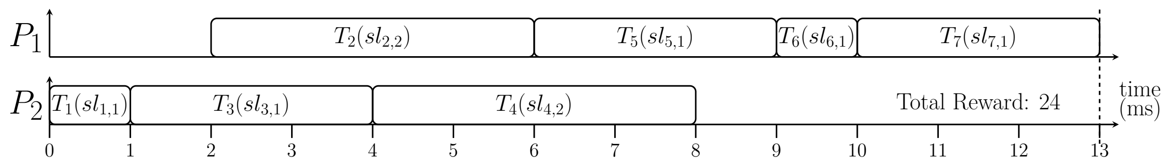


Figure 5.5: *The T-SAQA schedule for G as a gantt chart*

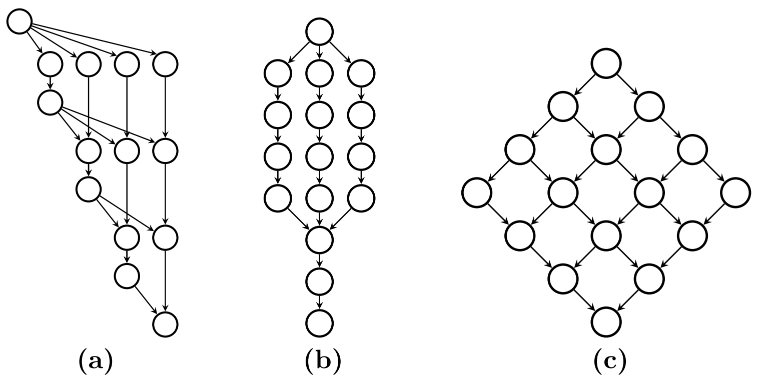


Figure 5.6: *(a) Gaussian Elimination [1], (b) Epigenomics [2] (c) Laplace [4]*

for further service-level enhancements and hence, the algorithm terminates. The gantt chart representation of the final schedule is given in Figure 5.5. The aggregated reward returned by the T-SAQA is 24 which is greater than the reward (18) generated by G-SAQA. □

5.3 Experimental Evaluation

The performance of the proposed strategies G-SAQA and T-SAQA, have been experimentally evaluated w.r.t. to ILP-SANC using real-world benchmark PTGs as well as randomly generated PTGs.

5.3.1 Performance evaluation using benchmark PTGs

Experimental Setup: Three benchmark PTGs namely, *Gaussian Elimination* [60], *Epigenomics* [2], *Laplace* [4] have been employed to experimentally test and compare the algorithms. Figures 5.6a, 5.6b and 5.6c, respectively show the structural representations of *Gaussian Elimination*, *Epigenomics* and *Laplace*.

- The task graph representation of a *Gaussian Elimination* equation solver is determined by the number (χ) of linear equations (also called matrix sizes) which the algorithm attempts to solve. A *Gaussian Elimination* task graph contains $((\chi^2 + \chi - 2)/2)$ nodes when the number of equations to be solved is χ. As example, Figure 5.6a shows a Gaussian Elimination task graph containing 14 ($= (5^2 + 5 - 2)/2$; $\chi = 5$) nodes.

- *Epigenomics* represents a procedure for genome sequencing operations in a parallel pipelined manner. The number of nodes in an *Epigenomics* PTG is given by $(4\gamma + 4)$, where γ denotes the number of parallel branches. As example, for the Epigenomics PTG shown in Figure 5.6b, the number of parallel branches, $\gamma = 3$ and therefore, the graph has 16 task nodes.

- The *Laplace* PTG corresponds to the Laplace algorithm for equation solving with the number of nodes in a PTG being represented by φ^2, where φ is the size of matrix given as an input to the algorithm. As example, for the Laplace PTG shown in Figure 5.6c, the number of parallel branches $\varphi = 4$ and so, the graph has 16 task nodes.

The data associated with the PTGs and also the computing platform, have been varied over carefully chosen ranges of values to evaluate and exhibit the efficacy of the proposed algorithms on various possible scenarios that they may encounter in practice: (1) *Matrix Sizes*: $\chi = \{5, 7, 10, 14\}$ (*Gaussian Elimination*); *Parallel Branches*: $\gamma = \{3, 5, 12, 24\}$ (*Epigenomics*); *Matrix Sizes*: $\varphi = \{4, 5, 7, 10\}$ (*Laplace*). (2) *Number of processors*: Platforms consisting of 2, 4, and 8 processors have been considered. (3) *Communication to Computation Ratio* $CCR = \{0.5, 1, 2\}$ (CCR is the ratio of the average communication to computation cost; i.e., $CCR = \frac{1}{|\mathcal{E}|}\Sigma m_{ij} \Big/ (\frac{1}{n \times p}[\Sigma_{i=1}^{n}\Sigma_{r=1}^{p} e_{i1r}])$). (4) *Number of service-levels* of each task T_i is taken as, $|SL_i| = \{3, 5\}$. (5) *Execution time* e_{i1r} for any task T_i at the base service-level (sl_{i1}) on processor P_r, is chosen randomly from a uniform distribution $[10ms, 50ms]$. The execution time (e_{ilr}) for other service-levels (from sl_{i2}) of T_i are allocated values lying between 110% and 150% of T_i's execution times at the immediately lower service-level ($e_{i(l-1)r}$). (6) The QoS values (QoS_{il}) of T_i are chosen from the range $[1, 200]$ are allocated values such that they are *monotonically* increasing with its service-levels. (7) *Communication times* m_{ij} between the PTG nodes are also chosen from a uniform random distribution $[10ms, 50ms]$. In order to maintain the required CCR, the obtained m_{ij} values are appropriately scaled. (8) The PTG is associated with the single end-to-end *deadline* which is derived from the schedule length obtained by applying the PEFT *list scheduling* heuristic scheme [1]. In particular, we compute two schedule length values D_L and D_H by assigning the PTG task nodes at sl_{i1} and $sl_{i|SL_i|}$, respectively. The actual deadline D is obtained using a parameter called *Deadline extension Rate* (DR): $D = D_L + ((D_H - D_L) \times DR)$, where $DR \in \{0, 0.25, 0.5, 0.75, 1\}$. For example, the different deadlines values corresponding to various DRs for a PTG with $D_L = 20$ and $D_H = 40$ are 20, 25, 30, 35, 40. System configuration used for experimentation: (i) software tool: CPLEX optimizer [10] version 12.6.2.0, (ii) operating system: Linux Kernel 3.10.0-693.21.1.el7.x8, and (iii) CPU: Intel(R) Xeon(R).

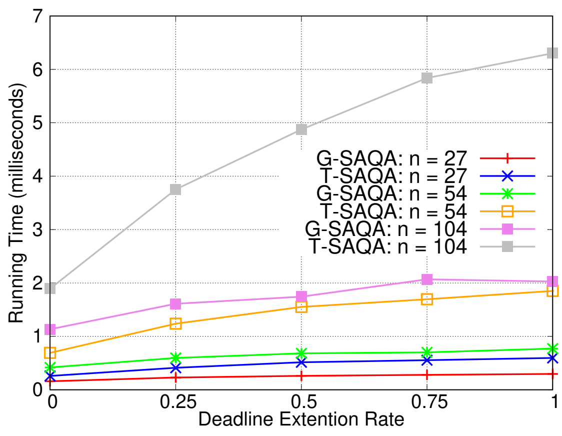
Performance Metrics: (1) *Normalized Reward*: NR (in %) $= \frac{R_{ACT}}{R_{MAX}} \times 100$, where, R_{ACT} denote the QoS obtained and R_{MAX} represents the maximum QoS, for the PTG. (2) *Running time*: Total time taken to compute the solution for a given PTG. Each data point in the plots for scheduler run-times is obtained as the average over 100 runs of the scheduler, on different PTG instances produced by carefully varying a chosen set of parameters.

Experiment-1: Comparing Running Time: The first part of this experiment compares the running times of ILP-SANC, G-SAQA, and T-SAQA. The number of processors p is fixed at 8, CCR at 1, number of service-levels $|SL_i|$ at 3, χ at 5 (Gaussian Elimination), γ at 3 (Epigenomics) and φ at 4 (Laplace). Table 5.3 shows the results. It can be seen that *both the heuristic schemes (G-SAQA and T-SAQA) are about* $\sim 10^6$ *times faster on an average than the optimal strategy ILP-SANC.* The second part of this experiment compares the running times of G-SAQA and T-SAQA by varying the number of task nodes in the PTG, while fixing p at 4, CCR at 1, $|SL_i|$ at 5. The experimental results are depicted in Figure 5.7. The results show that the running-time of T-SAQA is always greater than G-SAQA. This is because ASAP and ALAP times need to be recomputed for all tasks in the PTG after each service-level upgradation in T-SAQA (refer time-complexity analysis of T-SAQA). As example, in the Epigenomics PTG with $DR = 0.25$ (refer Figure 5.7b), the running time for $n = 100$, using G-SAQA and T-SAQA are ~1.2 ms and ~3.2 ms, respectively.

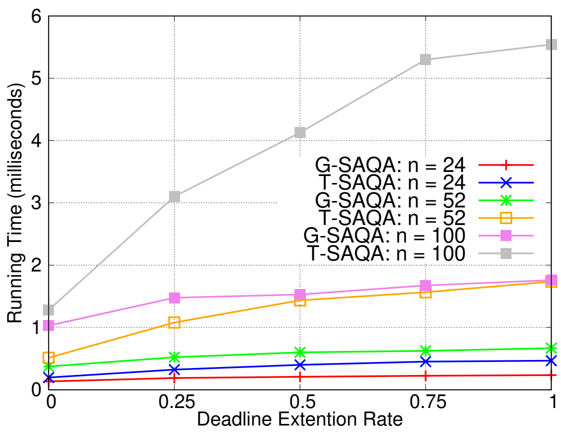
PTG	Strategy	Deadline Extension Rate				
		0	0.25	0.5	0.75	1
Gaussian Elimination	ILP-SANC	477.288s	216.231s	60.2835s	16.9734s	3.7769s
	G-SAQA	95.78μs	98.53μs	100.79μs	109.45μs	113.15μs
	T-SAQA	117.87μs	130.67μs	138.49μs	145.92μs	154.8μs
Epigenomics	ILP-SANC	3213.06s	1509.76s	206.503s	33.414s	4.1416s
	G-SAQA	95μs	102.23μs	102.15μs	110.31μs	117.56μs
	T-SAQA	118.55μs	135.08μs	143.15μs	156.6μs	159.18μs
Laplace	ILP-SANC	11510.4s	1931.33s	233.483s	32.2053s	9.8382s
	G-SAQA	112.74μs	122.17μs	121.25μs	130.6μs	139.51μs
	T-SAQA	139.27μs	157.6μs	168.76μs	182.08μs	189.86μs

Table 5.3: *Comparing run-times of ILP-SANC, G-SAQA and T-SAQA*

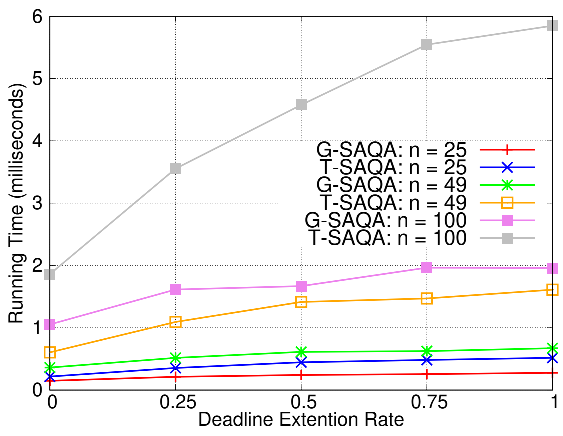
Experiment-2: Variation on the number of processors: This experiment compares ILP-SANC, G-SAQA, and T-SAQA, as the number of processors p is increased from 2 to 8, while fixing CCR to 1, number of service-levels $|SL_i|$ to 3 for all tasks. Figure 5.8 shows the results. It can be observed that the normalized reward NR *improves with increase in the number of processors, for any given deadline extension rate value.* This may be attributed to the fact that as number of processors become higher, residual capacity increases. This residual capacity has been used by the system to enhance the tasks' service-levels which in turn result in higher NR. Additionally, it may be noted that ILP-SANC being optimal, always outperforms its heuristic counter-parts G-SAQA and T-SAQA. Among the heuristic algorithms, T-SAQA which better harnesses the slack available in the base PEFT schedule, it seems to always deliver better performance



(a) *Gaussian Elimination*



(b) *Epigenomics*



(c) *Laplace*

Figure 5.7: *Running time comparison of G-SAQA and T-SAQA*

than G-SAQA. For example, let us consider the Epigenomics PTG with $DR = 0.25$ (Figure 5.8b). NR values obtained using ILP-SANC for $p = 2$ and $p = 8$ are ∼87% and ∼100%, respectively. Also, for a fixed number of processors (say, $p = 2$), ILP-SANC is seen to deliver ∼29% and ∼17% better results compared to G-SAQA and T-SAQA, respectively.

The relative degradation of the proposed schemes have been measured by using a parameter called *Relative QoS* ($\mathcal{R}$), which is defined as $\mathcal{R}(\%) = \frac{R_{optimal} - R_{heuristic}}{R_{optimal}}$. Table 5.4 depicts the distribution of relative QoS corresponding to the two proposed heuristic schemes using three different benchmark PTGs on systems consisting of 2, 4, and 8

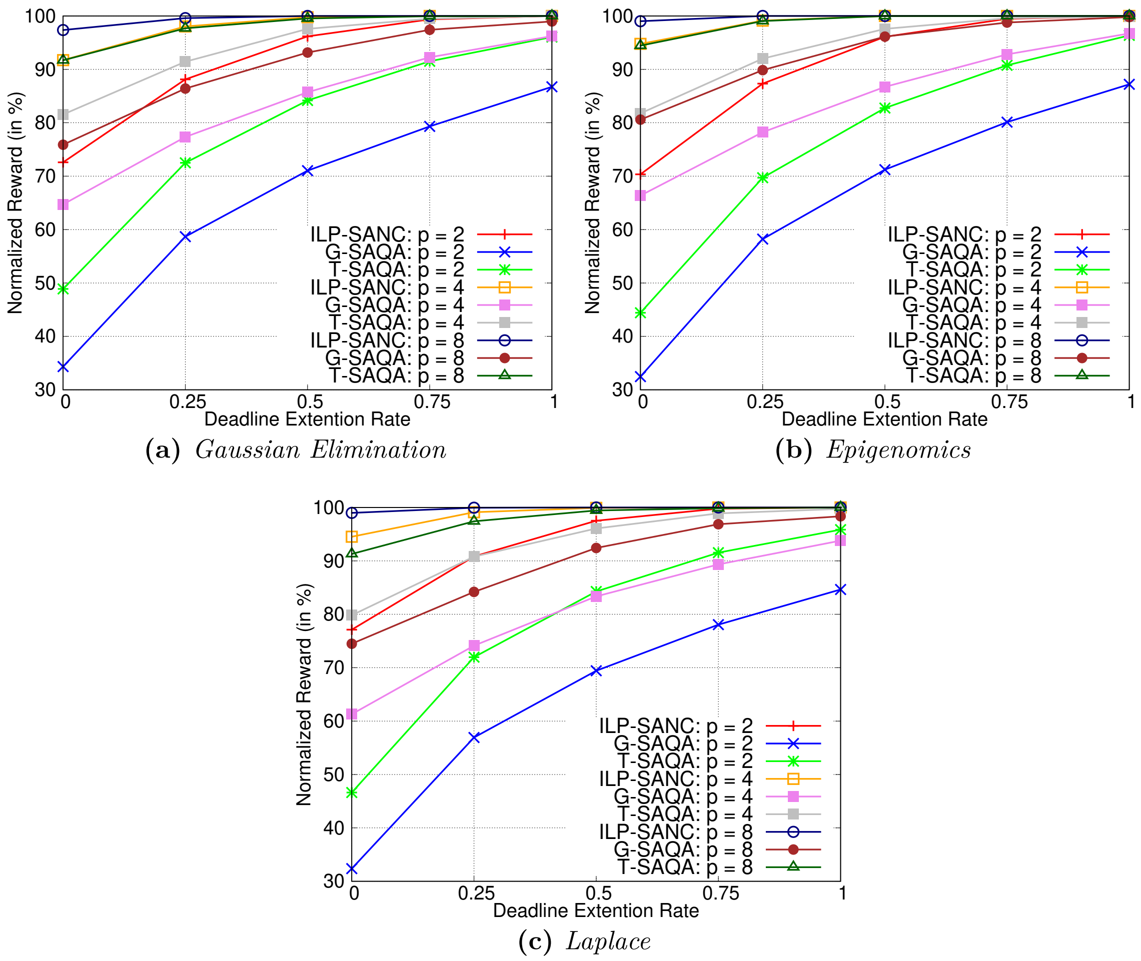


Figure 5.8: *Effect of varying processors*

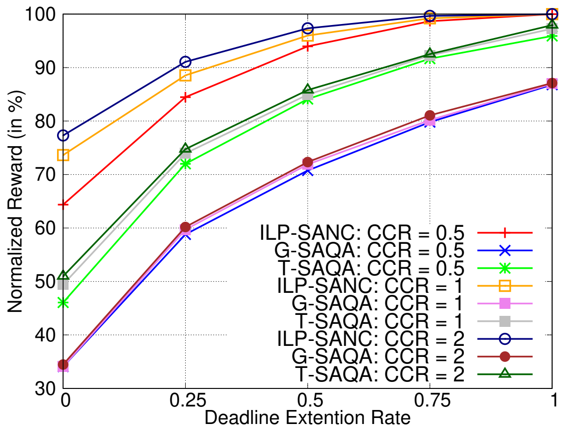
processors. For any fixed combination of these parameters, 100 test cases have been executed. For a specific benchmark and a particular number of processors, we have shown the number of test cases for which deviation $\mathcal{R}$ in performance of the heuristic algorithm is within 10%, between 10% and 20%, and above 20%, respectively. For example, when the selected PTG is *Gaussian Elimination* and $p = 2$, the deviation in performance is within 10% for 42 test cases, between 10% and 20% for 42 test cases, above 20% for 16 test cases, within the total 100 test cases considered. Further, we found that T-SAQA is able to produce same results as the optimal strategy ILP-SANC, in many cases. For example, T-SAQA is able to produce the same reward as the ILP-SANC in 36 test cases out of the total of 100 test cases considered for *Gaussian Elimination* with $DR = 0.5$, $CCR = 1$ and $p = 4$.

PTG	Strategy	#Processors (with DR fixed at 0.5)								
		$p = 2$			$p = 4$			$p = 8$		
		$\mathcal{R} \leq 10$	$10 < \mathcal{R} \leq 20$	$\mathcal{R} > 20$	$\mathcal{R} \leq 10$	$10 < \mathcal{R} \leq 20$	$\mathcal{R} > 20$	$\mathcal{R} \leq 10$	$10 < \mathcal{R} \leq 20$	$\mathcal{R} > 20$
Gaussian Elimination	G-SAQA	0	22	78	28	52	20	76	18	6
	T-SAQA	42	42	16	96	4	0	100	0	0
Epigenomics	G-SAQA	0	18	82	32	56	12	90	10	0
	T-SAQA	30	56	14	96	4	0	100	0	0
Laplace	G-SAQA	0	10	90	28	32	40	76	22	2
	T-SAQA	40	38	22	82	18	0	100	0	0

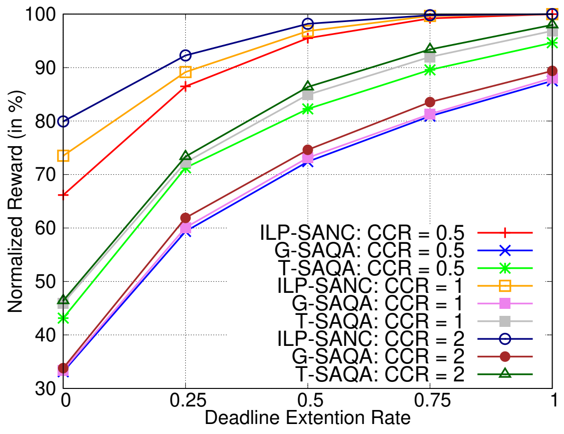
Table 5.4: *Distribution for deviation in performance (w.r.t. ILP-SANC) G-SAQA and T-SAQA*

Experiment-3: Varying CCR: We vary CCR from 0.5 to 2, while fixing p to 2 and number of service-levels $|SL_i|$ to 3 for all tasks. Figure 5.9 depicts the results for this experiment. For fixed values of p, n and D, higher values of CCR imply lower computation demands of tasks on processors at any given service-level. Such lower computation demand naturally enhances the possibility of task service-level upgradation. Consequently, this leads to an increase in the obtained QoS (NR). Additionally, higher values of CCR imply higher communication demands imparted by message nodes. The removal of such heavy message nodes (when both the predecessor and successor task nodes of the message node are assigned onto the same processor) in turn results in higher time savings which can also be used to enhance the service-levels of tasks to produce higher QoS (NR). For example, in the Epigenomics PTG with $DR = 0.25$ (Figure 5.9b), the normalized reward NR obtained using ILP-SANC for CCR = 0.5 and CCR = 2 are ∼86% and ∼ 92%, respectively. Further, for the same set of system parameters, it can be seen that ILP-SANC delivers ∼27% and ∼15% better results compared to G-SAQA and T-SAQA, respectively. Among the heuristic schemes, T-SAQA performs better than

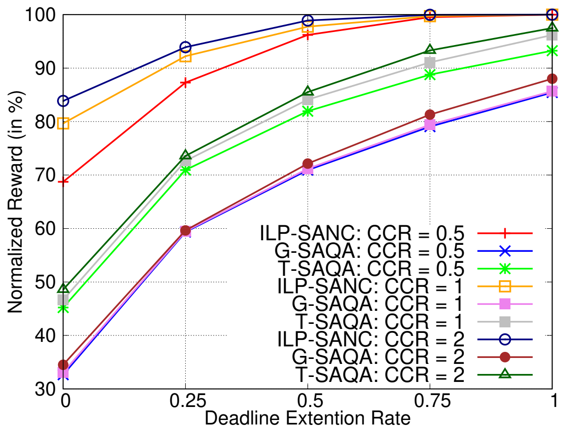
G-SAQA in all cases, as expected.



(a) *Gaussian Elimination*



(b) *Epigenomics*

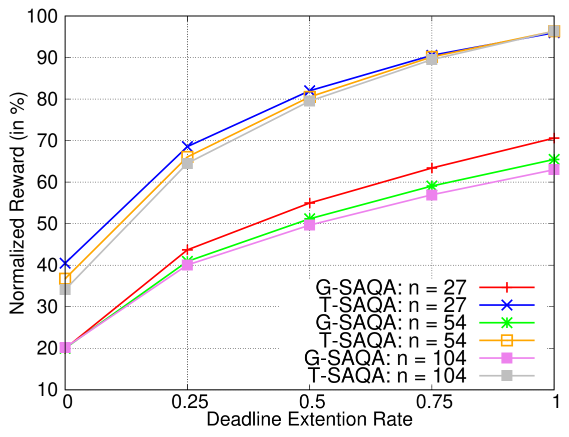


(c) *Laplace*

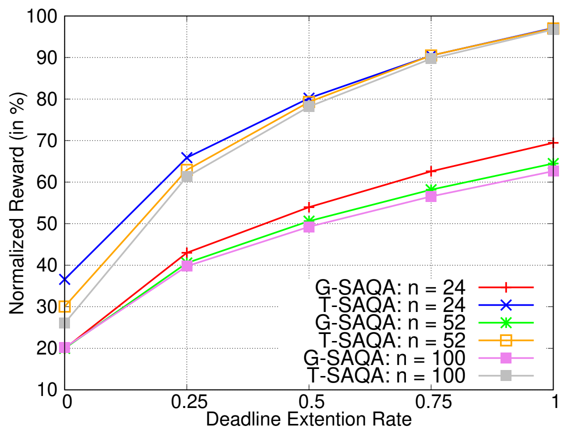
Figure 5.9: *Effect of varying CCR*

Experiment-4: Varying the number of tasks: In this experiment, we compare the performance of G-SAQA and T-SAQA by varying the number of task nodes in the PTGs as follows: Gaussian Elimination: $\{27, 54, 104\}$; Epigenomics: $\{24, 52, 100\}$; Laplace: $\{25, 49, 100\}$. Also, we set p to 4, CCR to 1, number of service-levels $|SL_i|$ to 5. Figure 5.10 depicts the results for this experiment. It can be seen from the figures that *the NR values decrease with increase in the number of tasks.* As the number of tasks increase with the deadline remaining fixed, the total available slack decreases, which in turn leads to lower rewards. For example, in the Epigenomics PTG with $DR = 0.5$

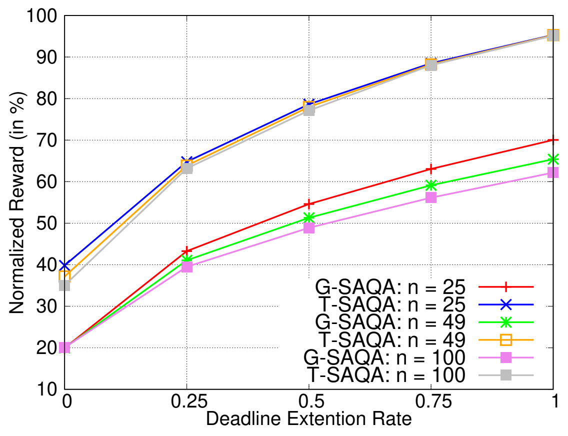
(Figure 5.10b), the normalized reward obtained using G-SAQA for $n = 24$ and $n = 100$ are ∼54% and ∼49%, respectively. It can also be seen from the figures that in all cases, T-SAQA returns higher NR values than G-SAQA due to its ability to better utilize the total slack available in the system. For example, in the Epigenomics PTG with $DR = 0.25$ (Figure 5.10b), the normalized reward obtained for $n = 24$ and $n = 100$ using G-SAQA are ∼43% and ∼40%, respectively, and for T-SAQA are ∼66% and ∼61%, respectively.



(a) *Gaussian Elimination*



(b) *Epigenomics*



(c) *Laplace*

Figure 5.10: *Effect of varying number of tasks*

Experiment-5: Comparison with the state-of-the-art: This experiment compares ILP-SANC, G-SAQA and T-SAQA against the *list based* heuristic scheduling algorithm PEFT [1]. The essential objective of PEFT is to minimize makespan corresponding to a task graph in which all tasks have only a single service-level. Therefore, as PEFT is task service-level oblivious, in order to apply PEFT within our framework, service-levels of all tasks must be fixed before its application. After assigning selected service-levels to task nodes, PEFT is run on the PTG and the resulting normalized reward NR and makespan values are noted. ILP-SANC, G-SAQA and T-SAQA are then executed on the same PTG with the makespan value delivered by PEFT, as deadline. To improve normalized reward values for PEFT, we have selectively chosen higher service-levels for those tasks which deliver higher reward gains with respect to additional execution time consumed. The experimental results are depicted in Table 5.5. It may be observed that like optimal solution approach ILP-SANC, our proposed heuristic approaches G-SAQA and T-SAQA are also able to achieve higher normalized rewards compared to PEFT for any given deadline bound, unless the deadline is so relaxed that PEFT is also able to assign highest service-levels to all tasks.

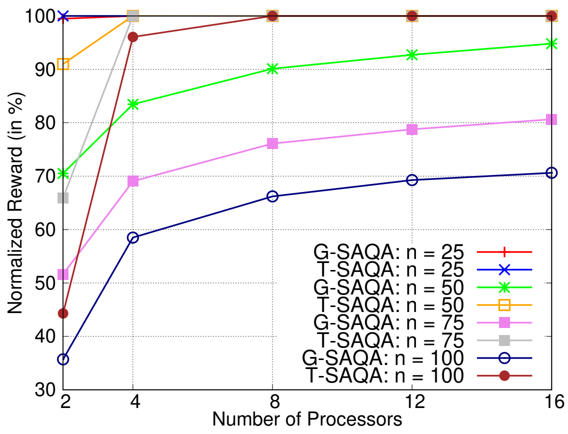
PTG	Strategy	Percentage of Tasks Upgraded				
		0	0.25	0.5	0.75	1
Gaussian Elimination	ILP-SANC	75.11	77.11	91.12	99.13	100
	G-SAQA	35.68	54.09	78.69	88.79	100
	T-SAQA	52.57	69.75	86.32	96.76	100
	PEFT	30.08	49.39	71.37	83.84	100
Epigenomics	ILP-SANC	88.63	89.60	98.94	100	100
	G-SAQA	47.89	75.31	87.58	96.23	100
	T-SAQA	58.87	82.35	96.73	98.97	100
	PEFT	45.95	67.36	81.18	94.91	100
Laplace	ILP-SANC	81.23	86.42	94.83	99.63	100
	G-SAQA	44.65	72.29	81.74	94.73	100
	T-SAQA	55.97	80.65	95.71	97.98	100
	PEFT	35.45	61.47	79.13	90.17	100

Table 5.5: *Comparison of ILP-SANC, G-SAQA and T-SAQA with PEFT [1]*

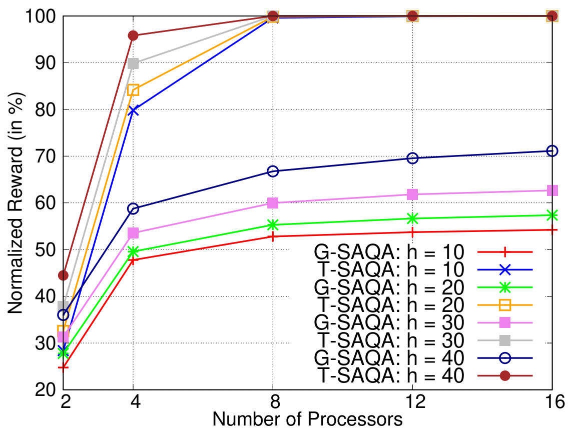
5.3.2 Performance evaluation using randomly generated PTGs

Experimental Setup: In this section, we have considered randomly generated PTGs in place of benchmark PTGs, to evaluate the performance of our proposed schemes. For this purpose, we have used the following parameters: (1) the number of tasks (n) in the PTG: varied from 25 to 100, (2) the total number of levels in the PTG (denoted by $\mathcal{L}$): generated from a normal distribution having $\langle \mu, \sigma \rangle$ values equal to $\left\langle \sqrt{n}, \sqrt{n} * 0.1 \right\rangle$, (3) the number of nodes at any level in the PTG: generated from another normal distribution having $\langle \mu, \sigma \rangle$ values being $\langle n/\mathcal{L}, (n/\mathcal{L}) * 0.1 \rangle$, (4) the number of processors (p): varied from 2 to 16, (5) heterogeneity (h): varied from 10 to 40. For a given value of h, the execution times of a task on different processors have been generated by selecting values from a uniform distribution $[(30 - h/2), (30 + h/2)]$. It may be observed that heterogeneity in the execution times of a task on different processors monotonically increase with h. All result points on any given line in the plots in Figures 5.11a and 5.11b are generated by running a specific algorithm on the same task graph and the same value of deadline. To compute this deadline, the PEFT algorithm is used to schedule the task graph and obtain its *makespan* for the case when $p = 2$, $h = 10$, and $n = 100$. The deadline is then given by the minimum number D, which is a multiple of 100 and is greater than or equal to the obtained makespan. For example, let the average makespan produced by PEFT after 100 iterations to be 1556. So, the deadline D for this case becomes 1600.

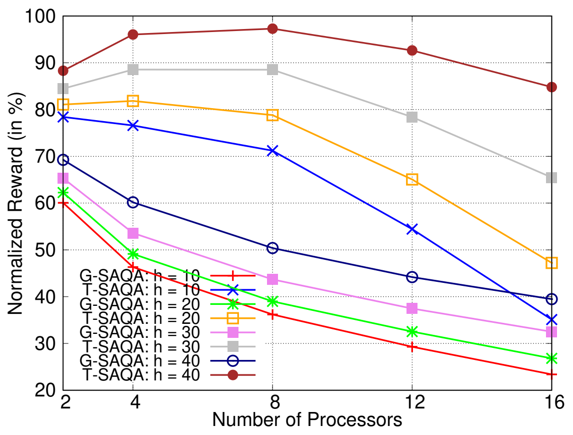
Experiment-5: Varying the number of tasks and processors: In this experiment, we compare the performance of G-SAQA and T-SAQA by varying the number of task nodes in the PTG from 25 to 100 and number of processors from 2 to 16. Also, we set CCR to 1, the number of service-levels $|SL_i|$ to 5 for all tasks. Figure 5.11a depicts the results for this experiment. It may be observed that the obtained trends for randomly generated PTGs are very similar to those obtained for benchmark PTGs. Even for this case, NR improves as #processors become higher because residual capacity increases. For example, when $n = 100$, the normalized reward obtained using T-SAQA for $p = 4$ and $p = 8$ are ~96% and ~100%, respectively. *The NR values may be seen to*



(a) *Varying n and p*



(b) *Varying h and p*



(c) *Varying h and p keeping $\mathcal{W}$ same*

Figure 5.11: *Effect of varying #tasks, #processors and heterogeneity*

decrease with the increase in the number of tasks. As the number of tasks increase with the deadline remaining fixed, the total available slack decreases, which in turn leads to lower rewards. For example, at $p = 4$ the NR value obtained using G-SAQA for $n = 25$ and $n = 100$ are ∼100% and ∼59%, respectively. Similar to the previous experiments, T-SAQA returns higher or equal NR values than G-SAQA, for all cases.

Experiment-6: Varying heterogeneity: This experiment compares the performance of G-SAQA and T-SAQA by varying the heterogeneity (h) among processors from 10 to 40 and number of processors from 2 to 16. Also, we set n to 100, CCR to 1, number of service-levels $|SL_i|$ to 5. Figure 5.11b depicts the results for this experiment. It can be observed that the normalized reward NR *increases with increase in the heterogeneity among processors.* With the increase in heterogeneity, difference in the execution times of a task on different processors, also increase. This shows that both G-SAQA and T-SAQA can effectively harness task-to-processor affinities resulting from the underlying system heterogeneity and conduct efficient resource allocation so that the overall aggregate reward is maximized. For example, the NR values obtained using T-SAQA for $h = 10$ and $h = 40$ are ∼80% and ∼96%, respectively, at $p = 4$. It can also be seen that in all cases, T-SAQA outperforms G-SAQA.

Experiment-7: Varying the number of processors while keeping normalized workload same: In this experiment, we have maintained the *normalized workload* same instead of *absolute workload.* Here, we have defined *normalized workload* $\mathcal{W}$ of the task nodes in a PTG as,

$$\mathcal{W} = \frac{\sum_{i=1}^{n} \sum_{l=1}^{|SL_i|} \sum_{r=1}^{p} e_{ilr}}{D \times p^2 \times \sum_{i=1}^{n} |SL_i|} \tag{5.3}$$

where, D is the deadline, p is the number of processors, $|SL_i|$ is the number of service-levels of task T_i and e_{ilr} is the execution time of T_i at service-level sl_{il} on processor P_r.

The results of the experiment are reported in Figure 5.11c. It can be seen from the figure that the normalized rewards (NR) decrease with increase in the number of processors. This is because in order to maintain normalized workload at a particular value, the average execution time of each task must be increased p''/p' times, when the number of processors is increased from p' to p'' (say). When the execution time of a task is relatively higher, the difference in execution times of the task between consecutive service-levels will also be higher. On the other hand, although the total system capacity increases with the increase in the number of processors, the capacity of individual processors remain same. The non-preemptive task nodes which are now bigger in terms of their execution times must still be entirely executed within the capacity of a single processor. From these observations, it may be easily inferred that increase in the number of processors (with normalized workload remaining same) leads to the possibility of, (i) decrease in the number of tasks which may be feasibly accommodated within a single processor (leading to decrease in resource utilization), (ii) reduction in the degree of service-level enhancement of tasks (leading to decrease in rewards). In addition, it may be observed that the structure of the PTG (number of task nodes and their inter dependencies) used as input to the experiment continues to remain same as the number of processors is increased. With the structure remaining same, the inherent available parallelism associated with the input PTG do not change when the available number of processors increase. Thus, when the total number of processors is sufficiently high, the total resource utilization will ultimately start to exhibit a decreasing trend.

5.4 Case Study: Traction Controller

To exhibit the practical applicability of the proposed strategies to real-world design, a case study using a *Traction Controller* (TC) application present in automotive systems, is discussed here. TC helps in actively stabilizing an automobile so that it can continue in its stipulated path even when road conditions are slippery [5]. Figure 5.12a depicts the block diagram of TC as adopted from [5]. The corresponding PTG representation which consists of 10 task nodes $\{T_1, T_2, \ldots, T_{10}\}$, is shown in Figure 5.12b. For the purpose of this case study, we assume that the PTG is to be executed on a two processor heterogeneous distributed platform, $P = \{P_1, P_2\}$. Table 5.6 lists the execution times of the task nodes on the different processors in platform P. The deadline is assumed to be 134 *ms*.

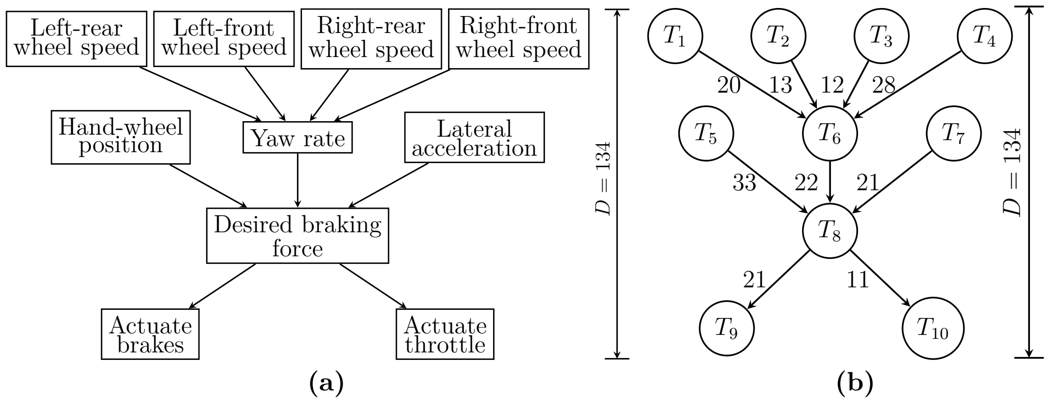


Figure 5.12: *Traction Control application's (a) Block Diagram [5], (b) PTG representation*

		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
sl_{i1}	P_1	14	15	14	27	20	22	25	16	26	18
	P_2	26	17	11	29	10	17	28	24	25	14
	QoS_{i1}	41	112	97	5	137	23	48	64	13	37
sl_{i2}	P_1	17	19	17	32	25	25	32	20	29	23
	P_2	32	21	13	35	12	19	35	30	28	18
	QoS_{i2}	53	163	104	88	143	81	64	80	95	113
sl_{i3}	P_1	19	22	18	36	28	30	35	22	34	26
	P_2	37	24	14	40	13	22	38	34	33	21
	QoS_{i3}	143	164	190	190	171	99	149	157	135	135

Table 5.6: *Computation time (in ms) of task nodes in Traction Control PTG*

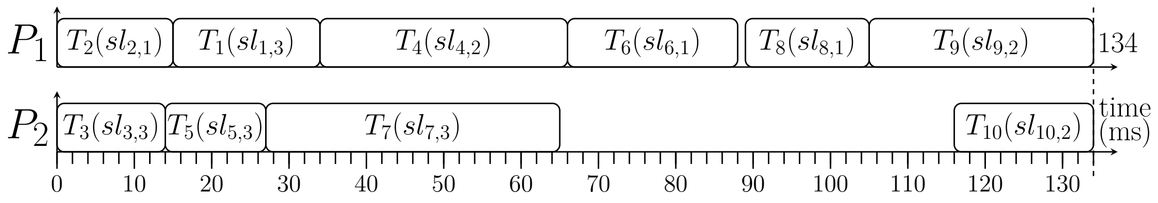


Figure 5.13: *The ILP-SANC schedule for G (Figure 5.12b) as a gantt chart; Reward* = 1148

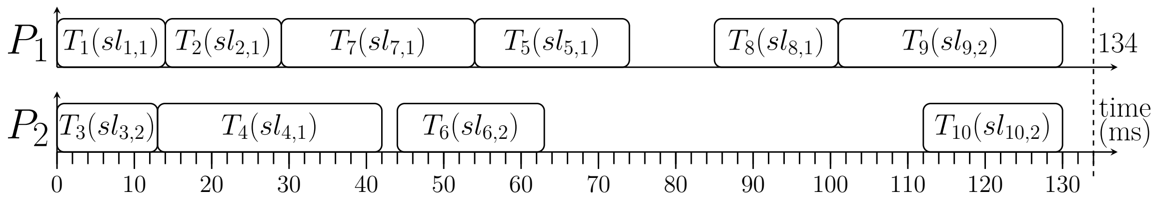


Figure 5.14: *The G-SAQA schedule for G (Figure 5.12b) as a gantt chart; Reward* = 800

We have employed ILP-SANC, G-SAQA and T-SAQA to generate schedules for the PTG of TC with the objective of maximizing aggregate reward. We summarize below the important observations associated with the obtained schedules for ILP-SANC (Figure 5.13), G-SAQA (Figure 5.14), and T-SAQA (Figure 5.15):

- *Heterogeneity Modeling*: Each PTG node consumes distinct execution times depending on the processor allocated to it. As example, in Figure 5.13, T_3 consumes

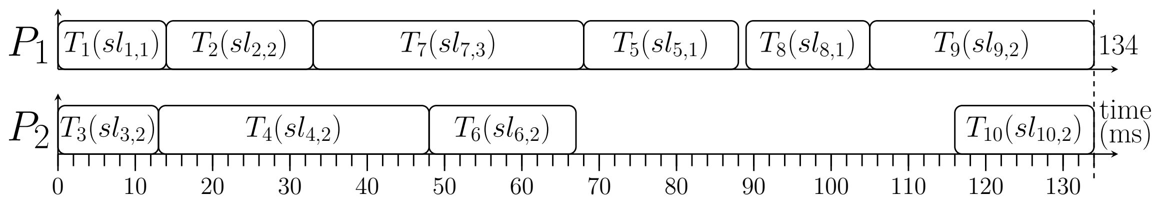


Figure 5.15: *The T-SAQA schedule for G (Figure 5.12b) as a gantt chart; Reward* = 1035

14 *ms* as it is scheduled on P_2 at service-level 3.

- Implementation of ILP-SANC using the CPLEX optimizer generates 409 constraints and takes $\sim$ 3.07 secs to produce the solution. The corresponding schedule (in Figure 5.13) delivers a reward of 1148 ($NR = 74.89$). On the other hand, the schedules computed using G-SAQA (Figure 5.14) and T-SAQA (Figure 5.15) take 223 μs and 259 μs, respectively to generate their solutions while producing rewards of 800 ($NR = 52.19$) and 1035 ($NR = 67.51$), respectively. Similar to the results trend obtained in the experiments section (Section 5.3), we observe that the optimal solution ILP-SANC delivers significantly higher rewards (348 higher w.r.t. G-SAQA; 113 higher w.r.t. T-SAQA). However, the run-time overhead associated with ILP-SANC is about $\sim 10^6$ times higher than the heuristic algorithms. On the other hand, the heuristic algorithms are much faster than ILP-SANC while delivering acceptably good solutions as necessary during quick design iteration. Finally, T-SAQA may be seen to produce better solution although at the cost of a slightly higher run-time overhead.

5.5 Summary

This chapter considers the problem of computing heuristic schedules for PTGs with multiple service levels, executing on fully-connected distributed systems consisting of heterogeneous processors. In the previous chapter (Chapter 4), two distinct ILP based optimal solution approaches ILP-SATC and ILP-SANC are proposed to solve the same problem. Though ILP-SANC significantly improves scalability of the solution compared to ILP-SATC, its run time is still high and sensitive to the number of tasks. Appreciating the necessity of a fast but efficient algorithm for the problem at hand, especially for situation when quick solutions are needed at design-time or run-time, we have proposed two heuristics namely, G-SAQA and T-SAQA. Extensive experiments have been carried-out using benchmark and randomly generated PTGs to evaluate performance of the proposed strategies (G-SAQA, T-SAQA). The obtained results show that *both the heuristic schemes (G-SAQA and T-SAQA) are* $\sim 10^6$ *times faster than the optimal strategy ILP-SANC.* Further, the solution qualities delivered by T-SAQA is at least as good as G-SAQA. However, the computational overheads associated with T-SAQA is significantly higher than G-SAQA. Finally, a case study on a *Traction Controller* (TC) application has been presented. The next chapter extends the problem of scheduling PTGs on fully-connected platforms to CPSs where the processors are connected through a limited number of bus based shared communication channels.

Chapter 6

PTG Scheduling on Shared-Bus Based Heterogeneous Platforms

The PTG scheduling techniques considered in the previous chapters (Chapter 4 and Chapter 5) assumed a fully connected heterogeneous platform. Assumption of a fully connected platform helps to avoid the problem of communication resource contention, as is the case when the system is assumed to have shared data transmission channels. It may be appreciated that shared bus networks form a very commonly used communication architecture in CPSs [41, 42]. Therefore, this chapter proposes the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for the co-scheduling of real-time PTGs executing on a distributed platform consisting of a set of heterogeneous processing elements interconnected by heterogeneous shared buses. Although, both the approaches produce optimal solutions, ILP-NC suffers significantly lower computational overheads compared to ILP-ETR. In addition to the optimal solution approaches, we propose a fast but effective heuristic strategy called CC-TMS which consumes much lower time and space complexities while producing satisfactorily good solutions. The proposed schemes have been evaluated through an extensive set of experiments and a case study with a *Traction Controller* (TC) application is presented.

6.1 System Model

The system model associated with this work is presented by describing the platform, computation model and assumptions.

Platform: Figure 6.1 shows the pictorial representation of the *heterogeneous multiprocessor platform* ρ as considered here. The platform is composed of a resource set $\{R_1, R_2, \ldots, R_{p+b}\}$ among which, $\{R_1, R_2, \ldots, R_p\}$ denote a set $P = \{P_1, P_2, \ldots, P_p\}$ of p heterogeneous processors; whereas, resources $\{R_{p+1}, R_{p+2}, \ldots, R_{p+b}\}$ represent a set $B = \{B_1, B_2, \ldots, B_b\}$ of b heterogeneous shared buses. Each bus $B_r \in B$ is connected to all processors in P.

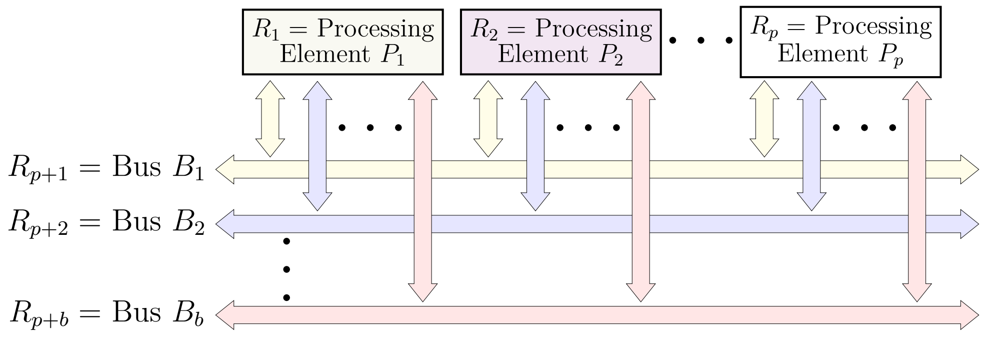

R_1 = Processing Element P_1
R_2 = Processing Element P_2
R_p = Processing Element P_p
R_{p+1} = Bus B_1
R_{p+2} = Bus B_2
R_{p+b} = Bus B_b

Figure 6.1: *Platform Model* ρ

Computation Model: A *Cyber-Physical System* (CPS) application as considered in this work is represented by a PTG $G = (\mathcal{V}, \mathcal{E}, ET, CT)$ where,

- $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n+m}\}$ represents the node. Among them, $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n\}$ represent a set $T = \{T_1, T_2, \ldots, T_n\}$ of n tasks. Similarly, $\{\mathcal{V}_{n+1}, \mathcal{V}_{n+2}, \ldots, \mathcal{V}_{n+m}\}$ denotes a set $M = \{M_1, M_2, \ldots, M_m\}$ of m messages which specify communication demand between task pairs.

- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denotes the edge set describing *dependency-constraints* between pairs of nodes in $\mathcal{V}$.

- ET is an *execution-time matrix* of size $n \times p$. Each element $e_{ir} \in ET$ captures the *execution demand* of task T_i (vertex $\mathcal{V}_i$) on processor P_r (resource R_r).

- CT is a *communication-time matrix* of size $m \times b$. Each element $c_{kr} \in CT$ captures the *communication time* associated with message M_k (vertex $\mathcal{V}_{n+k}$) on bus B_r (resource R_{p+r}).

The Assumptions:

1. PTG G has a single entry (source) node T_1 (having in-degree 0) and a single exit (sink) node T_n (having out-degree 0). If the input PTG contains multiple source/sink nodes, we add a single dummy source/sink task node (having execution time 0 on all processors) which connect to all original source/sink task nodes via dummy edges. Dummy message nodes (having transmission time 0 for all buses) are added to each of these new edges.

2. The entry (T_1) and exit (T_n) nodes are both tasks.

3. Any task T_i is preceded (except T_1)/succeeded (except T_n) by message node(s).

4. Any message M_k is preceded/succeeded by one task node only.

5. The communication overhead associated with a message node M_k is assumed to be negligible ($\forall B_r \in B$, $c_{kr} = 0$) when its preceding as well as succeeding task nodes are assigned to the same processor.

Example: Figure 6.2a shows an example of a PTG G which consists of 13 nodes $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{13}\}$. Among them, $\{\mathcal{V}_1, \ldots, \mathcal{V}_6\}$ are tasks and $\{\mathcal{V}_7, \ldots, \mathcal{V}_{13}\}$ are messages. Hence, $n = 6$ and $m = 7$. Figure 6.2b shows a sample platform model ρ consisting of four resources $R = \{R_1, R_2, R_3, R_4\}$. Among them R_1, R_2 denote two processors P_1, P_2 and R_3, R_4 denote two buses B_1, B_2. Thus, $p = 2$, $b = 2$. In Table 6.1, we show the execution time matrix ET. An element say $e_{1,1} = 4$ in this matrix specifies that task T_1 (corresponding to vertex $\mathcal{V}_1$) takes 4 units of time to finish execution on processor P_1. In the same way, Table 6.2 shows matrix CT, depicting communication time. An element say $c_{1,1} = 2$ in this matrix specifies that message M_1 (corresponding to vertex $\mathcal{V}_7$) takes 2 units of time for transmission over bus B_1 (assigned on resource R_3).

Problem Definition: Given a real-time PTG $G = (\mathcal{V}, \mathcal{E}, ET, CT)$ with end-to-end deadline D to be executed on a heterogeneous platform consisting of p processors and b buses, determine the start times for all task and message nodes, task-to-processor and

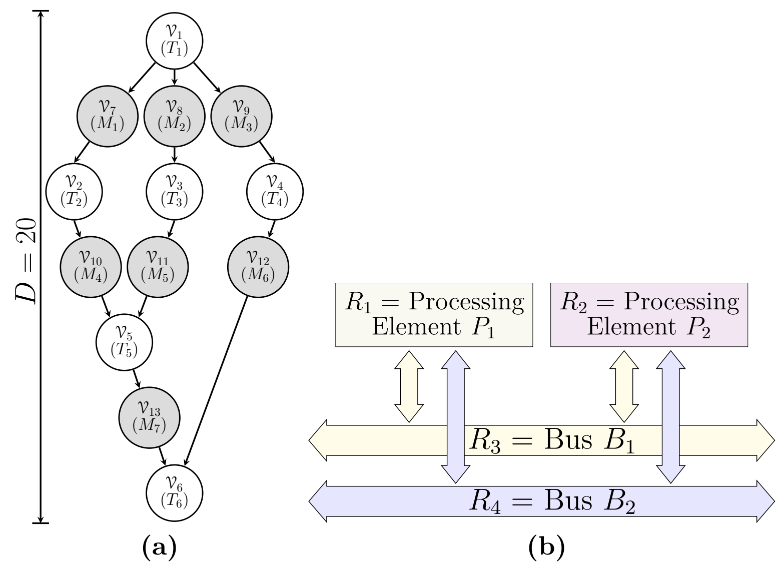


Figure 6.2: *(a) Example of a PTG G and (b) Model of Platform ρ*

	T_1	T_2	T_3	T_4	T_5	T_6
P_1	4	8	3	2	4	2
P_2	3	5	4	3	2	3

Table 6.1: *Execution time Matrix ET of task nodes*

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
B_1	2	4	5	3	3	1	3
B_2	3	3	3	4	2	3	2

Table 6.2: *Communication time Matrix CT of message nodes*

message-to-bus assignments, with the objective of minimizing the *overall makespan* (i.e., schedule length) and meets the deadline D.

6.2 Earliest/Latest Start Times for PTG Nodes

Unlike the discussions on the tasks' Earliest/Latest start times in Section 4.2 and Subsection 5.2.1 in Chapters 4 and 5, we present here the Earliest and Latest computation times for messages in addition to tasks.

Let, t_i^s and t_i^l be the *earliest* and *latest* time steps at which node $\mathcal{V}_i$ may start its execution. These upper and lower bounds on start times are determined separately for the task and message nodes in the given PTG. The t_i^s (ASAP time) and t_i^l (ALAP time) values for task nodes are computed as follows.

ASAP/ALAP computation procedure for task nodes:

1. We ignore message nodes in the PTG and assume directed edges between the predecessor and successor task nodes of each message node.

2. Set ASAP time of the source task node as: $t_1^s = 1$.

3. Set ALAP time for the sink task node as,
$$t_n^l = D - \min_{r \in [1,p]} e_{nr} + 1$$

4. ASAP times for the remaining task nodes (except T_1) are recursively determined (downward) as follows:
$$t_i^s = \max_{T_j \in pred(T_i)} (t_j^s + \min_{r \in [1,p]} e_{jr})$$
where, $pred(T_i)$ is the set of predecessors of task node T_i.

5. ALAP times for the remaining task nodes (except T_n) are recursively defined (upward) as follows:
$$t_i^l = \min_{T_j \in succ(T_i)} (t_j^l - \min_{r \in [1,p]} e_{ir})$$
where, $succ(T_i)$ is the set of successors of task node T_i.

Given the ASAP/ALAP times for task nodes in the PTG, we now compute these values for message nodes.

ASAP/ALAP computation procedure for message nodes:

1. The ASAP time of a message node M_k is defined as:

$$t^s_{n+k} = t^s_i + \min_{r\in[1,p]} e_{ir}$$

   where, T_i (i.e., vertex $\mathcal{V}_i$) is the predecessor task node of M_k (i.e., vertex $\mathcal{V}_{n+k}$).

2. Similarly, ALAP time of a node M_k is defined as follows:

$$t^l_{n+k} = t^l_j - \min_{r\in[1,b]} c_{kr}$$

   where, T_j is the successor task node of M_k.

Example (contd.): Let us assume the deadline D of PTG G (in Figure 6.2a) to be 20 time units. Table 6.3 shows the ASAP and ALAP times corresponding to each task and message node in G, obtained through the above discussed procedure. For example, ASAP and ALAP times of task node T_1 are 1 and 9, respectively.

	T_1	T_2	T_3	T_4	T_5	T_6	M_1	M_2	M_3	M_4	M_5	M_6	M_7
ASAP	1	4	4	4	7	9	4	4	4	9	7	6	9
ALAP	9	12	14	17	17	19	10	11	14	14	15	18	17

Table 6.3: *ASAP & ALAP times of nodes in G (Figure 6.2a)*

Next, we present two ILP based strategies namely, ILP-ETR and ILP-NC. The design philosophies of ILP-ETR and ILP-NC are similar to the formulations ILP-SATC (Chapter 4, Section 4.3) and ILP-SANC (Chapter 4, Section 4.4) respectively, proposed in Chapter 4 for PTG scheduling on fully connected heterogeneous platform. However, the objective functions and many of the constraints in the ILPs presented in this chapter require certain modifications with respect to those in Chapter 4, as the current work assumes a distributed platform connected via shared buses. In order to improve continuity of discussion, completeness and better readability, we have discussed in details all the constraints and objective functions involved in ILP-ETR and ILP-NC.

6.3 ILP Formulation: ILP-ETR

In this section, we present an ILP based solution to the PTG scheduling problem. First, let us consider a set of binary decision variables: $X = \{X_{irt} : i = 1, 2, \ldots, n+m;\ r = 1, 2, \ldots, p+b;\ t = 1, 2, \ldots, D\}$. The variable $X_{irt} = 1$, when the PTG node $\mathcal{V}_i$ (i.e., task node T_i/message node M_{i-n}) starts on the resource R_r (i.e., processing element P_r/bus B_{r-p}) at the t^{th} time step; $X_{irt} = 0$, otherwise. We now present the required constraints on the binary variables X to model the scheduling problem.

6.3.1 Unique Start Time Constraints

The start time of each *task node* should be unique. That is, each task node T_i must start its execution at a unique time step t on a distinct processing element P_r.

$$\forall i \in [1, n] \quad \sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} X_{irt} = 1 \tag{6.1}$$

Similarly, each *message node* M_k must have a unique start time if M_k is actually transmitted over a bus (refer Assumption 5). So, the following constraint must hold:

$\forall M_k |\ T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$\sum_{r=p+1}^{p+b} \sum_{t=t_{k'}^s}^{t_{k'}^l} X_{k'rt} = 1 - Y_k \tag{6.2}$$

where,

$$k' = n + k \text{ and } Y_k = \sum_{r=1}^{p} \sum_{t_1=t_i^s}^{t_i^l} \sum_{t_2=t_j^s}^{t_j^l} X_{irt_1} * X_{jrt_2}$$

It may noted that in the above equation, $Y_k = 1$ when both the predecessor (T_i) and successor (T_j) task nodes of message node M_k are assigned to the same processing element P_r, forcing the LHS of Equation 6.2 to become 0. Otherwise, $Y_k = 0$. As X_{irt_1} and X_{jrt_2} are binary decision variables, we linearize their multiplication by introducing another binary decision variable $U_{krt_1t_2}$ $(= X_{irt_1} * X_{jrt_2})$ as shown below:

$$Y_k = \sum_{r=1}^{p} \sum_{t_1=t_i^s}^{t_i^l} \sum_{t_2=t_j^s}^{t_j^l} U_{krt_1t_2} \tag{6.3}$$

6.3.2 Linearization of Non-linear Term

Now, the non-linear variables $U_{krt_1t_2}$ can be linearized using the following four inequalities.

$$X_{irt_1} \geqslant U_{krt_1t_2} \tag{6.4}$$

$$X_{jrt_2} \geqslant U_{krt_1t_2} \tag{6.5}$$

$$U_{krt_1t_2} \geqslant X_{irt_1} + X_{jrt_2} - 1 \tag{6.6}$$

$$U_{krt_1t_2} \in \{0, 1\} \tag{6.7}$$

6.3.3 Resource Constraints

Resource bounds must be satisfied at each time step for both processing elements and buses. Any resource R_r can execute/transmit at most one task/message node at a given time. In this regard, it may be noted that a task node T_i can only be executing on processing element P_r at time t, if it has started at most $t - e_{ir} + 1$ time steps earlier.

$$\forall t \in [1, D] \text{ and } \forall r \in [1, p] \quad \sum_{i=1}^{n} \sum_{t'=\psi}^{t} X_{irt'} \leqslant 1 \tag{6.8}$$

where, $\psi = t - e_{ir} + 1$.

Similarly, a message node M_k can only be transmitting through the bus B_r at time t, if it has started at most $t - c_{kr} + 1$ time steps earlier. The range of t is

$$\forall t \in [1, D] \text{ and } \forall r \in [1, b] \quad \sum_{k=1}^{m} \sum_{t'=\psi}^{t} X_{k'r't'} \leqslant 1 \tag{6.9}$$

where, $k' = k + n$, $r' = r + p$ and $\psi = t - c_{kr} + 1$.

6.3.4 Dependency Constraints

The dependencies between nodes must be satisfied. The following three constraints enforce satisfaction of the precedence relationships among task and message nodes of a

PTG. Constraints 6.10 and 6.11 assert that the preceding task node (say, T_i) of any message node (say, M_k) completes its execution (i) before the start of the succeeding task node (say, T_j) of M_k (in case, both T_i and T_j are assigned to the same processing element) and, (ii) before the start of M_k (in case, both T_i and T_j are assigned to different processing elements).

$\forall M_k |\ T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$\sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} (t + e_{ir}) * X_{irt} \leqslant \sum_{r=1}^{p} \sum_{t=t_j^s}^{t_j^l} t * X_{jrt} \tag{6.10}$$

$$\sum_{r=1}^{p} \sum_{t=t_i^s}^{t_i^l} (t + e_{ir}) * X_{irt} \leqslant \sum_{r=p+1}^{p+b} \sum_{t=t_{k'}^s}^{t_{k'}^l} t * X_{k'rt} + C * Y_k \tag{6.11}$$

where, $k' = n + k$ and C is a constant. It may be observed that by setting C to a sufficiently large value, the constraint in Equation 6.11 is trivially satisfied when both T_i and T_j are assigned to the same processing element ($Y_k = 1$). Suppose, the M_k^{th} message node is scheduled on a bus (i.e., $Y_k = 0$). Then, task node T_j ($= succ(M_k)$) should commence its execution only after the completion of M_k. This constraint is represented as follows:

$\forall M_k |\ T_j = succ(M_k)$,

$$\sum_{r=p+1}^{p+b} \sum_{t=t_{k'}^s}^{t_{k'}^l} (t + c_{kr}) * X_{k'rt} \leqslant \sum_{r=1}^{p} \sum_{t=t_j^s}^{t_j^l} t * X_{jrt} \tag{6.12}$$

where, $k' = n + k$. It is noteworthy that when $Y_k = 1$, the constraint imposed by Equation 6.2 enforces $\sum_{r=p+1}^{p+b} \sum_{t=t_{k'}^s}^{t_{k'}^l} X_{k'rt}$ to be 0. Hence, $\sum_{r=p+1}^{p+b} \sum_{t=t_{k'}^s}^{t_{k'}^l} (t + c_{kr}) * X_{k'rt}$ in the LHS of Equation 6.12 also reduces to 0. So, Constraint 6.12 is implicitly satisfied, when Y_k is 1.

6.3.5 Deadline Constraint:

All tasks in the PTG have to complete their execution within the deadline D. This can be satisfied by restricting the finish time of the sink node to be at most the deadline D.

This constraint can be written as,

$$\sum_{r=1}^{p}\sum_{t=t_n^s}^{t_n^l} X_{nrt} * (t + e_{nr}) - 1 \leq D \tag{6.13}$$

6.3.6 Objective Function

Our objective is to minimize the schedule length of PTG G. It can be achieved by minimizing the finish time of the sink node T_n. Hence, the objective function can be written as:

$$Minimize \sum_{r=1}^{p}\sum_{t=t_n^s}^{t_n^l} X_{nrt} * (t + e_{nr}) \tag{6.14}$$

subject to constraints presented in Equations 6.1 - 6.13.

6.3.7 Complexity Analysis

The complexity of the proposed formulation ILP-ETR can be analyzed in terms of the total number of constraints and the total number of variables per constraint. Such an analysis for ILP-ETR is presented in Table 6.4. The total complexity of ILP-ETR (in terms of number of constraints) can be obtained as $O(n + m \times D \times max\{p, b\})$. Considering $m >> n$ and $p >> b$, the total complexity becomes $O(m \times p \times D)$.

Example (contd.): Applying the ILP-ETR procedure discussed above on our example PTG G (Figure 6.2a), we obtain the schedule represented through the gantt chart depicted in Figure 6.3. This problem generates 4927 constraints and takes 0.8 seconds when solved using the CPLEX optimizer [10]. It may be noted that the schedule assigns unique start times to all tasks/messages and satisfies resource bounds, dependency constraints and deadline. A further observation is that, the message nodes M_1, M_4 and M_7 have not been actually scheduled. This is because, all the predecessor and successor task nodes of M_1, M_4 and M_7 (i.e., T_1, T_2, T_5 and T_6) are scheduled by the ILP-ETR on the same processing element P_2. The optimal schedule length of G is obtained as 16 time units.

Constraint Type	Equation No.	#Constraints	#Variables Per Constraint
Unique Start Time	6.1	$O(n)$	$O(p \times D)$
	6.2	$O(m)$	$O(max\{(b \times D), (p \times D^2)\})$
Resource	6.8	$O(p \times D)$	$O(n \times D)$
	6.9	$O(b \times D)$	$O(m \times D)$
Dependency	6.10	$O(m)$	$O(p \times D)$
	6.11	$O(m)$	$O(max\{p, b\} \times D)$
	6.12	$O(m)$	$O(max\{p, b\} \times D)$
Deadline	6.13	$O(1)$	$O(p \times D)$
Linearization	6.4	$O(m \times D \times max\{p, b\})$	$O(1)$
	6.5	$O(m \times D \times max\{p, b\})$	$O(1)$
	6.6	$O(m \times D \times max\{p, b\})$	$O(1)$

Table 6.4: *Complexity of ILP-ETR*

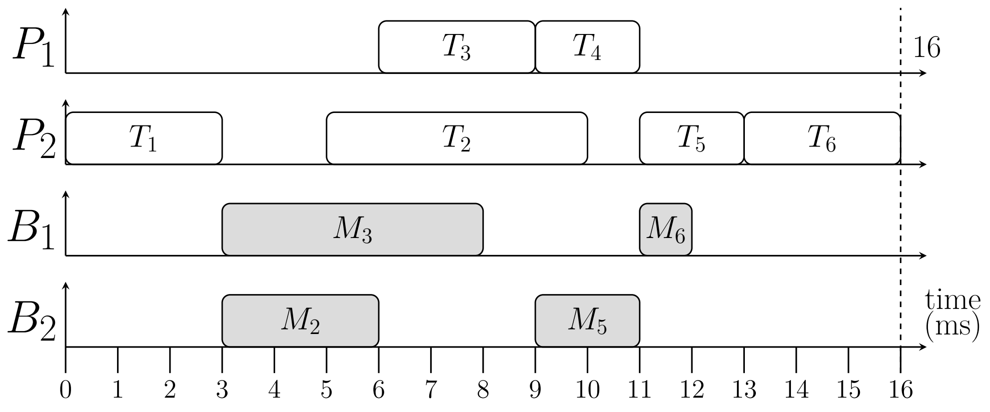


Figure 6.3: *The schedule for the PTG (Figure 6.2a) using ILP-ETR*

6.4 ILP Formulation: ILP-NC

It may be observed that the complexity of ILP-ETR presented in the earlier section depends on the number of processors, the deadline and the number of edges associated with a given PTG. In order to improve its scalability, we propose an improved ILP formulation based on the *non-overlapping approach* [9] which sets constraints and variables in such a way that no two tasks executing on the same processor overlap in time. Further, the total number of constraints required to compute a schedule for a PTG becomes independent of the deadline of a PTG.

Before presenting ILP-NC, we first introduce the set of decision variables. Start time of each node is captured by an integer decision variable $S_i \in \mathbb{Z}^+$, where $\mathbb{Z}^+$ denotes the set of positive integers. The formulation also uses three sets of binary decision variables namely, X_{ir}, α_{ij}, and β_{ij}. Here, variables X_{ir} are used to capture task-to-processor and message-to-bus mappings for the task and message nodes, respectively. Variables α_{ij} are used to determine the precedence order of execution between mutually independent task pairs. Variables β_{ij} are used to determine message transmission precedence orders among mutually independent message-pairs. Variable $X_{ir} = 1$, if the PTG node $\mathcal{V}_i$ (i.e., task node T_i/message node M_{i-n}) is assigned to the resource R_r (i.e., processing element P_r/bus B_{r-p}); $X_{irt} = 0$, otherwise. Variable $\alpha_{ij} = 1$, if task T_i starts before task T_j; $\alpha_{ij} = 0$, otherwise. Finally, variable $\beta_{ij} = 1$, if message M_i starts before message M_j; $\beta_{ij} = 0$, otherwise. Now, we present the required set of constraints on decision variables to schedule a PTG on a given heterogeneous distributed platform.

6.4.1 Unique Resource Assignment:

Each task node T_i can execute only on one processing element P_r (resource R_r).

$$\forall i \in [1, n] \quad \sum_{r=1}^{p} X_{ir} = 1 \tag{6.15}$$

Similarly, each message node M_k should be transmitted uniquely through one bus B_r (resource R_{p+r}).

$\forall M_k | T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$\sum_{r=p+1}^{p+b} X_{k'r} = 1 - \sum_{r=1}^{p} Z_{ijr} \tag{6.16}$$

where, $k' = n + k$ and $Z_{ijr} = X_{ir} * X_{jr}$. It may be noted that, $\sum_{r=1}^{p} Z_{ijr} = 1$, when both the predecessor (T_i) and successor (T_j) task nodes of message node M_k are assigned onto the same processing element P_r, forcing the LHS of Equation 6.16 to become 0.

6.4.2 Dependency Constraints:

The dependencies between nodes must be satisfied. For the dependency relation $T_i \rightarrow M_k \rightarrow T_j$, T_i must finish its execution before message node M_k starts its transmission. However, M_k vanishes when both T_i and T_j are executed on the same processing element. Thus, the message transmission time corresponding to M_k must be neglected in this case.

$\forall M_k |\ T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$S_i + \sum_{r=1}^{p} e_{ir} * X_{ir} \leq S_k + C * \sum_{r=1}^{p} Z_{ijr} \tag{6.17}$$

where, $Z_{ijr} = X_{ir} * X_{jr}$ and C is a sufficiently large constant. It can be seen that when both tasks T_i and T_j execute on the same processor P_r, Z_{ijr} becomes 1, causing the Equation 6.17 to be trivially true (as $S_k << C$). The term, $S_i + \sum_{r=1}^{p} e_{ir} * X_{ir}$, captures the absolute finish time of T_i when assigned on processor P_r.

Similarly, for the dependency relation $T_i \rightarrow M_k \rightarrow T_j$, M_k must finish its transmission before task node T_j starts execution.

$\forall M_k |\ T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$S_k + \sum_{r=p+1}^{p+b} c_{kr} * X_{k'r} \leq S_j + C * \sum_{r=1}^{p} Z_{ijr} \tag{6.18}$$

where, $k' = n + k$, $Z_{ijr} = X_{ir} * X_{jr}$ and C is a sufficiently large constant. Equation 6.18, produces m constraints and the number of variables per constraint is $O(max\{p, b\})$. Finally, for the dependency relation $T_i \rightarrow M_k \rightarrow T_j$, T_i must be enforced to complete before T_j. This constraint becomes important when T_i and T_j are executed on the same processing element, causing message node M_k to vanish.

$\forall M_k |\ T_i = pred(M_k)$ and $T_j = succ(M_k)$,

$$S_i + \sum_{r=1}^{p} e_{ir} * X_{ir} \leq S_j \tag{6.19}$$

6.4.3 Non-overlapping Constraints:

The execution of any two tasks T_i and T_j assigned to the same processor cannot be overlapped in time. The dependency constraints discussed above automatically enforce the non-overlapping property for task pairs which share an explicit precedence relationship between them. For the remaining mutually independent task pairs, this constraint is enforced as follows:

$\forall T_i, T_j$ $(i = 1, \ldots, n-1;\ j = i+1, \ldots, n)$ | T_i and T_j do not have an ancestor-descendant relationship,

$$S_i + \sum_{r=1}^{p} e_{ir} * X_{ir} - S_j - C * \left(1 - \alpha_{ij}\right) - C * \left(1 - \sum_{r=1}^{p} Z_{ijr}\right) \leq 0 \qquad (6.20)$$

$$S_j + \sum_{r=1}^{p} e_{jr} * X_{jr} - S_i - C * \alpha_{ij} - C * \left(1 - \sum_{r=1}^{p} Z_{ijr}\right) \leq 0 \qquad (6.21)$$

where, $Z_{ijr} = X_{ir} * X_{jr}$ and C is a sufficiently large constant.

Here, Constraint 6.20 enforces non-overlap between the executions of T_i and T_j, when T_i starts before T_j. Similarly, Constraint 6.21 enforces this property when T_j starts before T_i. The last term in the LHS of both Equations 6.20 and 6.21 vanishes, when T_i and T_j are assigned onto the same processor P_r. Otherwise, the large constant C makes both the constraints to be trivially satisfied.

It may be noted that the second last term in the LHS of Equation 6.20 (i.e., $C * (1 - \alpha_{ij})$) vanishes when $\alpha_{ij} = 1$. That is, T_i starts before T_j. Otherwise, the constant C makes the constraint to be trivially satisfied. Similarly, the term $(C * \alpha_{ij})$ in Equation 6.21 vanishes when T_j starts before T_i and makes the constraint to be trivially satisfied, otherwise.

When T_i starts before T_j on the same processor P_r (i.e., $Z_{ijr} = \alpha_{ij} = 1$), Constraint 6.20 enforces T_j to commence after the completion of execution of T_i. A similar set of arguments hold for Constraint 6.21 as well.

Similarly, the transmission of any two messages M_i and M_j assigned onto the same bus cannot be overlapped in time. The dependency constraints discussed above automatically enforce the non-overlapping property for message pairs which share an explicit precedence relationship between them. For the remaining mutually independent message-pairs, this constraint is enforced as follows:

$\forall M_i, M_j$ $(i = 1, \dots, m-1;\ j = i+1, \dots, m) \mid M_i$ and M_j do not have an ancestor-descendant relationship,

$$S_i + \sum_{r=p+1}^{p+b} c_{ir} * X_{i'r} - S_j - C * \left(1 - \beta_{ij}\right) - C * \left(1 - \sum_{r=p+1}^{p+b} Z_{i'j'r}\right) \leq 0 \tag{6.22}$$

$$S_j + \sum_{r=p+1}^{p+b} c_{jr} * X_{j'r} - S_i - C * \beta_{ij} - C * \left(1 - \sum_{r=p+1}^{p+b} Z_{i'j'r}\right) \leq 0 \tag{6.23}$$

Where $i' = n + i$, $j' = n + j$ and $Z_{i'j'r} = X_{i'r} * X_{j'r}$.

6.4.4 Linearization of Non-linear Term

The non-linear variables $Z_{ijr} = X_{ir} * X_{jr}$ in Equations 6.16, 6.17, 6.18, 6.20 and 6.21 can be linearized using the following four inequalities.

$$Z_{ijr} \leq X_{ir} \tag{6.24}$$

$$Z_{ijr} \leq X_{jr} \tag{6.25}$$

$$Z_{ijr} \geq X_{ir} + X_{jr} - 1 \tag{6.26}$$

$$Z_{ijr} \in \{0, 1\} \tag{6.27}$$

Similarly, $Z_{i'j'r} = X_{i'r} * X_{j'r}$ in Equations 6.22 and 6.23 can be linearize using above four equations, where $i = i'$ and $j = j'$.

6.4.5 Deadline Constraint:

All tasks in the PTG have to complete their execution within the deadline D. This can be satisfied by restricting the finish time of the sink node to be at most the deadline D. This constraint can be written as,

$$S_n + \sum_{i=1}^{p} (e_{nr} * X_{nr}) - 1 \leq D \tag{6.28}$$

6.4.6 Objective Function

Our objective is to minimize the makespan (schedule length) while satisfying all schedulability constraints. The objective function can be written as:

$$Minimize\ S_n + \sum_{i=1}^{p}(e_{nr} * X_{nr}) - 1 \tag{6.29}$$

subject to constraints presented in Equations 6.15 - 6.28.

6.4.7 Complexity Analysis

The complexity analysis for ILP-NC is presented in Table 6.5. The total complexity of ILP-NC (in terms of the number of constraints) is $O(max\{n^2 \times p, m^2 \times b\})$. It may be noted that the complexity of ILP-NC is independent of the deadline of a PTG.

Constraint Type	Equation No.	#Constraints	#Variables Per Constraint
Unique Resource	6.15	$O(n)$	$O(p)$
	6.16	$O(m)$	$O(max\{p, b\})$
Dependency	6.17	$O(m)$	$O(p)$
	6.18	$O(m)$	$O(max\{p, b\})$
	6.19	$O(m)$	$O(p)$
Non-overlapping	6.20	$O(n^2)$	$O(p)$
	6.21	$O(n^2)$	$O(p)$
	6.22	$O(m^2)$	$O(b)$
	6.23	$O(m^2)$	$O(b)$
Deadline	6.28	$O(1)$	$O(p)$
Linearization	6.24	$O(max\{n^2 \times p, m^2 \times b\})$	$O(1)$
	6.25	$O(max\{n^2 \times p, m^2 \times b\})$	$O(1)$
	6.26	$O(max\{n^2 \times p, m^2 \times b\})$	$O(1)$

Table 6.5: *Complexity of ILP-NC*

Example (contd.): Applying ILP-NC procedure discussed above on our example PTG G (Figure 6.2a), we obtain the schedule represented through the gantt chart depicted in Figure 6.4. The optimal schedule produced (16) is same as that obtained using ILP-ETR. However, it is worth noting that ILP-NC generates only 286 constraints and takes 0.1 seconds to produce the solution. On the contrary, ILP-ETR produced 4927 constraints and takes 0.8 seconds.

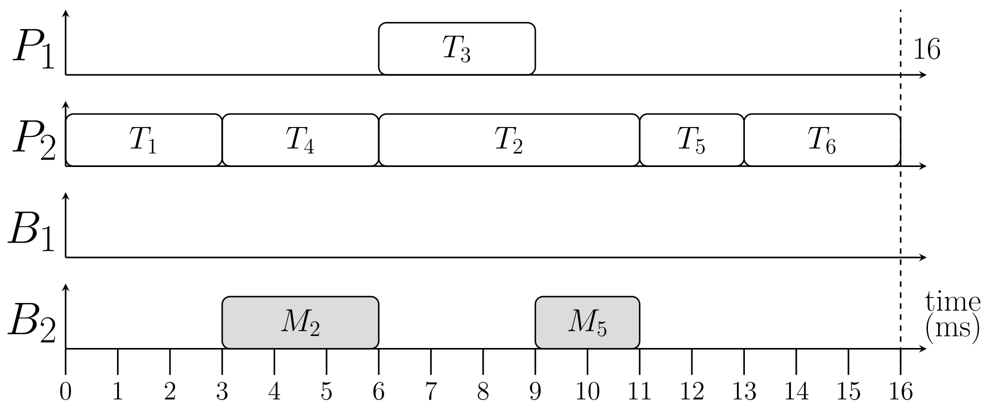


Figure 6.4: *The schedule for the PTG (Figure 6.2a) using ILP-NC*

Though, ILP-NC shows an appreciable improvement in terms of scalability compared to ILP-ETR, it also suffers from high computational overheads (in terms of running time) as the number of nodes in a PTG and/or the number of resources, increase. For example, in our experiments, we have observed that the ILP-NC takes more than ~4 *hrs* to find feasible schedules for PTGs with ~80 nodes, on a platform with four heterogeneous processors and two shared buses. It may be noted that such large time overheads may often not be affordable, especially when multiple quick design iterations are needed during design space exploration. Therefore, we propose a low-overhead heuristic namely, *Contention Cognizant Task and Message Scheduler* (CC-TMS) in the next section.

6.5 Heuristic: CC-TMS

Typically, *list scheduling* based heuristic techniques have been employed to compute feasible schedules for PTGs executing on heterogeneous platforms. Some examples of this category of schemes are HEFT [6], PEFT [1] and HSV [11]. These algorithms try to generate an offline schedule for the given PTG executing on a fully-connected heterogeneous platform with the goal of obtaining minimum makespans, while ensuring precedence as well as resource constraints. On the contrary, our work considers a heterogeneous distributed platform which uses shared buses for interconnecting processing elements. For this platform setting, we devise a heuristic based algorithm CC-TMS. Before presenting the algorithm, we define a few related attributes such as *upward rank*, *Earliest Start Time* (EST) and *Earliest Finish Time* (EFT).

6.5.1 Upward Rank

The upward rank of a node in G is determined in a recursive fashion by traversing PTG upwards, starting from the sink node T_n towards the source node T_1. For T_n, the upward rank is defined as:

$$rank_u(T_n) = \overline{e_n} \tag{6.30}$$

where, $\overline{e_n}$ denotes the average execution time of T_n over all processors in P. For the rest of the task and message nodes, upward rank is determined as:

$$rank_u(T_i) = \overline{e_i} + \max_{M_k \in succ(T_i)} rank_u(M_k) \tag{6.31}$$

where, $succ(T_i)$ represents the message nodes which immediately succeed task T_i and,

$$rank_u(M_k) = \overline{c_k} + rank_u(T_j) \tag{6.32}$$

where, $\overline{c_k}$ is the average data transmission time of message node M_k over all buses in B, and $rank_u(T_j)$ is the rank of the successor task node T_j of M_k.

6.5.2 Earliest Start and Finish Time

$EST(T_i, P_r)$ of a task T_i on processor P_r denotes the earliest time at which T_i can start on P_r. Similarly, $EST(M_k, B_r)$ denotes the earliest time at which transmission of a message M_k can commence on bus B_r. We recursively compute EST of each node starting from the source node of PTG G. The EST of the source node (T_1) which is a task node is,

$$EST(T_1, P_r) = 0 \tag{6.33}$$

The EST of other task nodes are computed as,

$$EST(T_j, P_r) = max\{avail[P_r], \max_{M_k \in pred(T_j)} AFT(M_k)\} \tag{6.34}$$

where, $avail[P_r]$ denotes the earliest instant at which P_r is available and $AFT(M_k)$ represents the actual time at which the predecessor message M_k finishes transmission. The EST of the message nodes are computed as,

$$EST(M_k, B_r) = max\{avail[B_r], AFT(T_i)\} \tag{6.35}$$

where, $avail[B_r]$ denotes the earliest instant at which B_r becomes available for message transmission and $AFT(T_i)$ is the actual time at which the predecessor task T_i of M_k finishes execution.

The EFT of all task and message nodes are computed as,

$$EFT(T_i, P_r) = EST(T_i, P_r) + e_{ir} \tag{6.36}$$

$$EFT(M_k, B_r) = EST(M_k, B_r) + c_{kr} \tag{6.37}$$

For any message node M_k, when both its predecessor as well as successor tasks T_i and T_j are scheduled onto the same processor, then existence of the message node M_k becomes immaterial and is not transmitted. That is, $EST(M_k, B_r) = EFT(M_k, B_r) = AFT(T_i)$.

6.5.3 Co-scheduling Tasks and Messages

The proposed algorithm called CC-TMS is given in Algorithm 4. Initially, the CC-TMS algorithm computes the upward rank ($rank_u$) of each node (line 1) and makes a priority list (*TaskPriorityList*) of the task nodes in PTG G sorted in non-increasing order of the tasks' upward ranks (line 2). Next, it initializes $avail[P_r]$ to 0 for each $P_r \in P$ and $avail[B_r]$ to 0 for each $B_r \in B$ (line 3). At any given iteration, the first node in the

```
ALGORITHM 4: CC-TMS
Input: PTG $G$, Platform $\rho$
Output: Schedule of task and message nodes (Start time of each node and
        mapping of tasks/messages to processors/buses)
1  Compute upward rank ($rank_u$) of each node in $G$
2  Let $TaskPriorityList$ be the list of tasks arranged in non-increasing order of
     the tasks' upward ranks
3  For each processor $P_r \in P$ and bus $B_r \in B$, set $avail[P_r]$ and $avail[B_r]$ to 0
4  while $TaskPriorityList$ is non-empty do
5      Select the first task $T_i$, from the list $TaskPriorityList$
6      Let $MsgPriorityList_i$ be the list consisting of all predecessor message
         nodes of task $T_i$ arranged in non-increasing order of $rank_u$
7      for each processor $P_r \in P$ do
8          Tentatively assign $T_i$ on $P_r$
9          For each bus, set $tempAvail[B_r] = avail[B_r]$
10         for each $M_j \in MsgPriorityList_i$ do
11             for each bus $B_k$ do
12                 Compute $EFT(M_j, B_k)$
13             Tentatively assign $M_j$ on that bus $B_k$ for which $EFT(M_j, B_k)$ is
                 minimum and update $tempAvail[B_k]$ as $EFT(M_j, B_k)$
14         Compute $EFT(T_i, P_r)$
15     Actually assign $T_i$ on that $P_r$ on which $EFT(T_i, P_r)$ is minimum and
         update $avail[P_r]$ as $EFT(T_i, P_r)$
16     Given $T_i$ on $P_r$, assign all its predecessor message nodes on buses such that
         their EFT's are minimized and update $avail[B_r]$ accordingly
17     Remove $T_i$ from $TaskPriorityList$
```

current task priority list is selected and scheduled along with its predecessor message nodes (line nos 4 to 17).

The steps within each iteration subsequent to the selection of a task node T_i is explained as follows: First, Algorithm 4 computes the priority list $MsgPriorityList_i$ consisting of all predecessor message nodes of T_i, arranged in non-increasing order of their upward ranks (line 6). Then, T_i is tentatively assigned on each processor $P_r \in P$ (line 8) and for any such tentative allocation, each predecessor message node M_j is tentatively assigned to that bus $B_k \in B$ for which EFT value of the message node M_j is minimum (lines 10 to 13). The message-to-bus allocation is conducted in the order as prescribed by the message priority list. Based on these message node allocations, $EFT(T_i, P_r)$ is computed for all $P_r \in P$. Finally, T_i is actually assigned on the processor (say, P_x) for which its EFT is minimum. Given this allocation of T_i on P_x, its predecessor message nodes are assigned on buses such that their EFT's are minimized (line 16).

6.5.4 Complexity Analysis

The computation of the upward ranks for the nodes in PTG G requires a single traversal pass over all nodes and consumes $O(n+m)$ time (line no. 1). Creation of *TaskPriorityList* and *MsgPriorityList*$_i$ for each task T_i can be computed along with the generation of the nodes' rank values. Through additional list data structures and limited sorting over small constant number of elements, these lists can be obtained within constant time additional complexity on average, over the $O(n+m)$ overhead incurred for upward rank generation. The initialization of $avail[P_r]$ and $avail[B_r]$ takes $O(p+b)$ time (line no. 3). Next, it may be observed that the complexity of the *while* loop (line nos. 4 to 17) is dominated by the total complexity of line no. 12 within the inner-most *for* loop. This total complexity may be obtained by multiplying the complexity of $EFT()$ function at this step with the total number of times line no. 10 is executed in the algorithm. From Equations 6.36 and 6.34, it may be inferred that $EFT(M_j, B_k)$ incurs constant time complexity. Also, we see that the number of times line no. 10 is executed is: $(n \times p \times (n-1) \times b)$, where n denotes the total number of times the while loop is executed, while p, $(n-1)$, and b denote the number of times the for loops at line nos. 7, 10, and 11 are executed, respectively. Hence, the total complexity of the while loop is given by $O(n \times p \times (n-1) \times b)$. Therefore, the complexity of the CC-TMS algorithm may be represented as $O(n^2 \times p \times b)$. This time complexity is slightly higher than the running time of HEFT [6] and PEFT [1] algorithms ($O(n^2 \times p)$) since these algorithms assume *fully-connected* platform against a shared bus system as considered in this work.

T_1	T_2	T_3	T_4	T_5	T_6	M_1	M_2	M_3	M_4	M_5	M_6	M_7
24	18	14	7	8	2.5	20.5	17.5	11	11.5	10.5	4.5	5

Table 6.6: *Upward rank of nodes in PTG G (Figure 6.2)*

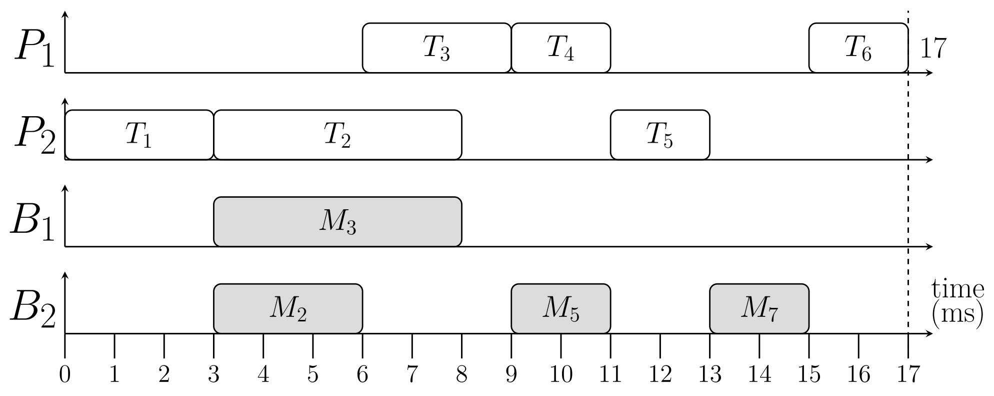


Figure 6.5: *The schedule for the PTG (Figure 6.2a) using CC-TMS*

6.5.5 Example

Refer to the PTG shown in Figure 6.2a. The CC-TMS algorithm (Algorithm 4) initially computes the upward rank of each node (listed in Table 6.6) and makes a sorted priority list of task nodes ($TaskPriorityList$): $\{T_1$, T_2, T_3, T_5, T_4, $T_6\}$. Time of availability for the processors and buses are set to, $avail[P_1] = avail[P_2] = avail[B_1] = avail[B_2] = 0$. The execution of the algorithm is as follows:

T_1 : The CC-TMS algorithm selects task T_1 from the task priority list ($TaskPriorityList$). Since T_1 does not have any predecessors, its message priority list ($MsgPriorityList_1$) is empty.

P_1 : $EST(T_1, P_1) = 0$, $EFT(T_1, P_1) = 0 + 4 = 4$

P_2 : $EST(T_1, P_2) = 0$, $EFT(T_1, P_2) = 0 + 3 = 3$.

Since $EFT(T_1, P_1) > EFT(T_1, P_2)$, T_1 is assigned to processor P_2 with start time 0. Availability of processor P_2 is updated to $avail[P_2] = 3$. Then, T_1 is removed from the task priority list.

T_2 : Next, task T_2 is selected from the task priority list. The message priority list of task T_2 is $\{M_1\}$.

P_1 : If T_2 is assigned on processor P_1 then $tempAvail[B_1] = tempAvail[B_2] = 0$;

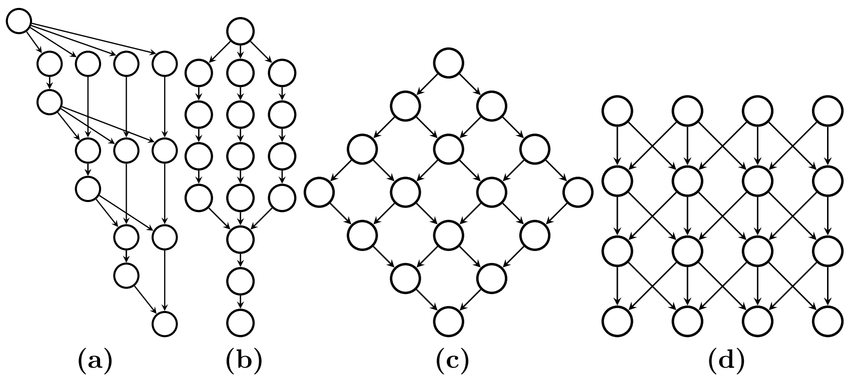


Figure 6.6: *(a) Gaussian Elimination [6], (b) Epigenomics [2] (c) Laplace [4] (d) Stencil [4]*

$EST(M_1, B_1) = EST(M_1, B_2) = 3$; $EFT(M_1, B_1) = 3{+}2 = 5$, $EFT(M_1, B_2) = 3 + 3 = 6$; $tempAvail[B_1] = 5$; $EST(T_2, P_1) = 5$, $EFT(T_2, P_1) = 5 + 8 = 13$.

P_2 : If T_2 is assigned on processor P_2 then M_1 becomes invalid. $EST(M_1, B_1) = EST(M_1, B_2) = 3$; $EFT(M_1, B_1) = EFT(M_1, B_2) = 3$; $EST(T_2, P_2) = 3$, $EFT(T_2, P_2) = 3 + 5 = 8$.

Since $EFT(T_2, P_1) > EFT(T_2, P_2)$, T_2 is assigned to processor P_2 and the message M_1 is discarded (as predecessor T_1 and successor T_2 of M_1 are scheduled on the same processor). Availability of processor P_2 is updated to $avail[P_2] = 8$.

Similarly, the schedule generation proceeds for the rest of the task and message nodes. The resulting list schedule is shown in Figure 6.5. It may be noted that the obtained schedule length is 17 (slightly greater than the optimal makespan (16) returned by both ILP-ETR and ILP-NC).

6.6 Experimental Evaluation

The performance of the strategies proposed in this work, ILP-ETR, ILP-NC and CC-TMS, has been experimentally evaluated using real-world benchmark PTGs.

Experimental Setup: Four benchmark PTGs namely, *Gaussian Elimination* [6], *Epigenomics* [2], *Laplace* [4] and *Stencil* [4], have been employed to experimentally test and compare the algorithms. Figures 6.6a, 6.6b, 6.6c and 6.6d, respectively show the structural illustrations of the four PTGs considered.

- The task graph representation of a *Gaussian Elimination* equation solver is determined by the number (χ) of linear equations which the algorithm attempts to solve. A *Gaussian Elimination* task graph contains $((\chi^2 + \chi - 2)/2)$ task nodes and $(\chi^2 - \chi - 1)$ message nodes when the number of equations to be solved is χ. As example, Figure 6.6a shows a Gaussian Elimination task graph containing 14 task nodes and 19 message nodes when $\chi = 5$.

- *Epigenomics* represents a procedure for genome sequencing operations in a parallel pipelined manner. Given the number of parallel branches γ in an *Epigenomics* PTG, the number of task nodes and message nodes in it are given by $(4\gamma + 4)$ and $(5\gamma + 2)$, respectively. As example, for the Epigenomics PTG shown in Figure 6.6b, the number of parallel branches, $\gamma = 3$ and therefore, the graph has 16 task nodes and 17 message nodes.

- The *Laplace* PTG corresponds to the Laplace algorithm for equation solving with the number of task and message nodes being represented as φ^2 and $(2\varphi^2 - 2\varphi)$, respectively, where φ is the size of matrix given as input to the algorithm. As example, for the Laplace PTG shown in Figure 6.6c, the number of parallel branches $\varphi = 4$ and so, the graph has 16 task nodes and 24 message nodes.

- The *Stencil* PTG is associated with the procedure for solving partial differential equations. The solution proceeds level-wise with each level containing ξ task nodes. The Stencil PTG corresponding to a solution which involves λ levels, contains $(\lambda \times \xi)$ task nodes and $[(\lambda - 1) \times (3\xi - 2)]$ message nodes. In the experiments, we have assumed $\lambda = \xi$ in the interest of simplicity. For example, Figure 6.6d shows a Stencil task graph containing 16 task nodes and 30 message nodes ($\lambda = 4$).

The data associated with the PTGs and also the computing platform, have been varied over carefully chosen ranges of values to evaluate and exhibit the efficacy of the proposed algorithms on various possible scenarios that they may encounter in practice: (1) *Matrix Sizes*: $\chi = \{3, 4, 5, 6\}$ (*Gaussian Elimination*); *Parallel Branches*: $\gamma = \{2, 3, 4, 5\}$ (*Epigenomics*); *Matrix Sizes*: $\varphi = \{2, 3, 4, 5\}$ (*Laplace*); *Number of Levels*: $\lambda = \{2, 3, 4, 5\}$ (*Stencil*). (2) *Number of processors*: Platforms consisting of $p = \{2, 4, 6, 8\}$ processors have been considered. (3) *Number of buses*: Platforms consisting of $b = \{1, 2, 3, 4\}$ buses have been considered. (4) *Communication to Computation Ratio* $CCR = \{0.5, 1, 1.5, 2\}$ (CCR denotes the ratio of the average cost of communication to computation for the given PTG). (5) A uniform random distribution ([10 ms, 30 ms]) has been used to generate *execution times* e_{ir} for each task(T_i)-to-processor(P_r) mapping. (6) Another uniform random distribution ([10 ms, 30 ms]) is used to generate *communication time* c_{kr} for each message(M_k)-to-bus(B_r) mapping. Based on the desired CCR required in a given scenario, the obtained communication times are accordingly scaled. (7) Deadline Extension Ratio ($DR \in \{1, 1.25, 1.5, 1.75, 2\}$). Here, the parameter DR refers to the ratio between the given deadline corresponding to the execution of a PTG and its obtained optimal schedule length. The CPLEX optimizer [10] version 12.6.2.0 has been used. All experiments have been carried-out on a system having Intel(R) Xeon(R) CPU running Linux Kernel 3.10.0-693.21.1.el7.x86_64.

Performance Metric: Performance evaluation of the proposed CC-TMS algorithm and its comparison with ILP based solution has been conducted with the metric called *Makespan Ratio* which is defined as,

$$Makespan\ Ratio = \frac{Optimal\ Makespan(ILP)}{Heuristic\ Makespan(CC-TMS)} \times 100 \tag{6.38}$$

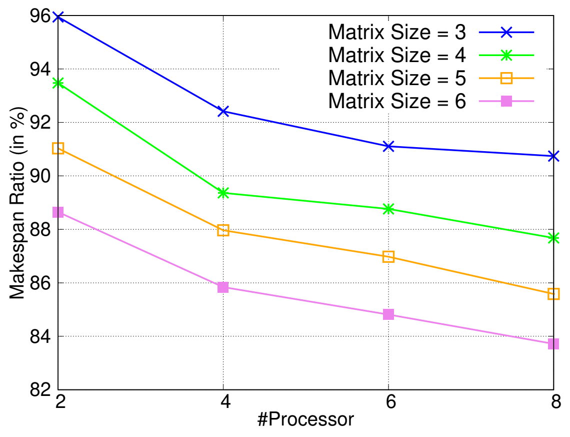
Experiment-1: Comparing ILP-ETR and ILP-NC: We set p to 4, b to 2 and CCR to 1. The matrix size (χ) for *Gaussian Elimination*, number of parallel branches (γ) for *Epigenomics*, matrix size (φ) for *Laplace* and number of levels (λ) for *Stencil*) are set to 3 and 4. The running time of both ILP-ETR and ILP-NC, when the deadline extension ratio is varied from 1 to 2, is shown in the Table 6.7. It may be seen from the table that ILP-NC comprehensively outperforms ILP-ETR for all scenarios considered. The results also show that run-times corresponding to ILP-ETR are significantly more sensitive to the increase in both the number of nodes as well as deadline extension rate, compared to ILP-NC. Further, it may be observed that run-times of the ILP schemes differ vastly between PTGs of different types. For example, let us consider *Guassian Elimination* and *Stencil* PTGs with shape parameter = 3 and deadline extension rate = 1. In this scenario, ILP-ETR takes $0.94s$ and $11m : 6s$ for *Guassian Elimination* and *Stencil*, respectively. Similarly, ILP-NC takes $0.7s$ and $23.23s$ for *Guassian Elimination* and *Stencil*. This variation in the running times of ILPs for different PTG types is mainly due to the difference in their internal structures. More specifically for a given shape parameter, it may be observed that (1) the structure (in terms of the number of nodes and edges) of *Stencil* (refer Figure 6.6d) is far denser than *Guassian Elimination* (refer Figure 6.6a) and (2) the number of constraints and variables that must be simultaneously taken into consideration to compute a solution is higher for *Stencil* compared to *Guassian Elimination.* Consequently, computations times required to generate optimal solutions for *Stencil* increase steeply as the number of nodes increase, thus exhibiting poor scalability. On the contrary, it may be observed from Table 6.7 that computation times for the other types of PTGs namely, *Guassian Elimination, Epigeneomics, Laplace*, are significantly lower compared to *Stencil*, for a similar number of total nodes. Thus, it may be inferred from the results that *Guassian Elimination, Epigeneomics, Laplace*, exhibit much better scalability than *Stencil.* Further, it may be noted that the running-time of ILP-NC for a particular PTG with a given shape parameter, remains almost constant independent of the laxity available before deadline. For example, the running-time of ILP-NC for *Stencil* remains as $\sim$ 23s when the value of the shape parameter λ is 3, irrespective of the variation in deadline extension rate.

Experiment-2: Varying the number of processors: The results of this experiment is depicted in Figure 6.7. Here, the number of processors (p) is varied from 2 to 8, matrix size χ (*Gaussian Elimination*) between 3 and 6, number of parallel branches

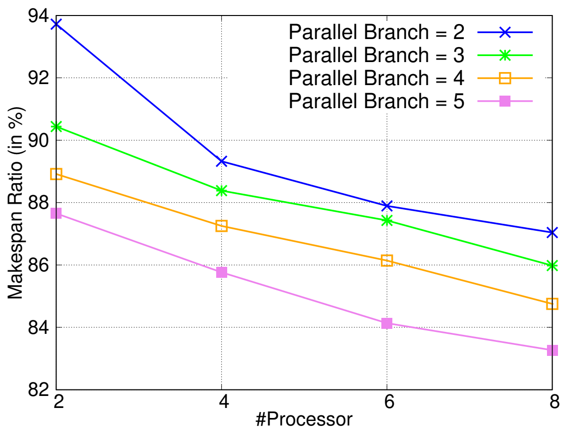
PTG	Shape Parameter	Task Nodes	Message Nodes	Total Nodes	ILP Version	Deadline Extension Ratio				
						1	1.25	1.5	1.75	2
Gaussian Elimination	$\chi = 3$	5	5	10	ILP-ETR	0.94s	20.1s	2m:19s	7m:27s	21m:9s
					ILP-NC	0.7s	0.65s	0.49s	0.68s	0.73s
	$\chi = 4$	9	11	20	ILP-ETR	3m:8s	32m:30s	2h:43m:9s	7h:33m:58s	15h:4m:10s
					ILP-NC	4.03s	3.94s	4.08s	4.02s	3.99s
Epigeneomics	$\gamma = 3$	16	17	33	ILP-ETR	29m:24s	4h:21m:9s	19h:39m:38s	@	@
					ILP-NC	56.43s	56.58s	55.33s	55.54s	55.76s
	$\gamma = 4$	20	22	42	ILP-ETR	20h:11m:19s	@	@	@	@
					ILP-NC	2m:50s	2m:50s	2m:50s	2m:49s	2m:51s
Laplace	$\varphi = 3$	9	12	21	ILP-ETR	4h:43m:30s	@	@	@	@
					ILP-NC	11.28s	11.48s	11.45s	11.34s	11.26s
	$\varphi = 4$	16	24	40	ILP-ETR	@	@	@	@	@
					ILP-NC	5m:50s	5m:50s	5m:51s	5m:52s	5m:50s
Stencil	$\lambda = 3$	9	14	23	ILP-ETR	11m:6s	21h:31m:29s	@	@	@
					ILP-NC	23.23s	23.06s	22.95s	22.97s	23.13s
	$\lambda = 4$	16	30	46	ILP-ETR	@	@	@	@	@
					ILP-NC	6h:40m:58s	6h:40m:6s	6h:39m:14s	6h:38m:24s	6h:37m:9s

Table 6.7: *Running time of ILP-ETR and ILP-NC. The symbol @ represents run-times greater than 24 hours*

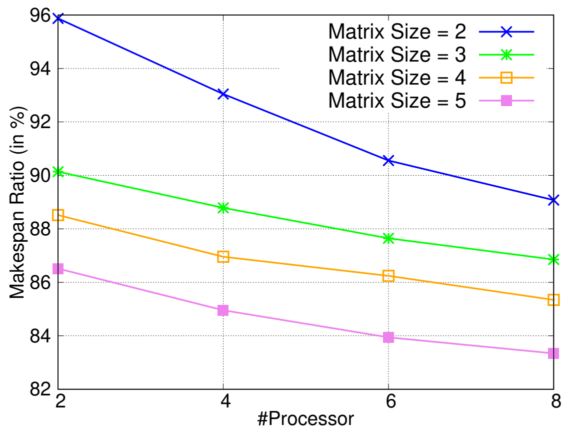
γ for (*Epigenomics*) from 2 to 5, matrix size φ (*Laplace*) between 2 and 5 and matrix size λ (*Stencil*) from 2 to 5, while keeping the number of buses (b) to be fixed at 2 and CCR at 1. We observe that for any given number of parallel branches (*Epigenomics*) or fixed matrix size (*Gaussian Elimination, Laplace, Stencil*), the *makespan ratio* becomes lower with increase in the number of processors. This may be attributed to the fact that as the number of processors increase, CC-TMS gains more flexibility to assign tasks on different processors in the platform. However, this does not lead to the reduction in the number of message nodes and ultimately pulling down the performance of CC-TMS. In comparison, the ILP remains mostly unaffected by changes in the number of processors. For example, in *Gaussian Elimination* (Figure 6.7a) with *Matrix Size* $= 3$, the *Makespan Ratio* for $p = 2$ and $p = 8$ are ~96% and ~91%, respectively. It can also be noted that as the number of nodes becomes higher, the heterogeneity in execution times and communication times in the PTG also increases. This results in the poorer performance of CC-TMS as compared to the ILP scheme.



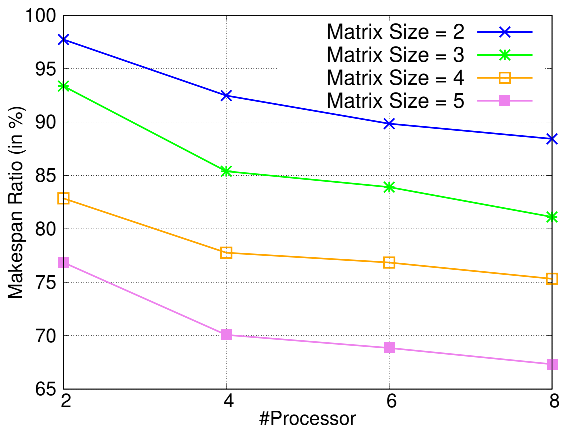
(a) *Gaussian Elimination*



(b) *Epigenomics*



(c) *Laplace*

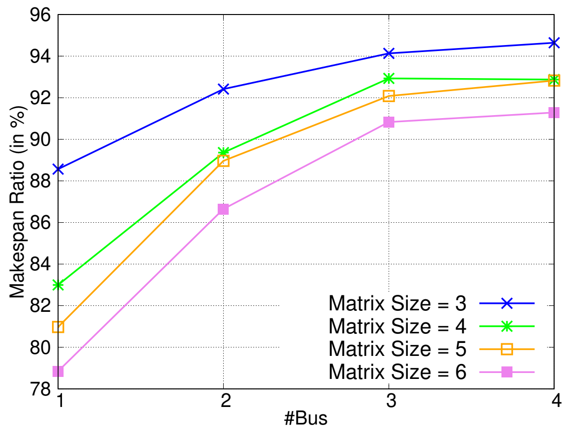


(d) *Stencil*

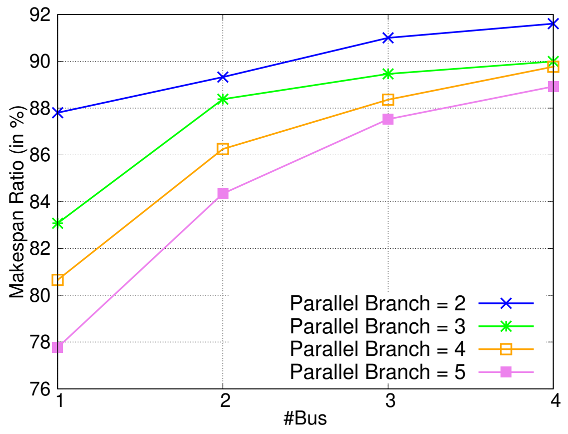
Figure 6.7: *Effect of varying processors*

Experiment-3: Varying the number of buses: For this experiment, the number of buses (b) has been increased from 1 to 4, matrix size χ (*Gaussian Elimination*) is varied between 3 and 6, number of parallel branches γ (*Epigenomics*) between 2 and 5, matrix size φ (*Laplace*) between 2 and 5 and matrix size λ (*Stencil*) from 2 to 5, while keeping the number of processors (p) fixed to 4, and CCR to 1. Figure 6.8 illustrates the obtained results for this experiment. We observe that for any specific matrix size

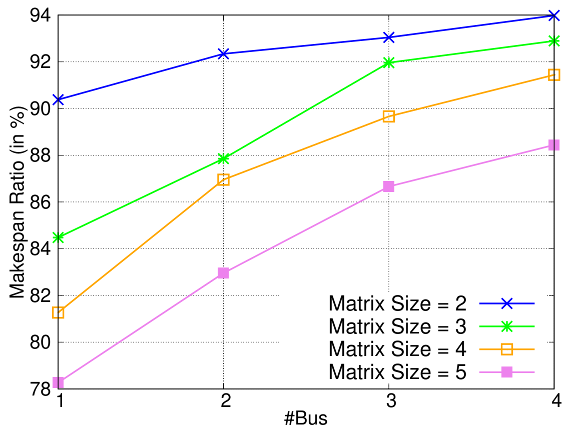
(*Gaussian Elimination, Stencil, Laplace*) or number of parallel branches (*Epigenomics*), the *Makespan Ratio* increases with increase in the number of buses. Decrease in resource contention w.r.t. message transmission which occurs as the number of buses increase, may be considered to be the cause for this phenomena. The result is a consequent increase in overall *Makespan Ratio*. As example, for *Gaussian Elimination* (Figure 6.8a), with *Matrix Size* = 3, the *Makespan Ratios* are ∼89% and ∼95% for $b = 1$ and $b = 4$, respectively.



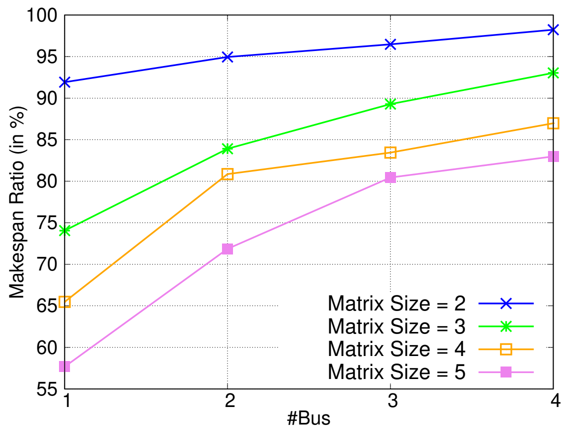
(a) *Gaussian Elimination*



(b) *Epigenomics*



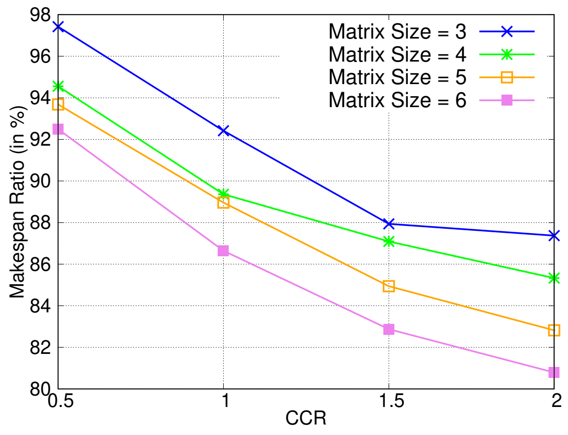
(c) *Laplace*



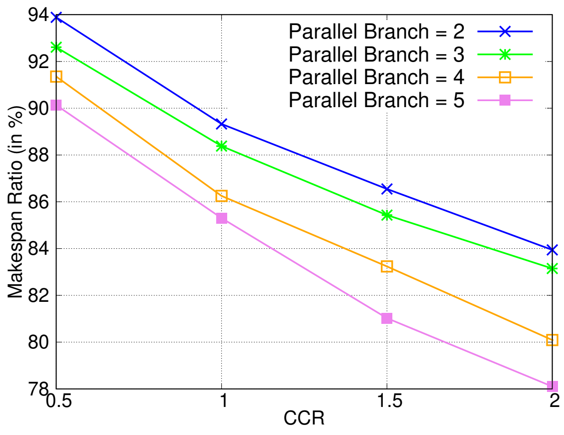
(d) *Stencil*

Figure 6.8: *Effect of varying buses*

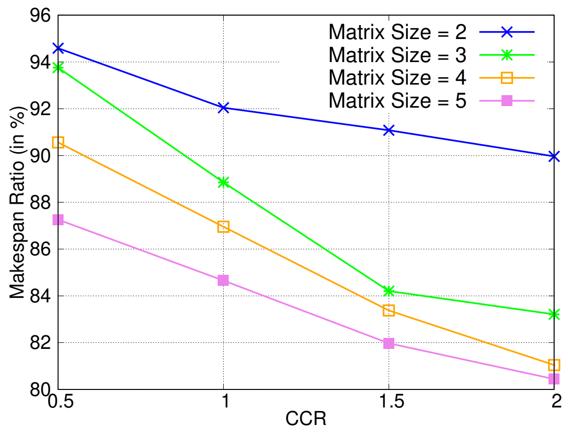
Experiment-4: Varying CCR: We vary CCR between 0.5 and 2, matrix size χ (*Gaussian Elimination*) between 3 and 6, number of parallel branches γ (*Epigenomics*) between 2 and 5, matrix size φ (*Laplace*) between 2 and 5 and matrix size λ (*Stencil*) between 2 and 5, while fixing the number of processors (p) and buses (b) to 4 and 2, respectively. Figure 6.9 shows the results. From the figure, we observe that for any particular matrix size (*Gaussian Elimination, Stencil, Laplace*) or number of parallel branches (*Epigenomics*), the *Makespan Ratio* decreases as CCR becomes higher. This is because, with the increase in CCR, the overall contention for message transmission increases, resulting in an overall decrease in the *Makespan Ratio*. As example, for *Gaussian Elimination* (Figure 6.9a) with *Matrix Size* = 3, the *Makespan Ratios* are ∼98% and ∼88% for $CCR = 0.5$ and $CCR = 2$, respectively.



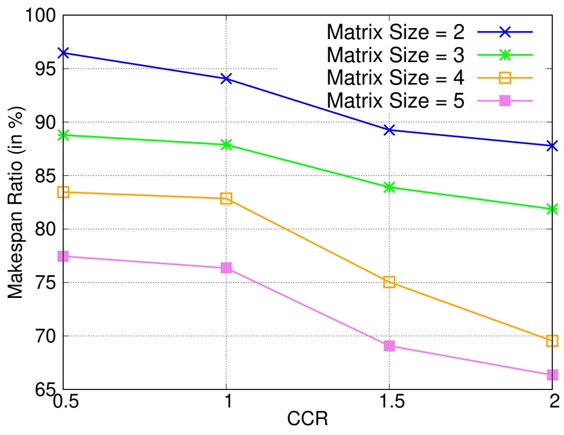
(a) *Gaussian Elimination*



(b) *Epigenomics*



(c) *Laplace*



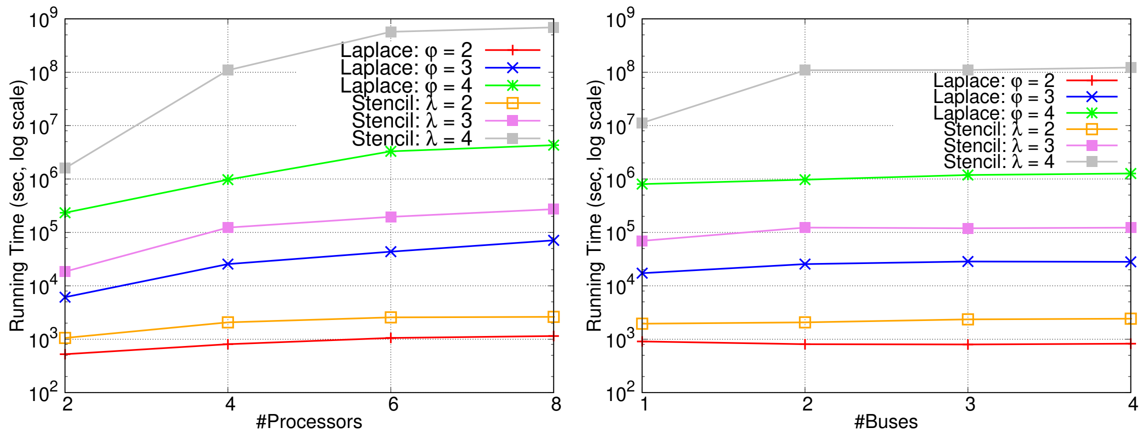
(d) *Stencil*

Figure 6.9: *Effect of varying CCR*

Experiment-5: Running time comparison: The performance of the designed algorithms with respect to incurred running times have been evaluated by measuring their actual average run-times over various data sets. Then, we have determined the *speedup* achieved by CC-TMS over ILP-NC. That is,

$$speedup = \frac{\text{Running time of ILP-NC}}{\text{Running time of CC-TMS}} \tag{6.39}$$

In Table 6.8, we present the speedups achieved by CC-TMS over ILP-NC, varying the number of processors (p) between 2 and 8, buses (b) between 1 and 4, matrix size χ (*Gaussian Elimination*) between 3 and 5, number of parallel branches γ (*Epigenomics*) from 2 to 4, matrix size φ (*Laplace*) from 2 to 4 and matrix size λ (*Stencil*) from 2 to 4, while keeping CCR fixed to 1. The actual speedups are 10^5 times the values (say, x) shown in the table (that is, actual speedup $= x \times 10^5$ times). Figure 6.10 shows the graphical representation of the obtained results for *Laplace* and *Stencil* PTGs. It may be seen that speedups increase with the number of nodes, processors and/or buses. The reason may be attributed to the high complexity of the ILP-NC solution along with its high sensitivity to the number of nodes, processors and buses. In comparison, the complexity of CC-TMS exhibits significantly lower sensitivity to the number of tasks, processors and buses.



(a) *Varying Processors* **(b)** *Varying Buses*

Figure 6.10: *Speedup of CC-TMS compared to ILP-NC*

Experiment-6: Comparison with related works: This experiment attempts to

PTG	Matrix Size	Speedup ($x \times 10^5$)							
		#Processors (#Buses = 2)				#Buses (#Processors = 4)			
		2	4	6	8	1	2	3	4
Gaussian Elimination	$\chi = 4$	0.05	0.11	0.19	0.23	0.08	0.11	0.12	0.12
	$\chi = 5$	0.29	1.32	1.67	1.86	1.45	1.32	1.2	1.13
	$\chi = 6$	3.48	10.15	20.83	24.44	11.08	10.15	10.16	12.64
Epigenomics	$\gamma = 2$	0.05	0.27	0.42	0.51	0.15	0.27	0.27	0.27
	$\gamma = 3$	0.61	1.51	1.85	2.42	1.24	1.51	1.38	1.37
	$\gamma = 4$	11.74	10.88	9.47	14.57	12.22	10.88	10.48	9.83
Laplace	$\varphi = 2$	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	$\varphi = 3$	0.06	0.26	0.44	0.71	0.17	0.26	0.29	0.28
	$\varphi = 4$	2.33	9.74	32.95	42.97	8.04	9.74	11.9	12.68
Stencil	$\lambda = 2$	0.01	0.02	0.03	0.03	0.02	0.02	0.02	0.02
	$\lambda = 3$	0.18	1.23	1.96	2.73	0.69	1.23	1.2	1.23
	$\lambda = 4$	15.93	1089.09	6702.85	5913.86	112.29	1089.09	1099.38	1223.74

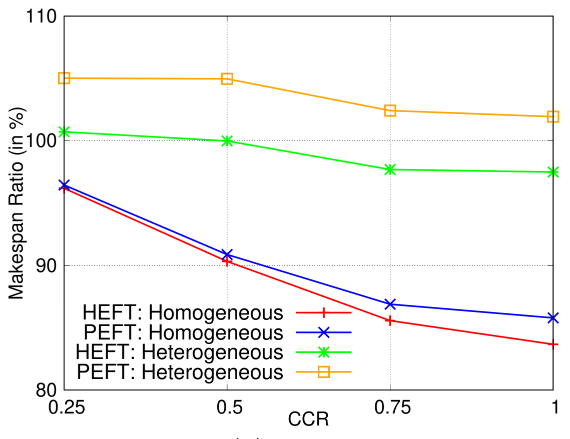
Table 6.8: *Running Time Comparison (Speedup of CC-TMS compared to ILP-NC)*

exhibit the relative performance of CC-TMS, particularly in relation to the use of shared buses in CC-TMS against dedicated communication channels in HEFT [6] and PEFT [1]. For this purpose, we have conducted experiments with two different benchmark PTGs namely, Laplace and Stencil, which significantly differ in terms of their demand for communication resources. Comparing the structures of these two benchmark PTGs, it may be observed that *the ratio of the number of message nodes to the number of task nodes* is higher for Stencil compared to Laplace. Thus, Stencil is more I/O bounded and significantly more intensive in terms of its demand for communication resources, compared to Laplace. In the experiment, we have fixed the matrix size for Laplace (φ) to 9 (n = 81, m = 144) and for Stencil (λ) to 8 (n = 64, m = 154). Execution times of each task node on the different processors are randomly chosen from a uniform distribution $[10, 30]$. CCR is varied from 0.25 to 1. We set the number of processors (p)

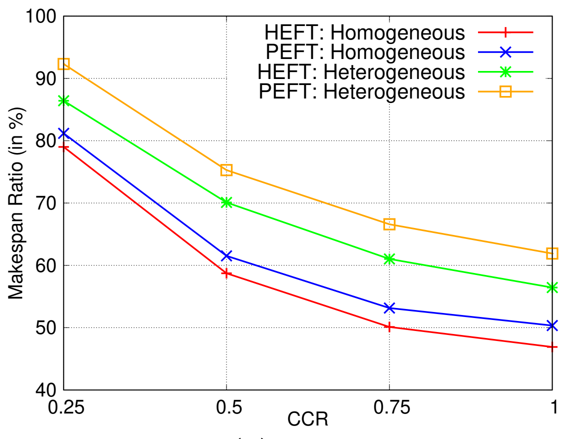
to 8, and the number of buses (b) to 4 (for CC-TMS).

While CC-TMS assumes a shared bus communication platform, PEFT and HEFT are oriented towards fully-connected processing platforms. We have conducted experiments assuming the communication platform to be *homogeneous* as well as *heterogeneous*. All the communication mediums (shared buses for CC-TMS, dedicated channels for HEFT and PEFT) are assumed to have the same bandwidth for the homogeneous case. On the other hand, bandwidths of the communication mediums have been randomly chosen from a uniform distribution $[0.9, 1.1]$ when a heterogeneous communication platform is considered. Finally, message sizes are appropriately derived from the generated execution times, such that the desired CCR is achieved for a given average communication bandwidth. We have used *Makespan Ratio* as the metric for comparison:

$$Makespan\ Ratio = \frac{HEFT\ or\ PEFT\ Makespan}{CC - TMS\ Makespan}$$



(a) *Laplace*

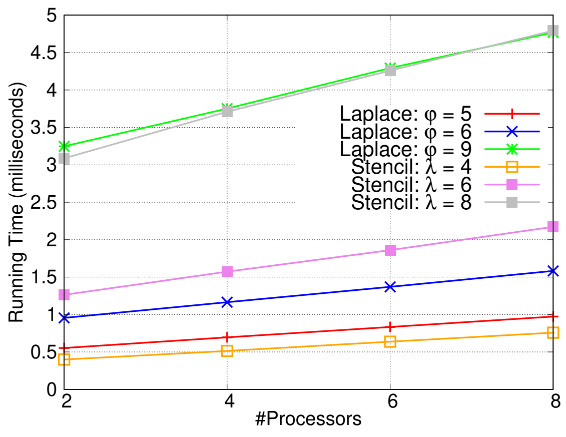


(b) *Stencil*

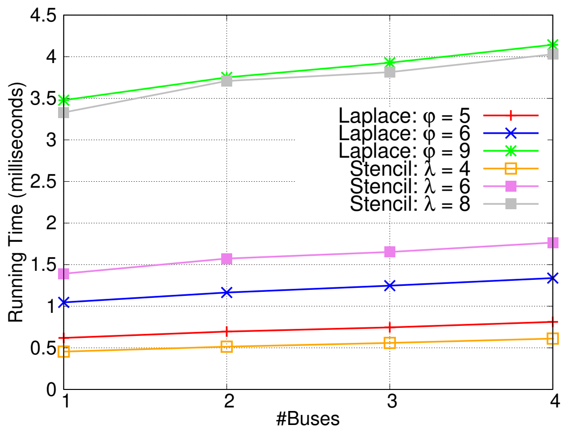
Figure 6.11: *Comparison among CC-TMS, HEFT and PEFT*

Figures 6.11a and 6.11b present the experimental results depicting the relative performance of CC-TMS on the Laplace and Stencil PTGs, respectively. Although bandwidths for the shared buses in CC-TMS and the communication channels in HEFT/PEFT have been generated assuming the same degree of heterogeneity, CC-TMS comes with the flexibility of being able to choose a bus which has better affinity for a message (relatively lower transmission time), because each bus is connected to all processors. In comparison, HEFT/PEFT with its fully interconnected platform, is forced to transmit a message from a source to a destination processor, through the dedicated communication channel between them, irrespective of the affinity of this message to the channel. Due to this flexibility of choosing a bus that takes less transmission time, CC-TMS performs relatively better especially when contention for the shared buses is comparatively low. This may be observed to happen with lower CCR values for both the Laplace and Stencil PTGs in Figures 6.11a and 6.11b. In addition, due to the relatively higher number of edges in Stencil compared to Laplace, inter-task dependencies are higher in Stencil than Laplace. Consequently, the average number of ready to execute tasks at any given time is higher for Laplace, allowing better utilization of processor resources. Due to this lower communication resource demand, CC-TMS is generally seen to perform significantly better with Laplace compared to Stencil. In fact, from the results obtained for Laplace (Figure 6.11a for the case when communication resources are heterogeneous, CC-TMS is seen to perform better than HEFT and at par with PEFT. However, when the communication resources (buses for CC-TMS, communication channels for HEFT and PEFT) are homogeneous, the advantage of CC-TMS derived through the ability to flexibly choose bus resources, has no effect. Consequently, relative performance of CC-TMS degrades for both Laplace and Stencil, as revealed in Figure 6.11. Again, due to its lower I/O boundedness, this degradation is less severe for Laplace (Figure 6.11a) than Stencil (Figure 6.11b).

Experiment-7: Scalability of CC-TMS: To show the scalability of CC-TMS, we have conducted an experiment with the Laplace and Stencil benchmark PTGs, where we have plotted variation in the run-times of CC-TMS as the number of nodes in the PTGs is varied from ∼50 to ∼200. For Laplace, we have varied the matrix size (φ) from 5 (n = 25, m = 40) to 9 (n = 81, m = 144). Similarly, matrix size (λ) for Stencil has been varied between 4 (n = 16, m = 30) and 8 (n = 64, m = 154). CCR has been fixed to 1. Figure 6.12a shows the results for run-time of CC-TMS as the number of processors (p) is varied from 2 to 8, while fixing the number of buses (b) at 2. It can be seen that the running-time of CC-TMS increases monotonically as the number of nodes increases. In Figure 6.12b, we show the variation in run-times as the number of buses is varied from 1 to 4, while keeping the number of processor fixed to 4. From the obtained results, we see that run-times of CC-TMS increase at a very moderate pace with respect to increase in the number of PTG nodes, processors, or buses. It may be noted that even for large PTGs containing about 200 nodes, solution generation times of CC-TMS remain below 5 ms. This points to the generic efficacy of CC-TMS in terms of scalability and indicates to its applicability to problem scenarios with PTGs consisting of a large number of nodes.



(a) *Varying processors; $b = 2$*



(b) *Varying buses; $p = 4$*

Figure 6.12: *Scalability of CC-TMS (running time in ms)*

6.7 Case Study: Traction Controller

To exhibit the practical applicability of the presented strategies to actual designs, we discuss a case study using a *Traction Controller* (TC) application present in automotive systems. TC helps in actively stabilizing an automobile so that it can continue in its stipulated path even when road conditions are slippery [5]. Figure 6.13a depicts the block diagram of TC as adopted from [5]. The corresponding PTG representation which consists of 12 task nodes $\{T_0, T_1, \ldots, T_{11}\}$ and 17 message nodes $\{M_1, M_2, \ldots, M_{17}\}$, is

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
P_1	200	200	200	200	150	300	175	400	150	200
P_2	185	185	185	185	160	320	150	375	175	220

Table 6.9: *Execution times of task nodes (in μs)*

shown in Figure 6.13b. Here, in order to adhere to the assumptions stated in Section 6.1, we have added, (i) two dummy task nodes T_0 and T_{11} as source and sink nodes, respectively, (ii) eight dummy message nodes M_{10} to M_{17}. For the purpose of this case study, the PTG is assumed to be executed on a two processor ($P = \{P_1, P_2\}$) heterogeneous distributed platform interconnected via two heterogeneous buses ($B = \{B_1, B_2\}$). Table 6.9 lists the task execution times associated with the processors in P. Table 6.10 lists the communication times of the message nodes on the different buses in B. The execution/communication times of the dummy task/message nodes on all processors/buses are set to 0.

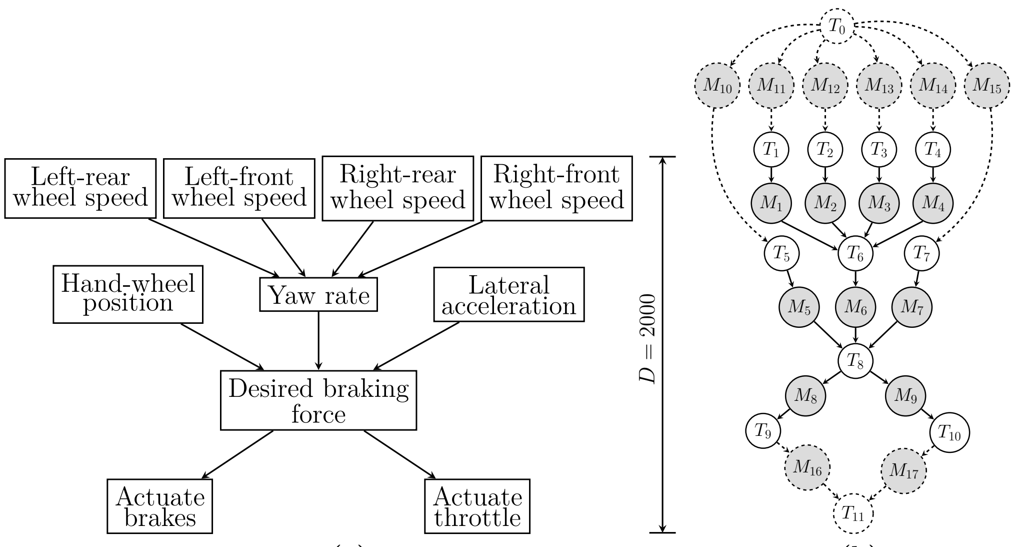


Figure 6.13: *Traction Control application's (a) Block Diagram [5], (b) PTG representation*

We have employed ILP-NC and CC-TMS to generate schedules for the PTG of TC with the objective of minimizing makespan. We summarize below the important obser-

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9
B_1	475	475	475	475	1100	275	1025	650	650
B_2	400	400	400	400	1200	280	550	630	630

Table 6.10: *Transmission times of message nodes (in μs)*

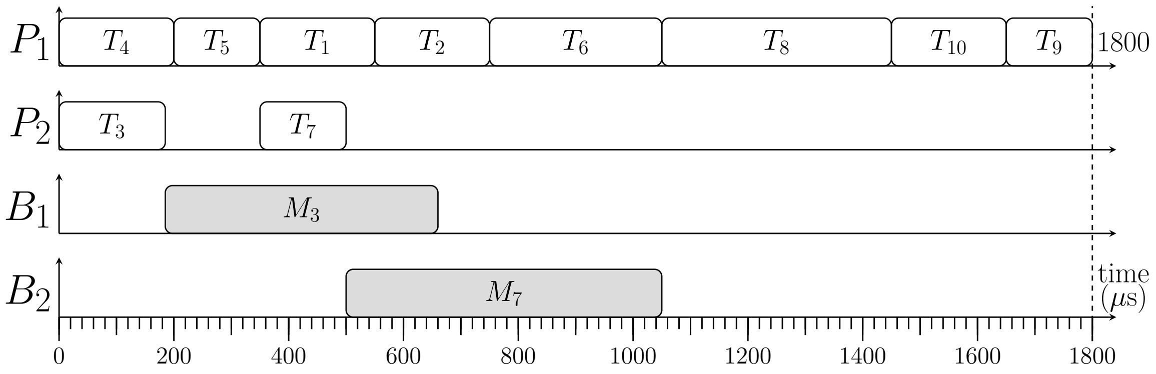


Figure 6.14: *The schedule for the PTG (Figure 6.13b) using ILP-NC*

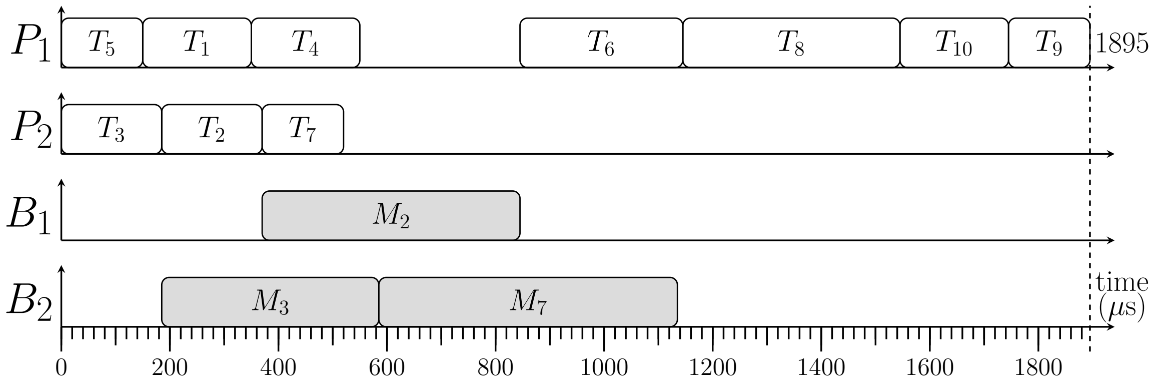


Figure 6.15: *The schedule for the PTG (Figure 6.13b) using CC-TMS*

vations associated with the obtained schedules for ILP-NC (Figure 6.14) and CC-TMS (Figure 6.15):

- *Constraint Satisfaction for ILP*: From the ILP-NC schedule (Figure 6.14), we can observe that, (i) on any given resource (processor/bus), no two tasks/messages commence execution/transmission at the same time, (ii) resource constraints have always been satisfied, (iii) constraints related to inter-node dependencies have been adhered to. For example, task T_1 commences execution on processor P_1 at time

350 μs. While being processed on P_1, no task barring T_1 executes on P_1 (thus, satisfying resource usage constraint). As both tasks T_2 and T_6 are allocated on to the same processor P_1, no communication overhead is incurred for transmission of the output of T_2 to T_6.

- *Heterogeneity Modeling*: Each PTG node consumes an appropriate execution/ transmission time according to the processor/bus allocated to it. As example, in Figure 6.14, T_1 consumes 200 μs as it is scheduled on P_1; M_3 takes 475 μs as it is transmitted over B_1 (commensurate with their execution and transmission demands as specified in Tables 6.9 and 6.10).

- Implementation of ILP-NC using the CPLEX optimizer generates 793 constraints and takes ∼2 secs to produce the solution. The corresponding schedule (in Figure 6.14) has a makespan of 1800 μs. On the other hand, the schedule computed using CC-TMS (Figure 6.15) takes 50 μs to generate the solution which has the makespan of 1895 μs. Similar to the result trends obtained in the experiments section, we observe that the optimal solution delivers lower makespans (CC-TMS makesapn − ILP makespan = 95). However, the run-time overhead associated with the ILP is about $\sim 10^5$ times higher than CC-TMS. Thus, while the ILP produces very efficient solutions which may be critical for resource-constrained embedded systems, the CC-TMS algorithm is much more scalable and faster producing reasonably good solutions which may be important when quick design iterations are required.

6.8 Summary

This chapter addressed the problem of scheduling PTGs to be executed on a heterogeneous distributed system interconnected via shared buses. It proposes two ILP based solution ILP-ETR and ILP-NC to compute optimal schedules. Though, ILP-NC shows an appreciable improvement in terms of scalability compared to ILP-ETR, its run time is high and sensitive to the number of nodes and the number of resources. Appreciating the necessity of a fast but efficient algorithm for the problem at hand, especially for situations when quick solutions are needed at design-time or run-time, we have proposed a heuristic namely, CC-TMS. Extensive experiments have been carried-out using benchmark PTGs for performance evaluation of the proposed strategies. The obtained results show that *the heuristic scheme (CC-TMS) is* $\sim 10^5$ *times faster than the ILP based optimal strategy ILP-NC*. Finally, we presented a case study using a real-world *Traction Controller* (TC) application. The next chapter endeavours towards the design of heterogeneous processor-shared bus co-scheduling strategies for a given set of independent periodic applications, each of which is modelled as a PTG.

Chapter 7

Scheduling Multiple Independent PTG Applications on Shared-Bus Platform

Works done in Chapters 4, 5 and 6 deal with the co-scheduling of a single task graph application. In this chapter, we endeavour towards the design of heterogeneous processor-shared bus co-scheduling strategies for a given set of independent periodic applications, each of which is modelled as a PTG. In particular, we have developed an ILP based optimal and heuristic strategy for the mentioned system model, whose objective is to minimize system level dynamic energy dissipation. To achieve energy savings, the processors in the system are assumed to be DVFS enabled and thus, the operating frequencies of these processors can be dynamically reconfigured to a discrete set of alternative voltage/frequency-levels at run-time. However, the ILP based optimal scheme called ILP-ES is associated with very high computational complexity and is not scalable even for small problem sizes. Therefore, we propose an efficient but low-overhead heuristic strategy called SAFLA which consumes drastically lower time and space complexities while generating good and acceptable solutions. Experimental results show that SAFLA is an effective scheduling scheme and delivers handsome savings in terms of lower energy consumption in most practical scenarios. We conclude the chapter after presenting a case study using *Electric Power Steering* (EPS), *Adaptive Cruise Controller* (ACC) and *Traction Controller* (TC) applications.

7.1 Models and Terminologies

This section introduces the system model along with associated assumptions, the power and energy model, the problem statement, and also presents an example system which illustrates the discussed models.

System Model: A *distributed heterogeneous multiprocessor system* has been considered. The system consists of p heterogeneous processors $P = \{P_1, P_2, \ldots, P_p\}$ connected through b heterogeneous shared buses $B = \{B_1, B_2, \ldots, B_b\}$. Each processor P_r is logically connected to all the buses. The pictorial representation of the platform is shown in Figure 7.1. The processors are DVFS enabled; that is, a processor P_r can execute at a discrete set of alternative (voltage/frequency) levels $L_r = \{1, 2, \ldots, |L_r|\}$, with each level $l \in [1, |L_r|]$ being associated with operating voltage V_{rl} and frequency f_{rl}. Here, f_{r1} (V_{r1}) and $f_{r|L_r|}$ ($V_{r|L_r|}$) represents the maximum and minimum operating frequencies (voltages) of P_r, respectively. Table 7.1 shows the voltage/frequency-levels corresponding to a system of three heterogeneous processors (P_1: Intel Pentium M, P_2: AMD Athlon-64, P_3: AMD Opteron 2218) [8].

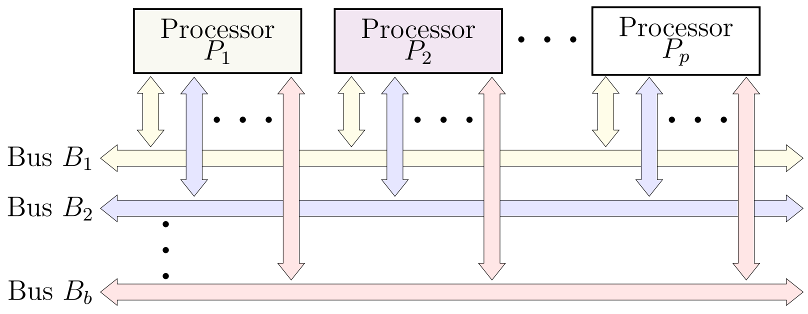


Figure 7.1: *Platform Model*

This work considers a set $\mathcal{G}$ of periodic applications $\{G^1, G^2, \ldots, G^N\}$, where each application $G^g \in \mathcal{G}$ is denoted by a *Precedence-constrained Task Graph* (PTG) as depicted in Figure 7.2. PTG G^g is described through a two-tuple $G^g = (\mathcal{V}^g, \mathcal{E}^g)$ where,

- $\mathcal{V}^g = \{T_1^g, T_2^g, \ldots, T_{n^g}^g, M_1^g, M_2^g, \ldots, M_{m^g}^g\}$ denotes $(n^g + m^g)$ nodes, where $T^g = \{T_1^g, T_2^g, \ldots, T_{n^g}^g\}$ is a set of n^g tasks, and $M^g = \{M_1^g, M_2^g, \ldots, M_{m^g}^g\}$ denotes m^g messages capturing inter-task data transmission demands.

Level	Intel Pentium M		AMD Athlon-64		AMD Opteron 2218	
	Voltage	Frequency	Voltage	Frequency	Voltage	Frequency
1	1.5	2	1.484	1.4	1.3	2.6
2	1.4	1.8	1.463	1.2	1.25	2.4
3	1.3	1.6	1.308	1	1.2	2.2
4	1.2	1.4	1.18	0.8	1.15	2
5	1.1	1.2	0.956	0.6	1.1	1.8
6	1	1			1.05	1

Table 7.1: *Sample voltage (volt) / frequency (GHz) pairs [8]*

- $\mathcal{E}^g \subseteq \mathcal{V}^g \times \mathcal{V}^g$ denotes the set of edges which describe *dependency-constraints* among nodes in $\mathcal{V}^g$.

- Each task T_i^g may potentially execute on any processor and at any voltage/frequency-level on that processor. The *worst-case execution time* (WCET) of a task T_i^g on a given processor P_r at voltage/frequency-level l is represented as e_{irl}^g.

- The WCET of T_i^g on P_r at its minimum (base) voltage/frequency-level is represented as e_{ir1}^g. Given e_{ir1}^g, execution times of T_i^g on P_r at any other voltage/frequency-level (except minimum level) is computed as $e_{irl}^g = \left\lceil \frac{e_{ir1}^g \times f_{r1}}{f_{rl}} \right\rceil$, where f_{r1} represents the maximum execution frequency, while f_{rl} denotes the current operating frequency of P_r.

- Each message $M_k^g \in M^g$ may possibly be transmitted through any bus $B_r \in B$ with the distinct communication time associated with this transmission being c_{kr}^g.

- The processors being heterogeneous, the *WCETs* of a task on two distinct processors are completely unrelated. Similarly, the data transmission times of a message on two distinct buses are completely unrelated.

- The WCETs of a task T_i^g on P_r is fixed to ∞ for all voltage/frequency-levels, to model the case in which the execution of T_i^g is infeasible on P_r. Similarly, the communication times of a message M_k^g is set to ∞, if the transmission of M_k^g is infeasible on B_r.

- D^g is the *implicit deadline* associated with the periodic application G^g.

Assumptions:

1. Each PTG has one start (source) and one end (sink) node. Both start and end nodes are tasks.
2. The parents (excluding source nodes) and children (excluding sink nodes) of a task T_i^g are message nodes.
3. A message node M_k^g has only one parent and one child node, and both are task nodes.
4. If the parent and child tasks of a message M_k^g are assigned on the same processor, data transmission time of M_k^g becomes negligible; i.e., $\forall B_r \in B$, $c_{kr}^g = 0$.

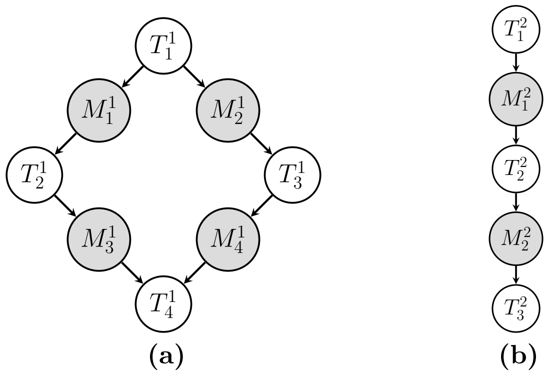


Figure 7.2: *Example of PTGs; (a) PTG G^1, Period $D^1 = 20$; (b) PTG G^2, Period $D^2 = 10$*

Given a set of persistent periodic applications $\{G^1, G^2, \ldots, G^N\}$ with their implicit deadlines (i.e., periods) $\{D^1, D^2, \ldots, D^N\}$, the *hyperperiod* ($\mathcal{H}$) is determined as *Least Common Multiple* (LCM) of the application periods, i.e., $\mathcal{H} = LCM(D^1, D^2, \ldots, D^N)$.

Processor	PTG G^1				PTG G^2		
	T_1^1	T_2^1	T_3^1	T_4^1	T_1^2	T_2^2	T_3^2
P_1	5	6	5	3	1	2	3
P_2	7	4	4	5	2	2	4
P_3	8	5	2	6	4	6	3

Table 7.2: *Execution times of tasks at processors' maximum voltage/frequency*

Bus	PTG G^1				PTG G^2	
	M_1^1	M_2^1	M_3^1	M_4^1	M_1^2	M_2^2
B_1	1	1	1	1	1	1
B_2	2	2	2	2	2	2

Table 7.3: *Communication times of message nodes*

The number of iterations/instances of an application PTG G^g within the hyperperiod $\mathcal{H}$ is obtained as, $I^g = \mathcal{H}/D^g$. So, G^g has I^g instances within $\mathcal{H}$ denoted as: $\{G^{g1}, G^{g2}, \ldots, G^{gI^g}\}$.

A PTG G^{gq} corresponding to any application instance is described through a two tuple $(\mathcal{V}^{gq}, \mathcal{E}^{gq})$, where $\mathcal{V}^{gq} = \{\mathcal{V}_1^{gq}, \mathcal{V}_2^{gq}, \ldots, \mathcal{V}_{n^g+m^g}^{gq}\}$ is the set of task and message nodes in G^{gq}, and $\mathcal{E}^{gq} \subseteq \mathcal{V}^{gq} \times \mathcal{V}^{gq}$ denotes the set of edges which describe *dependency-constraints* among nodes in $\mathcal{V}^{gq}$. The nodes $\mathcal{V}_1^{gq}, \mathcal{V}_2^{gq}, \ldots, \mathcal{V}_{n^g}^{gq}$ represent the n^g task nodes of application G^g, while $\mathcal{V}_{n^g+1}^{gq}, \mathcal{V}_{n^g+2}^{gq}, \ldots, \mathcal{V}_{n^g+m^g}^{gq}$ are the m^g message nodes of G^g. The n^g task nodes for the q^{th} instance G^{gq} (of G^g) is denoted by T_1^{gq}, T_2^{gq}, $\ldots$, $T_{n^g}^{gq}$. Similarly, the m^g message nodes of G^{gq} are represented as M_1^{gq}, M_2^{gq}, $\ldots$, $M_{m^g}^{gq}$. Thus, $\mathcal{V}_1^{gq} = T_1^{gq}$, $\mathcal{V}_2^{gq} = T_2^{gq}$, $\ldots$, $\mathcal{V}_{n^g}^{gq} = T_{n^g}^{gq}$, $\mathcal{V}_{n^g+1}^{gq} = M_1^{gq}$, $\mathcal{V}_{n^g+2}^{gq} = M_2^{gq}$, $\ldots$, $\mathcal{V}_{n^g+m^g}^{gq} = M_{m^g}^{gq}$. The preceding and succeeding message nodes of T_i^{gq} are denoted via the notations $pred(T_i^{gq})$ and $succ(T_i^{gq})$, respectively. Similarly, we use the notations $pred(M_k^{gq})$ and $succ(M_k^{gq})$ to represent the preceding and succeeding task nodes of M_k^{gq}.

It may be noted that the set of PTGs being persistent, a schedule for all instances of all PTGs in $\mathcal{H}$ will repeat every hyperperiod. It is therefore sufficient to focus on the generation of an efficient schedule for one hyperperiod, as is the objective of the algorithms proposed herein.

Power and Energy Models: Three different categories of power consumption are typically prominent in CMOS circuits namely, *dynamic*, *static*, and *short* circuit power [61]. Among these, dynamic power is the major source of power dissipation. It is defined as,

$$\mathcal{P} = a\mathcal{C}V^2 f \tag{7.1}$$

where f denotes the clock frequency, V represents the supply voltage, $\mathcal{C}$ is a constant which denotes the loading capacitance, and a is a constant indicating activity factor. In this work, we attempt to minimize dynamic power in line with the approach followed in other related works [8], [62], [63], [64]. The energy dissipated during the execution of a task T_i^g on processor P_r at level l (V_{rl}, f_{rl}) is calculated as,

$$E(T_i^g, r, l) = \alpha \times V_{rl}^2 \times f_{rl} \times \left\lceil \frac{e_{ir1}^g \times f_{r1}}{f_{rl}} \right\rceil \tag{7.2}$$

where V_{rl}, f_{rl} and f_{r1} are the supply voltage, execution frequency and maximum execution frequency of the processor P_r on which T_i^g executes. Here, $\alpha = a\mathcal{C}$. The difference in the energy consumption between two levels l_1 and l_2 ($l_1 < l_2$) can be computed as follows:

$$E'(T_i^g, r, l_1, l_2) = E(T_i^g, r, l_1) - E(T_i^g, r, l_2) \tag{7.3}$$

Example: Figure 7.2a depicts a sample PTG G^1 consisting of 8 nodes, with implicit deadline $D^1 = 20$ units. Among them, $\{T_1^1, T_2^1, T_3^1, T_4^1\}$ are tasks and $\{M_1^1, M_2^1, M_3^1, M_4^1\}$ are messages. Hence, $n^1 = m^1 = 4$. Similarly, Figure 7.2b shows a PTG G^2 which consists of 5 nodes, with implicit deadline $D^2 = 10$ units. We consider a sample platform model with three heterogeneous processors P_1, P_2, and P_3 interconnected via two shared buses B_1 and B_2. Here processor P_1, P_2 and P_3 represent Intel Pentium M, AMD Athlon and AMD Opteron 2218, respectively, and have voltage/frequency at different levels as shown in Table 7.1. In Table 7.2, we show the WCETs of the task nodes when they execute at the maximum voltage/frequency ($l = 1$) on their assigned processors. An entry $e_{1,1,1}^1 = 5$ in this table specifies that task T_1^1 takes 5 time units to complete its execution on processor P_1 at $l = 1$. Similarly, Table 7.3 shows data transmission times of the message nodes. An entry say $c^1_{1,1} = 1$ in this table denotes that message M^1_1 takes one time unit for transmission over bus B_1. Further, it may be noted that $\mathcal{H} = LCM(D^1, D^2) = 20$. Hence, the number of instances of G^1 and G^2 are $I^1 = 1$ and $I^2 = 2$, respectively.

Problem Statement: Given a set of periodic applications (PTGs) $\mathcal{G} = \{G^1, G^2, \ldots, G^N\}$ with their end-to-end deadlines $(D^1, D^2, \ldots, D^N)$ and a set of p heterogeneous processors connected through b heterogeneous shared buses, determine for the task and message nodes of all instances of each PTG in a hyperperiod $\mathcal{H}$, *(i) task-to-processor assignments, (ii) processor level (voltages/frequencies) for each such task assignment, (iii) message-to-bus assignment, (iv) start times for all task and messages*, such that the *aggregate energy consumption is minimized*, while meeting constraints related to resource, deadline, and precedence.

7.2 Earliest/Latest Start Times for PTG Nodes

The Earliest and Latest start times computation technique is different from earlier chapters because here we need to consider concurrent executions of multiple PTGs.

Let t^{gq}_{is} and t^{gq}_{il} be the *earliest* and *latest* start times at which task T^{gq}_i of PTG G^g at its q^{th} iteration may start execution. It may be noted that a task may potentially be scheduled on any processor and at any available frequency. In addition, message node M^{gq}_k becomes invalid when its predecessor and successor task nodes T^{gq}_i and T^{gq}_j are scheduled on the same processor. Hence, in order to get all possible valid ranges of start times of task nodes, we, (i) ignore message nodes (M^{gq}_k) in the PTG and assume directed edges between the predecessor (T^{gq}_i) and successor (T^{gq}_j) task nodes of each message node, (ii) consider the least execution times (e^g_{ir1}) for each task T^{gq}_i in the PTG G^g. The t^{gq}_{is} (*As Soon As Possible* (ASAP) time) and t^{gq}_{il} (*As Late As Possible* (ALAP) time) values for task nodes are computed as follows.

ASAP/ALAP computation procedure for task nodes:

The ASAP of task nodes is computed as follows:
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$

1. $\forall T_i^{gq} \mid indeg(T_i^{gq}) = 0$, set ASAP time of T_i^{gq} as,

$$t_{is}^{gq} = 1 + D^g * (q - 1)$$

2. ASAP times of the remaining task nodes (except T_i^{gq}, where $indeg(T_i^{gq}) = 0$) are recursively determined (downward) as follows:

$$t_{is}^{gq} = \max_{T_j^{gq} \in predT(T_i^{gq})} (t_{js}^{gq} + \min_{r \in [1,p]} e_{jr1}^{g})$$

where, $predT(T_i^{gq})$ is the set of predecessors of task node T_i^{gq}.

The ALAP of task nodes is computed as follows:
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$

1. $\forall T_i^{gq} \mid outdeg(T_i^{gq}) = 0$, set ALAP time of T_i^{gq} as,

$$t_{il}^{gq} = (D^g * q) - \min_{r \in [1,p]} e_{ir1}^{g} + 1$$

2. ALAP times of remaining task nodes (except T_i^{gq}, where $outdeg(T_i^{gq}) = 0$) are recursively determined (upward) as follows:

$$t_{il}^{gq} = \min_{T_j^{gq} \in succ(T_i^{gq})} (t_{jl}^{gq} - \min_{r \in [1,p]} e_{ir1}^{g})$$

where, $succ(T_i^{gq})$ is the set of successors of task node T_i^{gq}.

ASAP/ALAP computation procedure for message nodes:

1. ASAP times of message node M_k^{gq} of PTGs at each iteration is defined as:
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$

$$t_{ks}^{gq} = t_{is}^{gq} + \min_{r \in [1,p]} e_{ir1}^{g}$$

where, T_i^{gq} is the predecessor task node of M_k^{gq}.

2. ALAP times of node M_k^{gq} is defined as follows:
   $\forall g \in [1, N]$, $\forall q \in [1, I^g]$
   $$t_{kl}^{gq} = t_{js}^{gq} - \min_{r \in [1,b]} c_{kr}^{g}$$
   where, T_j^{gq} is the successor task node of M_k^{gq}.

	PTG G^1				PTG G^2					
	Iteration-1				Iteration-1			Iteration-2		
	$T_1^{1,1}$	$T_2^{1,1}$	$T_3^{1,1}$	$T_4^{1,1}$	$T_1^{2,1}$	$T_2^{2,1}$	$T_3^{2,1}$	$T_1^{2,2}$	$T_2^{2,2}$	$T_3^{2,2}$
ASAP	1	6	6	10	1	2	4	11	12	14
ALAP	9	14	16	18	5	6	8	15	16	18

Table 7.4: *ASAP and ALAP times of task nodes in PTGs G^1 and G^2 (Figure 7.2, Table 7.2 & Table 7.3)*

	PTG G^1				PTG G^2			
	Iteration-1				Iteration-1		Iteration-2	
	$M_1^{1,1}$	$M_2^{1,1}$	$M_3^{1,1}$	$M_4^{1,1}$	$M_1^{2,1}$	$M_2^{2,1}$	$M_1^{2,2}$	$M_2^{2,2}$
ASAP	6	6	10	10	2	4	12	14
ALAP	13	15	17	17	5	7	15	17

Table 7.5: *ASAP and ALAP times of message nodes in PTGs G^1 and G^2 (shown in Figure 7.2, Table 7.2 & Table 7.3)*

Example: Let us consider PTGs G^1 and G^2 (in Figure 7.2). Table 7.4 shows the ASAP and ALAP times corresponding to the task nodes in G^1 and G^2 for all instances of their executions within the hyperperiod. For example, ASAP and ALAP times of task node $T_1^{1,1}$ in PTG G^1 (for the first instance) are 1, and 9, respectively. Similarly, Table 7.5 shows the ASAP and ALAP times corresponding to each message node in G^1 and G^2. □

Next, we present an ILP based strategy namely, ILP-ES. The design philosophy of ILP-ES is similar to the formulations of ILP-SATC (Chapter 4, Section 4.3) and ILP-ETR (Chapter 6, Section 6.3), respectively. However, the objective functions and many of the constraints in the ILP presented in this chapter require certain modifications with respect to those in Chapters 4 and 6, as the current work assumes a distributed platform connected via shared buses executing multiple PTGs. In order to improve continuity of discussion, completeness and better readability, we have discussed in details all the constraints and objective functions involved in ILP-ES.

7.3 ILP Formulation: ILP-ES

In this section, we present an *Integer Linear Programming* (ILP) based formulation namely, *ILP for Energy-aware Scheduling* (ILP-ES) to solve the scheduling problem considered in this work. First, let us consider a set of binary decision variables X^{gq}_{irlt} and Y^{gq}_{irt}. Here, $X^{gq}_{irlt} = 1$, if task T^{gq}_i in the q^{th} instance of a PTG G^g starts its execution at time step t on processor P_r at level l; $X^{gq}_{irlt} = 0$, otherwise. The variable $Y^{gq}_{krt} = 1$, if message M^{gq}_k in the q^{th} instance of a PTG G^g starts its transmission at time step t on bus B_r; $Y^{gq}_{krt} = 0$, otherwise. We now present the required constraints on the binary decision variables X and Y to model the scheduling problem.

7.3.1 Unique Start Time Constraints

The start time of each *task* should be unique. That is, each task node T^{gq}_i in the q^{th} instance of a PTG G^g must start its execution at a unique time step t on a distinct processor P_r at a certain level l.
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall i \in [1, n^g]$

$$\sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t^{gq}_{is}}^{t^{gq}_{il}} X^{gq}_{irlt} = 1 \tag{7.4}$$

The start time of each *message* should be unique. That is, each message node M^{gq}_k in the q^{th} instance of a PTG G^g must start its transmission at time step t on a bus B_r.
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall k \in [1, m^g]$

$$\sum_{r=1}^{b} \sum_{t=t^{gq}_{ks}}^{t^{gq}_{kl}} Y^{gq}_{krt} = 1 - Z_k \tag{7.5}$$

where,

$$Z_k = \sum_{r=1}^{p} \sum_{l_1=1}^{|L_r|} \sum_{t_1=t_{is}^{gq}}^{t_{il}^{gq}} \sum_{l_2=1}^{|L_r|} \sum_{t_2=t_{js}^{gq}}^{t_{jl}^{gq}} X_{irl_1t_1}^{gq} * X_{jrl_2t_2}^{gq}$$

It may be noted that in the above equation, $Z_k = 1$ when both predecessor (T_i^{gq}) and successor (T_j^{gq}) task nodes of message node M_k^{gq} are assigned to the same processor P_r, forcing the LHS of Equation 7.5 to become 0. Otherwise, $Z_k = 0$. As $X_{irl_1t_1}^{gq}$ and $X_{irl_2t_2}^{gq}$ are binary decision variables, we linearize their multiplication by introducing another binary decision variable $U_{krl_1t_1l_2t_2}^{gq}$ as shown below:

$$Z_k = \sum_{r=1}^{p} \sum_{l_1=1}^{|L_r|} \sum_{t_1=t_{is}^{gq}}^{t_{il}^{gq}} \sum_{l_2=1}^{|L_r|} \sum_{t_2=t_{js}^{gq}}^{t_{jl}^{gq}} U_{krl_1t_1l_2t_2}^{gq} \tag{7.6}$$

Now, the non-linear variables $U_{krl_1t_1l_2t_2}^{gq}$ can be linearized using the following four inequalities.

$$X_{irl_1t_1}^{gq} \geqslant U_{krl_1t_1l_2t_2}^{gq} \tag{7.7}$$

$$X_{jrl_2t_2}^{gq} \geqslant U_{krl_1t_1l_2t_2}^{gq} \tag{7.8}$$

$$U_{krl_1t_1l_2t_2}^{gq} \geqslant X_{irl_1t_1}^{gq} + X_{jrl_2t_2}^{gq} - 1 \tag{7.9}$$

$$U_{krl_1t_1l_2t_2}^{gq} \in \{0, 1\} \tag{7.10}$$

7.3.2 Resource Constraints

Resource bounds for processors must be satisfied at each time step. Any processor P_r can execute at most one task at a given time step. In this regard, it may be noted that a task node T_i^{gq} can only be executing on processor P_r at time t, if it has started at most $t - e_{ir1}^{g} + 1$ time steps earlier.
$\forall t \in [1, \mathcal{H}],\ \forall r \in [1, p]$

$$\sum_{g=1}^{N} \sum_{q=1}^{I^g} \sum_{i=1}^{n^g} \sum_{l=1}^{|L_r|} \sum_{t'=\psi}^{t} X_{irlt'}^{gq} \leqslant 1 \tag{7.11}$$

where, $\psi = t - e^g_{ir1} + 1$.

Similarly, a message node M^{gq}_k can only be transmitting through bus B_r at time t, if it has started at most $t - c^g_{kr} + 1$ time steps earlier.
$\forall t \in [1, \mathcal{H}]$, $\forall r \in [1, b]$

$$\sum_{g=1}^{N} \sum_{q=1}^{I^g} \sum_{k=1}^{m_g} \sum_{t'=\psi}^{t} Y^{gq}_{krt'} \leqslant 1 \tag{7.12}$$

where, $\psi = t - c^g_{kr} + 1$.

7.3.3 Dependency Constraints

The dependencies between nodes must be satisfied. Without loss of generality, let us consider the following relationship from the q^{th} iteration of a PTG G^g: $T^{gq}_i \rightarrow M^{gq}_k \rightarrow T^{gq}_j$. Constraints 7.13 and 7.14 assert that the preceding task node T^{gq}_i of message node M^{gq}_k completes its execution (i) before the start of the succeeding task node T^{gq}_j of M^{gq}_k (in case, both T^{gq}_i and T^{gq}_j are assigned to the same processor) and, (ii) before the start of M^{gq}_k (in case, both T^{gq}_i and T^{gq}_j are assigned to different processors).
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall M^{gq}_k |\ T^{gq}_i = pred(M^{gq}_k)$ and $T^{gq}_j = succ(M^{gq}_k)$,

$$\sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t^{gq}_{is}}^{t^{gq}_{il}} (t + e^g_{irl}) * X^{gq}_{irlt} \leqslant \sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t^{gq}_{js}}^{t^{gq}_{jl}} t * X^{gq}_{jrlt} \tag{7.13}$$

$$\sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t^{gq}_{is}}^{t^{gq}_{il}} (t + e^g_{irl}) * X^{gq}_{irlt} \leqslant \sum_{r=1}^{b} \sum_{t=t^{gq}_{ks}}^{t^{gq}_{kl}} t * Y^{gq}_{krt} + C * Z_k \tag{7.14}$$

where, C is a large constant. It may be observed that by setting C to a sufficiently large value, the constraint in Equation 7.14 is trivially satisfied when both T^{gq}_i and T^{gq}_j are assigned to the same processor ($Z_k = 1$). Suppose the message node M^{gq}_k is scheduled on a bus (i.e., $Z_k = 0$). Then, task node T^{gq}_j ($= succ(M^{gq}_k)$) should commence its execution only after the completion of M^{gq}_k. This constraint is represented as follows:
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall M^{gq}_k |\ T^{gq}_j = succ(M^{gq}_k)$,

$$\sum_{r=1}^{b} \sum_{t=t^{gq}_{ks}}^{t^{gq}_{kl}} (t + c^g_{kr}) * Y^{gq}_{krt} \leqslant \sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t^{gq}_{js}}^{t^{gq}_{jl}} t * X^{gq}_{jrlt} \tag{7.15}$$

It is noteworthy that when $Z_k = 1$, the constraint imposed by Equation 7.5 enforces $\sum_{r=1}^{b} \sum_{t=t_{ks}^{gq}}^{t_{kl}^{gq}} Y_{krt}^{gq}$ to be 0. Hence, the expression $\sum_{r=1}^{b} \sum_{t=t_{ks}^{gq}}^{t_{kl}^{gq}} Y_{krt}^{gq}$ in the LHS of Equation 7.15 also reduces to 0. So, Constraint 7.15 is implicitly satisfied, when Z_k is 1.

7.3.4 Deadline Constraint

All tasks in an instance q of a PTG G^g have to complete their executions within the deadline of instance q. This can be satisfied by restricting the finish times of all sink nodes of G^g in the instance q to be at most the deadline for that instance. The constraint can be written as:
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall T_i^{gq} \mid outdeg(T_i^{gq}) = 0$

$$\sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t_{is}^{gq}}^{t_{il}^{gq}} (t + e_{irl}^{g}) * X_{irlt}^{gq} \leqslant q * D^g \tag{7.16}$$

7.3.5 Objective Function

Our objective is to minimize the overall energy consumption of the system while satisfying all the schedulability constraints. The objective function can be written as:

$$Minimize \sum_{g=1}^{N} \sum_{q=1}^{I^g} \sum_{i=1}^{n^g} \sum_{r=1}^{p} \sum_{l=1}^{|L_r|} \sum_{t=t_{is}^{gq}}^{t_{il}^{gq}} X_{irlt}^{gq} \times E(T_i^g, r, l) \tag{7.17}$$

subject to constraints presented in Equations 7.4 - 7.16.

Complexity Analysis: Table 7.6 summarizes the complexity associated with each constraint of the presented formulation in terms of the order of the number of constraints and the number of variables per constraint. For example, the deadline constraint enforces that all sink task nodes across all instances of all PTGs must have to complete their executions within the deadlines of the respective instances (row-6 of Table 7.6). Hence, this constraint must be separately present for all n^g sink task nodes across I^g instances of all N PTGs and so, the number of constraints is $O(\sum_{g=1}^{N} I^g \times n^g)$ (as may be seen from #constraints column of Table 7.6). Further for each such task, the number of variables which must be considered is $O(p \times |L_r| \times D^g)$, as it can be observed from Equation 7.16.

Table 7.6 presents similar complexity analysis results for all other constraints. The total complexity of the proposed ILP formulation (in terms of the number of constraints) is $O((p \times (|L_r| \times D^g)^2) + (\sum_{g=1}^{N} I^g \times n^g))$.

Constraint Type	Equation No.	#Constraints	#Variables Per Constraint
Unique Start Time	7.4	$O(\sum_{g=1}^{N} I^g \times n^g)$	$O(p \times \|L_r\| \times D^g)$
	7.5	$O(\sum_{g=1}^{N} I^g \times m^g)$	$O(max\{(b \times D^g), (p \times (\|L_r\| \times D^g)^2)\})$
Linearization	7.7	$O(p \times (\|L_r\| \times D^g)^2)$	$O(1)$
	7.8	$O(p \times (\|L_r\| \times D^g)^2)$	$O(1)$
	7.9	$O(p \times (\|L_r\| \times D^g)^2)$	$O(1)$
	7.10	$O(p \times (\|L_r\| \times D^g)^2)$	$O(1)$
Resource Constraints	7.11	$O(p \times \mathcal{H})$	$O(\|L_r\| \times \sum_{g=1}^{N} I^g \times n^g \times D^g)$
	7.12	$O(b \times \mathcal{H})$	$O(\sum_{g=1}^{N} I^g \times m^g \times D^g)$
Dependency Constraints	7.13	$O(\sum_{g=1}^{N} I^g \times m^g)$	$O(p \times \|L_r\| \times D^g)$
	7.14	$O(\sum_{g=1}^{N} I^g \times m^g)$	$O(max\{(p \times \|L_r\| \times D^g), (b \times D^g)\})$
	7.15	$O(\sum_{g=1}^{N} I^g \times m^g)$	$O(max\{(p \times \|L_r\| \times D^g), (b \times D^g)\})$
Deadline	7.16	$O(\sum_{g=1}^{N} I^g \times n^g)$	$O(p \times \|L_r\| \times D^g)$

Table 7.6: *Complexity of ILP*

Due to high computational overheads, we have not implemented our proposed ILP based formulation ILP-ES to generate an optimal solution. With the insight gained through the formal ILP based presentation of the problem, we have designed an effective low-overhead heuristic scheme which can deliver satisfactory solutions within acceptable time bounds. The details of this heuristic scheme are discussed in the next section.

7.4 Proposed Scheme

For scheduling a single PTG on heterogeneous platforms, *list* based heuristic schemes have traditionally been proposed in the literature. A majority of these list scheduling algorithms assume a *fully interconnected system of processing elements* so that the scheduling technique under design does not have to deal with communication contention, in addition to its responsibility of allocating tasks on processors [1,6,11,65]. The primary scheduling objective in most of these algorithms have been to obtain *minimum schedule length* while guaranteeing precedence and resource related constraints. Unlike existing works which consider single PTGs, this work endeavors to tackle the scheduling problem involving *multiple independent real-time periodic PTGs* running on a shared bus based heterogeneous distributed system. Hence, the devised solution must generate a co-schedule which simultaneously resolve the contentions on both processors as well as shared bus resources.

For the problem considered in this work, we devise a heuristic algorithm namely, *Slack Aware Frequency Level Allocator* (SAFLA). First, SAFLA invokes the *Task and Message Co-scheduler* (TMC) which actually extends CC-TMS (Chapter 6, Algorithm 4) to compute an initial schedule with all processors operating at their highest voltages/frequencies (i.e., $l = 1$). For this purpose, TMC invokes the *Task Priority Generator* (TPG) function to construct a priority list of tasks named *TaskPriorityList*. The *priority of a node* is an estimate of the amount of workload that remains to be scheduled before the completion of the sink node. This list is used to govern the sequence in which tasks are selected for processor assignment, in TMC. Internally, TMC selects the next task from *TaskPriorityList* and its predecessor messages. Then, TMC allocates the task node to an appropriate processor and message nodes on one or more buses, so that completion time of the selected task is minimized. Finally, SAFLA takes the initial schedule produced by TMC as input and progressively lowers the processor operating frequencies in a level-by-level fashion such that aggregate energy corresponding to the schedule is minimized. In the next few subsections, we present the details of each stage within SAFLA.

7.4.1 Task Priority Generator (TPG)

The first stage determines a rank for each node in any instance of each PTG in the system using a priority generation mechanism. The priority of a node at the q^{th} instance of any PTG $G^g (\in \mathcal{G})$ is recursively obtained by moving upwards, beginning from the sink task nodes (i.e., nodes with $outdeg(T_i^{gq}) = 0$). The priority, $PG(T_i^{gq})$ of a sink node $T_i^{gq} \in \mathcal{V}^{gq}$ is computed as,

$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall T_i^{gq} \mid outdeg(T_i^{gq}) = 0$

$$PG(T_i^{gq}) = \overline{e_i^g} + (I^g - q) \times D^g \tag{7.18}$$

where, $\overline{e_i^g} = \frac{1}{p}\sum_{r=1}^{p} e_{ir1}^g$, is the average execution time of T_i^{gq} over all processors $P_r \in P$ (at the highest frequency; $l = 1$). The second term in the RHS, $(I^g - q) \times D^g$, ensures that the priority of sink node T_i^{gq} of the q^{th} instance of G^g within the hyperperiod, will be higher than the priority of $T_i^{g(q+1)}$ in the following instance $(q + 1)$ by the value of the relative deadline D^g.

The priorities of the remaining tasks and messages are defined as,
$\forall g \in [1, N]$, $\forall q \in [1, I^g]$

$$PG(T_i^{gq}) = \overline{e_i^g} + \max_{M_k^{gq} \in succ(T_i^{gq})} PG(M_k^{gq}) \tag{7.19}$$

$$PG(M_k^{gq}) = \overline{c_k^g} + PG(T_j^{gq}) \tag{7.20}$$

where, $succ(T_i^{gq})$ is the set of successor message nodes of the task node T_i^{gq}. Among these message nodes, the highest priority message node is employed to compute the priority of its predecessor T_i^{gq}. This process ensures that the priority of T_i^{gq} will always be higher than the priorities of all its successor nodes. In Equation 7.20, $\overline{c_k^g}(= \frac{1}{b}\sum_{r=1}^{b} c_{kr}^g)$ is the average communication time of the message node M_k^{gq} over all buses and $PG(T_j^{gq})$ denotes the priority of T_j^{gq}, the task which immediately succeeds M_k^{gq}.

Pseudo-code of *Task Priority Generator* (TPG) is presented in Algorithm 5. Initially, TPG computes the hyperperiod $\mathcal{H} = LCM(D^1, D^2, \ldots, D^N)$, for the given set of independent PTGs $\mathcal{G} = \{G^1, G^2, \ldots, G^N\}$ (line 1). Next, it finds out the total number of instances I^g of each PTG $G^g \in \mathcal{G}$, over $\mathcal{H}$. Then, the priorities of all nodes in all the PTG instances in the system are calculated based on Equations 7.18, 7.19, and 7.20. Finally, TPG generates the list *TaskPriorityList* consisting of only task nodes, which are organized in non-increasing order of their priorities (line 8).

Complexity Analysis of TPG: The nested *for* loops in line nos. 4 to 7 of TPG dominate its time complexity. Specifically, it takes $O(\sum_{g=1}^{N} I^g \times (n^g + m^g))$. It may be

ALGORITHM 5: TPG

```
Input: PTGs $\mathcal{G} = \{G^1, G^2, \dots, G^N\}$
Output: TaskPriorityList
1 Compute the hyperperiod $\mathcal{H} = LCM(D^1, D^2, \dots, D^N)$
2 for each PTG $G^g \in \mathcal{G}$ do
3     Compute the number of instances: $I^g = \frac{\mathcal{H}}{D^g}$
4 for each PTG $G^g \in \mathcal{G}$ do
5     for each instance $q$ ($\in \{1, 2, \dots, I^g\}$) of $G^g$ do
6         for each node $\mathcal{V}_i^{gq} \in \mathcal{V}^{gq}$ moving bottom-up do
7             Compute priority ($PG$) for $\mathcal{V}_i^{gq}$ using Equations 7.18, 7.19, and 7.20
8 Create a TaskPriorityList consisting of only the task nodes organized in
    non-increasing order of their priorities
```

noted that the number of elements in *TaskPriorityList* is $\sum_{g=1}^{N} I^g \times n^g$. The complexity involved in the sorting of *TaskPriorityList* is $O((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g))$. Hence, the total complexity of TPG is, $O(\sum_{g=1}^{N} I^g \times (n^g + m^g)) + O((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g)) = O((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g))$.

7.4.2 Task and Message Co-scheduler (TMC)

The objective of TMC is to compute an initial task and message schedule with all processors operating at their highest frequencies, while satisfying real-time and resource related constraints. For this purpose, TMC sequentially selects tasks from the head of *TaskPriorityList* and attempts to assign them along with their predecessor messages on appropriate processors and buses, so that the finish times of the selected tasks may be minimized. Here, TMC makes use of the notion of *Earliest Start Time* (EST) and *Earliest Finish Time* (EFT). $EST(T_i^{gq}, P_r)$ is the earliest start time of a task node T_i^{gq} in the q^{th} instance of a PTG G^g on processor $P_r \in P$. Similarly, $EST(M_k^{gq}, B_r)$ is the earliest start time of a message node M_k^{gq} in the q^{th} instance of a PTG G^g on bus $B_r \in B$. The EST values are computed recursively for each node beginning from the source node in the q^{th} instance of a PTG G^g. EST values of source nodes (which are task nodes) are computed as:

$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall T_i^{gq} \mid indeg(T_i^{gq}) = 0$

$$EST(T_i^{gq}, P_r) = max\{avail[P_r], (q-1) \times D^g\} \tag{7.21}$$

here, $avail[P_r]$ denotes the earliest time at which task execution can be commenced on processor P_r. The term $(q-1) \times D^g$ appropriately includes the offset corresponding to q^{th} instance of PTG G^g. The ESTs of other task nodes in the q^{th} instance of G^g are obtained as follows:

$$EST(T_j^{gq}, P_r) = max\{avail[P_r], \max_{M_k \in pred(T_j^{gq})} AFT(M_k^{gq})\} \tag{7.22}$$

where, $AFT(M_k^{gq})$ represents the actual completion time of the predecessor message M_k^{gq}. EST values associated with the message nodes are calculated as,

$$EST(M_k^{gq}, B_r) = max\{avail[B_r], AFT(T_i^{gq})\} \tag{7.23}$$

where, $avail[B_r]$ denotes the earliest instant at which message transmission can be commenced on bus B_r and $AFT(T_i^{gq})$ denotes the actual completion time of the predecessor T_i^{gq} of M_k^{gq}.

EFT values of all tasks and messages can be calculated as,

$$EFT(T_i^{gq}, P_r) = EST(T_i^{gq}, P_r) + e_{ir1}^g \tag{7.24}$$

$$EFT(M_k^{gq}, B_r) = EST(M_k^{gq}, B_r) + c_{kr}^g \tag{7.25}$$

When both the preceding and succeeding tasks T_i^{gq} and T_j^{gq} of message node M_k^{gq} are scheduled on the same processor, then the message node M_k^{gq} becomes immaterial as the message is not actually transmitted over a bus. That is, $avail[B_r] = c_{kr}^g = 0$ and $EST(M_k^{gq}, B_r) = EFT(M_k^{gq}, B_r) = AFT(T_i^{gq})$.

The proposed algorithm, TMC is presented in Algorithm 6. Initially, the TMC algorithm invokes $TPG(\mathcal{G})$ to generate *TaskPriorityList*, a sorted (non-increasing order of priorities) list of all task nodes present across all instances (in hyperperiod $\mathcal{H}$) of all PTGs in $\mathcal{G}$ (line 1). Then, the first node in *TaskPriorityList* is chosen and scheduled

ALGORITHM 6: TMC

```
Input: PTGs $\mathcal{G}$, Processors $P$, Buses $B$
Output: Task and message schedule (start times of all nodes along with
        assignment of tasks/messages to processors/buses) in each instance of a
        $G^g$
1  $TaskPriorityList$ = Invoke $TPG(\mathcal{G})$
2  Set $avail[P_r] = avail[B_r] = 0$, for all processors $P_r \in P$ and buses $B_r \in B$
3  while $TaskPriorityList \neq \emptyset$ do
4      Select the first entry $T_j^{gq}$, from $TaskPriorityList$
5      Let $MsgPriorityList$ be the list consisting of all predecessor message
         nodes of task $T_j^{gq}$ organized in non-increasing order of priorities
         (computed using $PG(M_k^{gq})$)
6      for all processors $P_r \in P$ do
7          Assume that $T_j^{gq}$ is assigned on $P_r$
8          For each bus, set $tempAvail[B_r] = avail[B_r]$
9          for each $M_k^{gq} \in MsgPriorityList$ do
10             for each bus $B_{r'}$ do
11                 Compute $EFT(M_k^{gq}, B_{r'})$
12             Assume that $M_k^{gq}$ is assigned on $B_{r'}$ that minimizes its EFT and
                 update $tempAvail[B_{r'}] = EFT(M_k^{gq}, B_{r'})$
13         Compute $EFT(T_j^{gq}, P_r)$
14     Assign $T_j^{gq}$ to $P_r$ that minimizes its EFT and update $avail[P_r]$ =
         $EFT(T_j^{gq}, P_r)$
15     Given $T_j^{gq}$ to $P_r$, assign all its predecessor messages on buses such that
         their EFT's are minimized and update $avail[B_r]$ accordingly
16     Remove $T_j^{gq}$ from $TaskPriorityList$
```

with its predecessor message nodes. These operations are repeated until the task priority list becomes empty (line 3 to 16).

The steps involved in the scheduling of a selected task node T_j^{gq} may be elaborated as follows: First, the *Task and Message Co-scheduler* (Algorithm 6) calculates the priority list *MsgPriorityList* consisting of all predecessor messages of T_j^{gq} sorted in non-increasing order of their priorities (line 5). T_j^{gq} is then tentatively allocated on $P_r \in P$ (line 7). Given the assignment of T_j^{gq} on P_r, its predecessor message nodes are selected from the priority list *MsgPriorityList* and tentatively assigned to the buses where their EFT values are minimized (lines 9 to 12). Subsequent to the tentative assignment of message nodes on buses, $EFT(T_j^{gq}, P_r)$ is computed. Similarly, EFT of T_j^{gq} on each processor

$P_r \in P$ is computed (lines 6 to 13). T_j^{gq} is finally assigned to that processor on which its EFT becomes minimum. Given the assignment of T_j^{gq} on a particular process P_r, its predecessor message nodes are assigned to buses such that their EFT's become minimum (line 15).

7.4.3 Complexity Analysis of TMC

The computation of *TaskPriorityList* takes $O((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g))$ (line no. 1). In order to initialize $avail[P_r]$ and $avail[B_r]$, $O(p+b)$ time is consumed (line no. 2). The overhead incurred by the *while* loop (line nos. 3 to 16) is dominated by the complexity associated with line no. 11 inside the inner-most *for* loop. By multiplying the complexity of $EFT()$ with the number of times line no. 10 is invoked, we can determine the complexity of the *while* loop. From Equations 7.25 and 7.23, it may be inferred that $EFT(M_k^{gq}, B_{r'})$ incurs $O(1)$ overhead. It may also be noted that $O(p \times (\sum_{g=1}^{N} I^g \times m^g) \times b)$ represents an upper bound on the number of times line no. 11 is executed. Though, the *while* loop at line no. 3 iterates for $O(\sum_{g=1}^{N} I^g \times n^g)$ times, the *for* loop at line no. 9 is invoked at most $O(\sum_{g=1}^{N} I^g \times m^g)$ times (as each message has only one child task). Given that p and b represent the total number of times the for loops in line nos. 6 and 10 are invoked, the complexity of the TMC algorithm can be written as $O(\max\{((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g)), (p \times (\sum_{g=1}^{N} I^g \times m^g) \times b)\})$.

$T_1^{2,1}$: TMC chooses task $T_1^{2,1}$ from $TaskPriorityList$. The message priority list of $T_1^{2,1}$ is empty, because it does not have any predecessor.

$P_1 :\ EST(T_1^{2,1}, P_1) = 0, EFT(T_1^{2,1}, P_1) = 0 + 1 = 1$

$P_2 :\ EST(T_1^{2,1}, P_2) = 0,\ EFT(T_1^{2,1}, P_2) = 0 + 2 = 2$

$P_3 :\ EST(T_1^{2,1}, P_3) = 0,\ EFT(T_1^{2,1}, P_3) = 0 + 4 = 4.$

Since $EFT(T_1^{2,1}, P_1) < EFT(T_1^{2,1}, P_2) < EFT(T_1^{2,1}, P_3)$, $T_1^{2,1}$ is assigned to processor P_1 with 0 as its start time. The value of $avail[P_1]$ is updated to 1. $T_1^{2,1}$ is then removed from $TaskPriorityList$.

$T_1^{1,1}$: Then, $T_1^{1,1}$ is chosen from $TaskPriorityList$. Similar to $T_1^{2,1}$, the message priority list of $T_1^{1,1}$ is also empty.

$P_1: \ EST(T_1^{1,1}, P_1) = 1, \ EFT(T_1^{1,1}, P_1) = 1 + 5 = 6$

$P_2: \ EST(T_1^{1,1}, P_2) = 0, \ EFT(T_1^{1,1}, P_2) = 0 + 7 = 7$

$P_3: \ EST(T_1^{1,1}, P_3) = 0, \ EFT(T_1^{1,1}, P_3) = 0 + 8 = 8.$

$EFT(T_1^{1,1}, P_1) < EFT(T_1^{1,1}, P_2) < EFT(T_1^{1,1}, P_3)$. So, $T_1^{1,1}$ is assigned to processor P_1 with start time 1 and $avail[P_1] = 6$. $T_1^{1,1}$ is removed from the list.

$T_2^{2,1}$: Next, $T_2^{2,1}$ is selected and its message priority list is $\{M_1^{2,1}\}$.

$P_1:$ If $T_2^{2,1}$ is assigned on processor P_1, then $M_1^{2,1}$ becomes invalid. $EST(M_1^{2,1}, B_1) = EST(M_1^{2,1}, B_2) = 6$; $EFT(M_1^{2,1}, B_1) = EFT(M_1^{2,1}, B_2) = 6$; $EST(T_2^{2,1}, P_1) = 6$, $EFT(T_2^{2,1}, P_1) = 6 + 2 = 8$.

$P_2:$ If $T_2^{2,1}$ is assigned on processor P_2 then $tempAvail[B_1] = tempAvail[B_2] = 0$; $EST(M_1^{2,1}, B_1) = EST(M_1^{2,1}, B_2) = 1$; $EFT(M_1^{2,1}, B_1) = 1 + 1 = 2$, $EFT(M_1^{2,1}, B_2) = 1 + 2 = 3$; $tempAvail[B_1] = 2$; $EST(T_2^{2,1}, P_2) = 2$, $EFT(T_2^{2,1}, P_1) = 2 + 2 = 4$.

$P_3:$ If $T_2^{2,1}$ is assigned on processor P_3 then $tempAvail[B_1] = tempAvail[B_2] = 0$; $EST(M_1^{2,1}, B_1) = EST(M_1^{2,1}, B_2) = 1$; $EFT(M_1^{2,1}, B_1) = 1 + 1 = 2$, $EFT(M_1^{2,1}, B_2) = 1 + 2 = 3$; $tempAvail[B_1] = 2$; $EST(T_2^{2,1}, P_3) = 2$, $EFT(T_2^{2,1}, P_3) = 2 + 6 = 8$.

Since $EFT(T_2^{2,1}, P_2) < EFT(T_2^{2,1}, P_1) < EFT(T_2^{2,1}, P_3)$, $T_2^{2,1}$ is assigned to processor P_2 and the message $M_1^{2,1}$ is assigned to bus B_1. Availability of processor P_2 and bus B_1 are updated to $avail[P_2] = 4$, $avail[B_1] = 2$.

The schedule is constructed in a similar manner for the remaining tasks/message nodes. The final schedule is depicted in Figure 7.3.

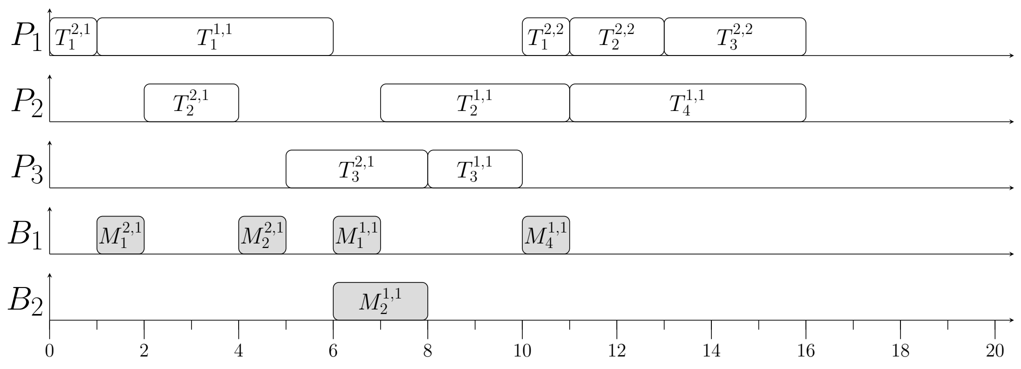


Figure 7.3: *Example (TMC): Gantt chart representation of the schedule*

7.4.4 Slack Aware Frequency Level Allocator (SAFLA)

Given the ordered processor and bus assignments as prescribed by TMC, the SAFLA algorithm attempts to assign appropriate voltage/frequency-levels to the tasks (during execution on their assigned processors) such that aggregate energy dissipated by the schedule is minimized. SAFLA is a heuristic approach which is able to deliver satisfactory solutions while incurring moderate polynomial time complexity. Initially, the algorithm starts by invoking TMC which returns task-to-processor mappings, message-to-bus mappings, and also the start times of the tasks and messages, assuming all task nodes to be executing with the processors operating at their maximum frequencies ($l = 1$). It may be noted that the available slacks in terms of the unused time slots within the schedule of each processor can be utilized for energy savings. SAFLA conducts such energy savings with a poised approach where at any time the level of an appropriate task is enhanced by one. This task level enhancement, which causes reduction in a processor's operating frequency with consequent increase in execution time, must be carried-out while taking care that this doesn't lead to overlapped executions of multiple tasks/messages assigned on the same resource. In order to ensure this, SAFLA adds a zero weighted edge ($\mathcal{V}_i \xrightarrow{0} \mathcal{V}_j$) between two mutually independent nodes $\mathcal{V}_i$ and $\mathcal{V}_j$, if both nodes are mapped to the same resource r and $\mathcal{V}_j$ immediately starts its execution/transmission after $\mathcal{V}_i$.

Next, SAFLA calculates the slack times for each task node T_i^{gq} as:

$\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall T_i^{gq} \in \mathcal{V}^{gq}$,

$$slack(T_i^{gq}) = ALAP(T_i^{gq}) - S(T_i^{gq}) \tag{7.26}$$

where, $S(T_i^{gq})$ is the start time of T_i^{gq} returned by the TMC algorithm and $ALAP(T_i^{gq})$ is the ALAP time of the task node T_i^{gq} in the q^{th} instance of PTG G_g.

1. The tasks' ALAP times are recursively calculated (in a bottom-up fashion) as follows:
   $\forall g \in [1, N]$, $\forall q \in [1, I^g]$, $\forall T_i^{gq} \in \mathcal{V}^{gq}$,
   $$ALAP(T_i^{gq}) = \begin{cases} q \times D^g - e_{ir}^g, & [\text{If } outdeg(T_i^{gq}) = 0] \\ \min\{q \times D^g, \phi\} - e_{ir}^g, & [\text{Otherwise}] \end{cases}$$
   where, $\phi = \min\limits_{\mathcal{V}_j^{g'q'} \in succ(T_i^{gq})} ALAP(\mathcal{V}_j^{gq})$
2. ALAP times of the message nodes are recursively determined (upward) as follows:
   $$ALAP(M_k^{gq}) = \min_{\mathcal{V}_j^{g'q'} \in succ(M_k^{gq})} ALAP(\mathcal{V}_j^{gq}) - c_{kr}^g$$
   where, $succ(M_k^{gq})$ denotes the successors of message M_k^{gq}.

At the next step, the currently assigned processor voltage/frequency-levels during the execution of each task instance is enhanced by using the slack available in the system. The selection of task nodes is based on a prioritization key called $cost(T_i^{gq})$ which is defined as follows:

$$cost(T_i^{gq}) = \max\left\{\frac{E'(T_i^g, r, l, l+1)}{e_{ir(l+1)}^g - e_{irl}^g}, \frac{E'(T_i^g, r, l, |L_r|)}{e_{ir(|L_r|)}^g - e_{irl}^g}\right\} \tag{7.27}$$

The RHS of Equation 7.27 has two components. The first component provides an estimate of the reduction in energy dissipated (per unit additional allocated time) due to the execution of T_i^{gq} on processor P_r, as its level is increased from l to $l+1$. The second component provides a similar measure, as the level of T_i^{gq} is increased from its current level l to its maximum level $|L_r|$. $cost(T_i^{gq})$ is obtained as the larger among these two components.

The SAFLA algorithm makes a max-heap of task nodes with $cost(T_i^{gq})$ as the key. The algorithm proceeds in an iterative fashion by repeatedly extracting task T_i^{gq} at the root of the heap, incrementing its associated processor voltage/frequency-level by 1 (if possible) and updating its slack and finish times. This level shift is actually allowed if a feasible schedule can still be generated after incorporating the additional execution time for T_i^{gq} within the schedule of P_r. If the maximum level $|L_r|$ has still not been reached ($l < |L_r|$), then SAFLA recomputes $cost(T_i^{gq})$ and reinserts T_i^{gq} into the heap. Additionally, SAFLA updates the start times and finish times of descendant nodes, and slack times of all nodes. This process repeats until residual resources are completely exhausted or all tasks execute with their assigned processors operating at the lowest frequencies ($l = |L_r|$). Algorithm 7 depicts a step-wise description of SAFLA.

Time Complexity of SAFLA: Assignment of base voltage/frequency-levels to processors takes $O(p)$ iterations (line no. 1). The complexity of computing a TMC schedule is $O(\max\{((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g)), (p \times (\sum_{g=1}^{N} I^g \times m^g) \times b)\})$. The overhead of adding dummy edges between mutually independent node pairs assigned on the same resource is $O(\sum_{g=1}^{N} I^g \times (n^g + m^g))$ (line nos. 3 to 6). Computation overheads associated with the calculation of $slack(T_i^{gq})$ and $cost(T_i^{gq})$ for all tasks (line nos. 7 to 11) are $O(\sum_{g=1}^{N} I^g \times (n^g + m^g))$. Formation of the initial max-heap (line no. 13) takes $O(\sum_{g=1}^{N} I^g \times n^g)$. The voltage/frequency-level upgradation process at line no. 14 iterates $O(|L_r| \times \sum_{g=1}^{N} I^g \times n^g)$ times. Modification of the voltage/frequency-level, finish time and along with the creation of *UpdateList* for the current task T_i^{gq} takes $O(1)$ time (line nos. 16 and 18 to 20). The overhead associated with the update of start and finish times of all descendant nodes of T_i^{gq} requires $O(\sum_{g=1}^{N} I^g \times (n^g + m^g))$ time (line nos. 21 to 27). Computation of $cost(T_i^{gq})$ and reinsertion of T_i^{gq} into the max-heap (line nos. 28 to 29) takes $O(\log(\sum_{g=1}^{N} I^g \times n^g))$ time. Computation of slack time for the task nodes in the max-heap require $O(\sum_{g=1}^{N} I^g \times (n^g + m^g))$ time (line nos. 30 to 31). So, the overhead associated with the *while* loop from line nos. 14 to 31 is $O(|L_r| \times \sum_{g=1}^{N}(I^g \times n^g) \times \sum_{g=1}^{N} I^g \times (n^g + m^g))$. Finally, time complexity of SAFLA can be written as $O(\max\{((\sum_{g=1}^{N} I^g \times n^g) \times log(\sum_{g=1}^{N} I^g \times n^g)), (p \times (\sum_{g=1}^{N} I^g \times m^g) \times$

ALGORITHM 7: SAFLA

Input: PTGs $\mathcal{G}$, Processors P, Buses B

Output: Schedule of tasks and messages (start times of nodes, level to execute each task)

```
1  Set maximum frequency ($l = 1$) to all processors
2  Compute start times of nodes and node-to-resource mapping using TMC (Algorithm 6)
3  for each resource $r$ do
4      for each mutually independent node pairs $(\mathcal{V}_i, \mathcal{V}_j)$ assigned on resource $r$ do
5          if $\mathcal{V}_j$ executes immediately after $\mathcal{V}_i$ then
6              Add zero weighted edge $(\mathcal{V}_i \xrightarrow{0} \mathcal{V}_j)$
7  for each PTG $G^g \in \mathcal{G}$ do
8      for each instance $q \in \{1, 2, \ldots, I^g\}$ do
9          for each task node $T_i^{gq} \in \mathcal{V}^{gq}$ do
10             Compute $slack(T_i^{gq})$ using Equation 7.26
11             Compute $cost(T_i^{gq})$ using Equation 7.27
12 Let $l$ be the currently chosen voltage/frequency-level at which the processor assigned to $T_i^{gq}$ should operate during its execution
13 Create a max-heap of tasks using $cost(T_i^{gq})$ as key
14 while max-heap is non-empty do
15     Remove root node $T_i^{gq}$
16     $\Delta e_i \leftarrow e^g_{ir(l+1)} - e^g_{irl}$
17     if $\Delta e_i \leq slack(T_i^{gq})$ then
18         Increase the processor level for task $T_i^{gq}$: $l \leftarrow l + 1$
19         Update finish time of $T_i^{gq}$: $F_i^{gq} \leftarrow F_i^{gq} + \Delta e_i$
20         Create an empty successor list, UpdateList and add $T_i^{gq}$ to it
21         while UpdateList is non-empty do
22             Remove the front node (say, $\mathcal{V}_{i_1}^{g_1q_1}$) from UpdateList
23             forall successor $\mathcal{V}_{i_2}^{g_2q_2}$ of $\mathcal{V}_{i_1}^{g_1q_1}$ do
24                 if $S_{i_2}^{g_2q_2} < F_{i_1}^{g_1q_1}$ then
25                     Update finish time of $\mathcal{V}_{i_2}^{g_2q_2}$: $F_{i_2}^{g_2q_2} \leftarrow F_{i_2}^{g_2q_2} + (F_{i_1}^{g_1q_1} - S_{i_2}^{g_2q_2})$
26                     Update start time of $\mathcal{V}_{i_2}^{g_2q_2}$: $S_{i_2}^{g_2q_2} \leftarrow S_{i_2}^{g_2q_2} + (F_{i_1}^{g_1q_1} - S_{i_2}^{g_2q_2})$
27                     Add $\mathcal{V}_{i_2}^{g_2q_2}$ to UpdateList
28         if $(l + 1) < |L_r|$ then
29             Compute $cost(T_i^{gq})$ (Equation 7.27) and reinsert $T_i^{gq}$ into the max-heap
30     forall tasks $T_i^{gq}$ in max-heap do
31         Compute $slack(T_i^{gq})$ using Equation 7.26
```

$b), (|L_r| \times \sum_{g=1}^{N}(I^g \times n^g) \times \sum_{g=1}^{N} I^g \times (n^g + m^g))\})$. Let use the symbols $\mathcal{N}$ and $\mathcal{M}$ to denote $\sum_{g=1}^{N} I^g \times n^g$ and $\sum_{g=1}^{N} I^g \times m^g$, respectively. Then, time complexity can be represented as: $O(\max\{(\mathcal{N} \times log\mathcal{N}), (p \times \mathcal{M} \times b), (|L_r| \times \mathcal{N} \times (\mathcal{N} + \mathcal{M}))\})$.

Example (contd.): Let us consider the PTG shown in Figure 7.2. The schedule constructed by TMC is depicted in Figure 7.3. From the given TMC schedule, it can be seen that $T_1^{2,1}$ and $T_1^{1,1}$ are scheduled on the same processor P_1. Hence, a zero-weighted edge from $T_1^{2,1}$ to $T_1^{1,1}$ is added. Similarly, for all mutually independent pairs of nodes scheduled on the same resource, a zero-weighted edge is added if one node starts immediately after another. The initial values of $slack(T_i^{gq})$ and $cost(T_i^{gq})$ corresponding to task nodes are as follows: $slack(T_1^{1,1}) = slack(T_2^{1,1}) = slack(T_3^{1,1}) = slack(T_4^{1,1}) = slack(T_1^{2,2}) = slack(T_2^{2,2}) = slack(T_3^{2,2}) = 4$, $slack(T_1^{2,1}) = slack(T_2^{2,1}) = slack(T_3^{2,1}) = 2$ and $cost(T_1^{1,1}) = cost(T_1^{2,1}) = cost(T_1^{2,2}) = cost(T_2^{2,2}) = cost(T_3^{2,2}) = 2.5$, $cost(T_2^{1,1}) = cost(T_2^{2,1}) = 1.14$, $cost(T_3^{1,1}) = 0.54$, $cost(T_4^{1,1}) = 1.26$, $cost(T_3^{2,1}) = 0.87$. A max-heap is built using these $cost(T_i^{gq})$ values. The task $T_1^{1,1}$ with the highest key value (currently, at the root of the max-heap) is extracted from the heap and its frequency is decreased by enhancing the processor level from $l = 1$ to $l = 2$ as the additional computation requirement, $\Delta e_1 = 6 - 5 = 1$, can be satisfied by the available slack ($slack(T_1^{1,1}) = 4$). The finish time of $T_1^{1,1}$ becomes, $F_1^{1,1} + \Delta e_1 = 6 + 1 = 7$. Now, with respect to the finish time of $T_1^{1,1}$, the start and finish times of the descendant nodes are updated accordingly. The cost value ($cost(T_1^{1,1}) = 2.79$) of task $T_1^{1,1}$ is recomputed using Equation 7.27 and it is reinserted into the max-heap as the assigned processor frequency is still not at the maximum possible level (i.e., minimum frequency). The slack times of all nodes are recomputed using Equation 7.26 ($slack(T_1^{1,1}) = slack(T_2^{1,1}) = slack(T_3^{1,1}) = slack(T_4^{1,1}) = 3$, $slack(T_1^{2,1}) = slack(T_2^{2,1}) = slack(T_3^{2,1}) = 2$ and $slack(T_1^{2,2}) = slack(T_2^{2,2}) = slack(T_3^{2,2}) = 4$).

Next $T_1^{1,1}$, the task having the largest key, is extracted from the heap and the processor level is upgraded from $l = 2$ to $l = 3$. Its finish time $F_1^{1,1}$ becomes 8. The cost value of task $T_1^{1,1}$ is recomputed and it is reinserted into the max-heap. Start and finish times

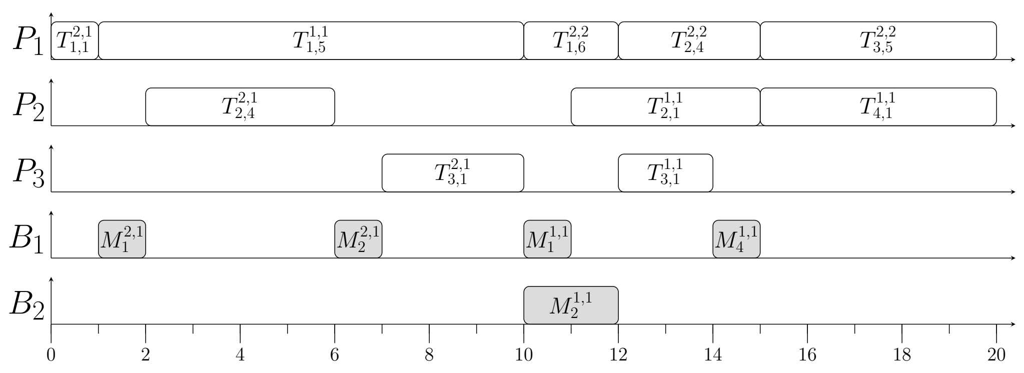


Figure 7.4: *Example (SAFLA): The schedule as a gantt chart*

of all successor nodes of $T^{1,1}_1$, and slack times of all task nodes in the heap are updated accordingly. This process repeats till the max-heap becomes empty (i.e., each task has either been assigned to the maximum processor level (i.e., minimum frequency) or its slack has become 0). Figure 7.4 show the gantt chart representing the final schedule. Energy consumed by the schedule generated by SAFLA to execute all the PTGs (in Figure 7.2) over the length of the hyperperiod is 87.05 units. This value corresponds to a reduction of 22.83 in comparison to the TMC schedule (that SAFLA takes as input) which consumes 109.88 units of energy. $\square$

7.5 Experimental Evaluation

The performance of SAFLA, the strategy proposed in this chapter, has been experimentally evaluated using real-world benchmark PTGs.

Experimental Setup: We have considered the following benchmark PTGs namely, *Gaussian Elimination* (GE) [6], *Epigenomics* [2], *Laplace* [4] and *Stencil* [4], to experimentally evaluate the algorithm's performance.

- A *Gaussian Elimination* task graph contains $((\chi^2+\chi-2)/2)$ task nodes & $(\chi^2-\chi-1)$ message nodes, where χ is an input parameter. Figure 7.5a shows a *Gaussian Elimination* task graph containing 14 task nodes and 19 message nodes, when $\chi = 5$.

- *Epigenomics* task graph contains $(4\gamma + 4)$ task nodes & $(5\gamma + 2)$ message nodes,

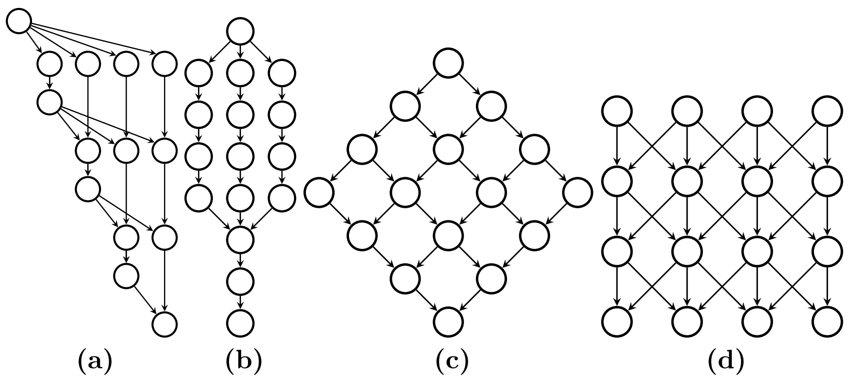


Figure 7.5: *(a) Gaussian Elimination (GE) [6], (b) Epigenomics [2] (c) Laplace [4] (d) Stencil [4]*

where γ is an input parameter. Figure 7.5b shows a *Epigenomics* task graph containing 16 task nodes and 17 message nodes, when $\gamma = 3$.

- The *Laplace* PTG contains φ^2 task nodes & $(2\varphi^2 - 2\varphi)$ message nodes, where φ is an input parameter. Figure 7.5c shows Laplace PTG containing the 16 task nodes and 24 message nodes, when $\varphi = 4$.

- The *Stencil* PTG consists of $(\lambda \times \xi)$ task nodes & $((\lambda - 1) \times (3\xi - 2))$ message nodes, where λ is an input parameter. For simplicity, we have assumed $\lambda = \xi$, in our experiments. For example, Figure 7.5d shows a *Stencil* PTG with 16 tasks and 30 messages, when $\lambda = 4$.

The parameters corresponding to the PTGs as well as the processing platforms are varied as follows: (1) *#task and message nodes*: $\chi = \{4\}$ (*Gaussian Elimination*), $\gamma = \{3\}$ (*Epigenomics*), $\varphi = \{4\}$ (*Laplace*), $\lambda = \{4\}$ (*Stencil*). (2) *#processors (p)*: we have considered 4 or 8 processor systems. (3) *#buses (b)*: Systems having 2 or 4 buses are considered. (4) *Communication-to-Computation Ratio*: CCR denotes the ratio of the average cost of communication to computation for a given PTG. We have conducted experiments for two distinct values of $CCR(\in \{0.25, 0.5\})$. (5) #frequency-levels of each processor is randomly chosen from the range [3, 6]. (6) Voltage (in *volt*) and frequency (in GHz) at each processor level are taken from the uniform random distribution [0.8, 3.2]. (7) Another uniform random distribution ([10 ms, 50 ms]) is employed to obtain *execution times* e^g_{ir1} for each task(T_i)-processor(P_r) combination (at the base voltage/frequency-level) and *communication times* c^g_{kr} for each message(M_k)-bus(B_r) combination. Given a desired CCR, the obtained data communication times c^g_{kr} are accordingly scaled.

Computation of the relative deadline of a PTG: For each PTG $G^g \in \{G^1, G^2, \ldots, G^N\}$, first TMC is used to determine the makespan ML_g considering all system resources to be assigned to G^g (i.e. G^g runs stand alone), and assuming all processors to be always executing at their maximum frequencies. After obtaining ML_g of all the PTGs, the relative deadline of G^g is computed using the formula:

$$D^g = \mathit{Round\text{-}off}(ML \times \{t_1 + \frac{ML_g - minML_g}{maxML_g - minML_g} \times t_2\}) \tag{7.28}$$

Here, $ML = \sum_{g=1}^{N} ML_g$. $maxML_g$ and $minML_g$ respectively represent the maximum and minimum ML_g values among the given set of PTGs. The parameters t_1 and t_2 are two constants whose values are set to 0.4 and 0.2, in our experiments. The *Round-off(x)* function is used to round-off the argument x to the lowest multiple of 100 which is greater than or equal to x.

Normalized Workload: It is the ratio of the total execution time of all the tasks over all iterations across all PTGs on all processors within a hyperperiod $\mathcal{H}$, to the total available time on all processors over $\mathcal{H}$. It is defined as,

$$NW = \frac{\sum_{g=1}^{N} I^g \times \left(\sum_{r=1}^{p} \sum_{i=1}^{n^g} e^g_{ir1}\right)}{\mathcal{H} \times p}$$

Normalized Workload Ratio: Let us consider two workloads NW_{act} and NW_{gen} which are identical in terms of the application DTGs considered, and individual deadlines of the applications. Hence, both workloads does have the same hyperperiod as well as the same number of instances for each DTG application. Workload NW_{act} is obtained from NW_{gen} by multiplying the execution times of each task in NW_{gen} by a constant NWR (*Normalized Workload Ratio*). Thus, NWR (in %) $= \frac{NW_{act}}{NW_{gen}} \times 100$.

Comparison against baseline algorithm: To evaluate and compare the performance of SAFLA, we have developed a baseline algorithm called, *Baseline Slack Aware Multi-*

PTG Scheduler (BSAMS). The basic structure of BSAMS is same as that of the SAFLA algorithm. However, the voltage/frequency-level upgrade mechanism in BSAMS differs from SAFLA, being based on the existing strategy *Multiple-Workflows-Slack-Time-Reclaiming* (MWSTR), proposed in [8]. While SAFLA employs a level-by-level enhancement approach, MWSTR attempts to greedily upgrade the voltage/frequency-level for a selected task to the maximum possible value by myopically attempting use of the entire slack available at any given time, if required. Thus for BSAMS, the voltage/frequency-level corresponding to the execution of a task may be upgraded by multiple levels in a single iteration of the algorithm. Due to this mechanism, while the energy optimization phase of SAFLA iterates $O(|L_r| \times \sum_g^N I^g \times n^g)$ times, that for BSAMS iterates only $O(\sum_g^N I^g \times n^g)$ times. However, although BSAMS has lower time-complexity, its solution qualities are significantly poorer than SAFLA as exhibited through our experiments.

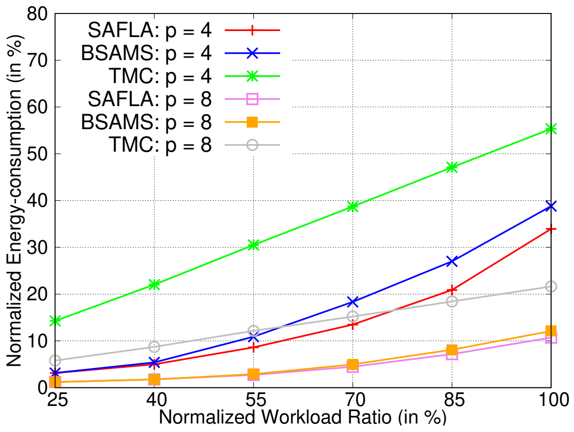
Performance Metrics: (1) *Normalized Energy-consumption* (NE): NE (in %) $= \frac{E_{ACT}}{E_{MAX}} \times 100$, where, E_{MAX} and E_{ACT} denote the maximum and actual energy dissipated in the execution of all instances of all PTGs over the length of the hyperperiod. In fact, E_{MAX} represents the total amount of energy that may be dissipated by continuously operating all processors at their maximum voltages/frequencies (i.e., minimum level) over the entire duration of the hyperperiod. E_{ACT} is the amount of energy consumed by the processors to execute tasks as per the schedule generated by TMC, SAFLA or BSAMS. (2) *Normalized Running Time* (NRT): It is the ratio of the total running time to the total number of nodes (including both tasks and messages) over the hyperperiod. NRT is defined as,

$$NRT = \frac{Total\ running\ time}{\sum_{g=1}^{N} I^g \times (n^g + m^g)} \tag{7.29}$$

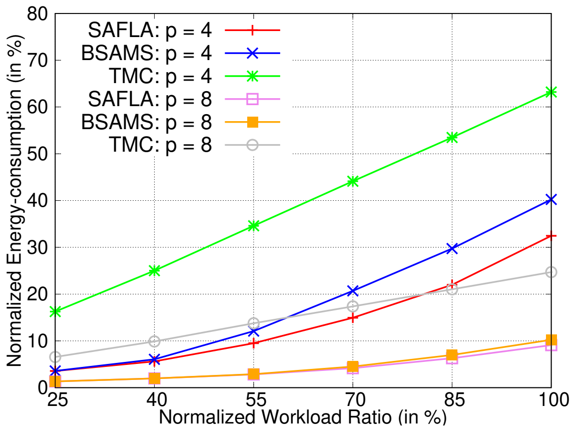
Each data point in the plots are obtained as the average over 500 runs of a specific scheduler, on different PTG instances produced by carefully varying a chosen set of parameters. The experiments are conducted on a system having Intel(R) Core(TM) i5-4300U CPU running Linux Kernel 4.15.0-88-generic.

Experiment-1: Variation in #processors: Figure 7.6 shows the experimental results. Here, #processors (p) is varied from 4 to 8, while fixing the number of buses (b)

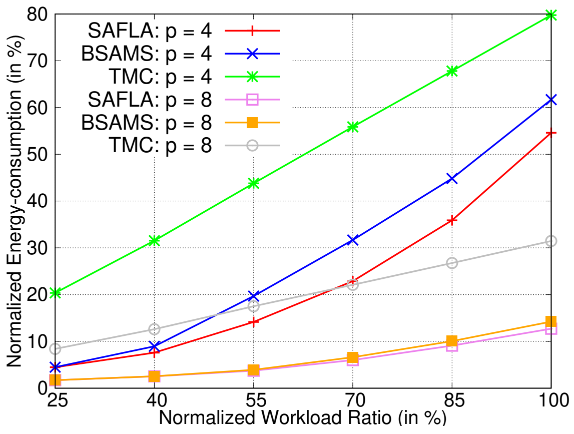
to 4 and CCR to 0.5. It can be observed that the normalized energy-consumption NE *becomes lower as #processors increases, for any given value of* NWR. This is because as #processors become higher, residual capacity increases, and the system uses it to decrease the processors' voltage/frequency (i.e., enhance processor level). For example, in *Gaussian Elimination* and *Epigenomics* (Figure 7.6a) with $NWR = 85\%$, the NEs returned by SAFLA for $p = 4$ and $p = 8$ are ∼21% and ∼7%, respectively.



(a) *GE, Epigenomics*



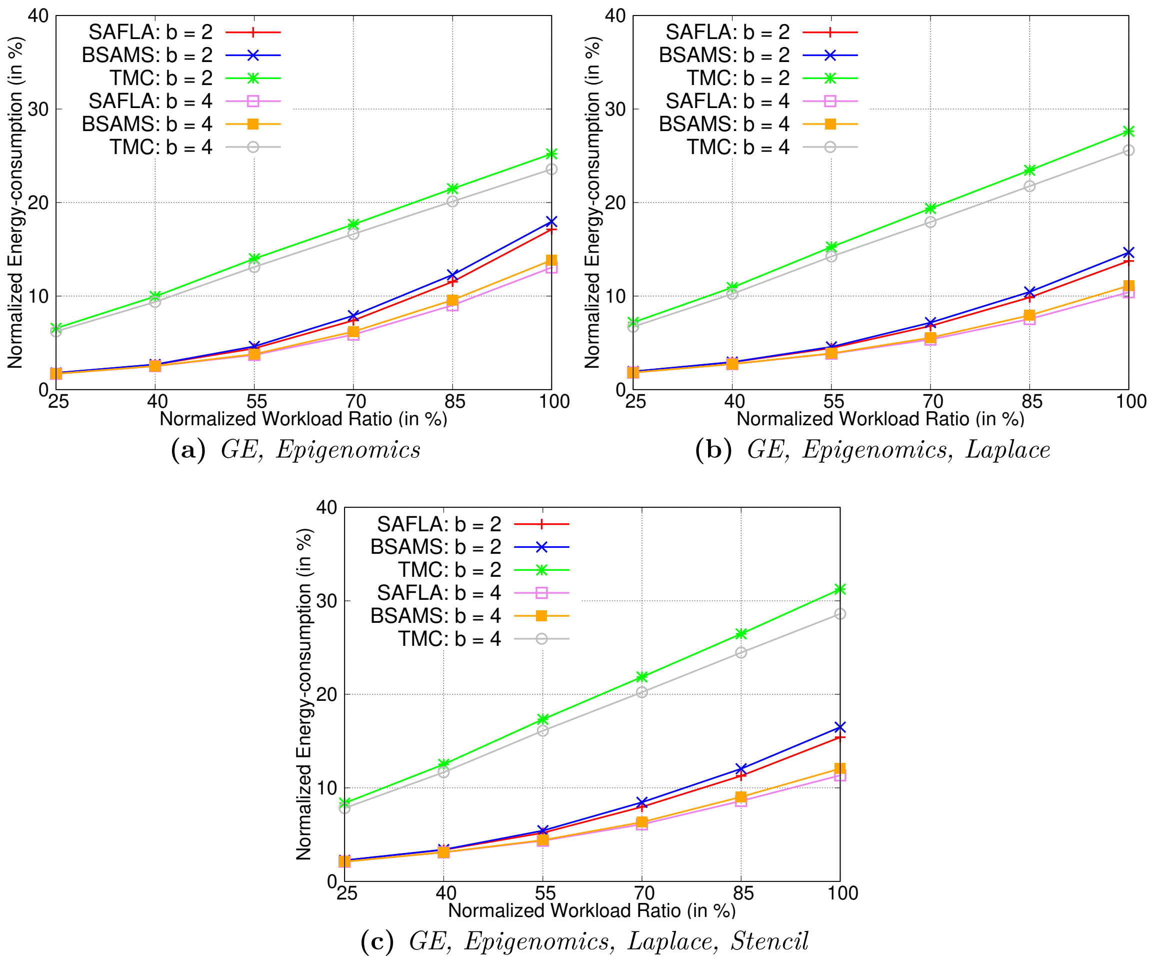
(b) *GE, Epigenomics, Laplace*



(c) *GE, Epigenomics, Laplace, Stencil*

Figure 7.6: *Variation in #processors*

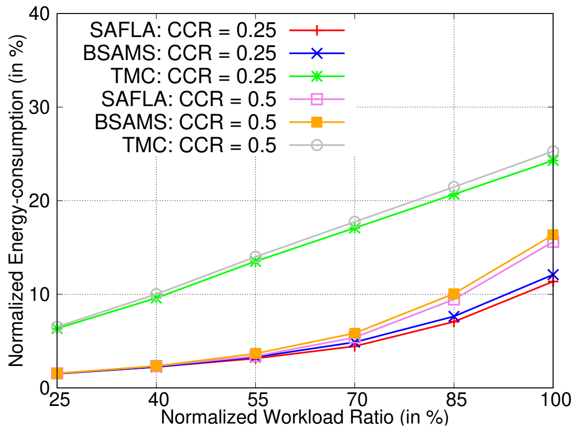
Experiment-2: Effect of variation in the no. of buses: The results of this experiment is depicted in Figure 7.7. Here, #buses (b) is varied from 2 to 4, while fixing #processors (p) to 8 and CCR to 0.5. It can be observed that the normalized energy-consumption *NE decreases with increase in the number of buses, for any given NWR value.* This may be attributed to the fact that as #buses become higher, the resource contention for communication becomes lower, which in turn increases the overall residual capacity. This capacity has been used by the system to enhance processor levels (i.e., decrease voltage/frequency) and it results in lower NE. For example, in *Gaussian Elimination* and *Epigenomics* (Figure 7.7a) with $NWR = 85\%$, the NEs returned by SAFLA for $b = 2$ and $b = 4$ are ∼12% and ∼9%, respectively.



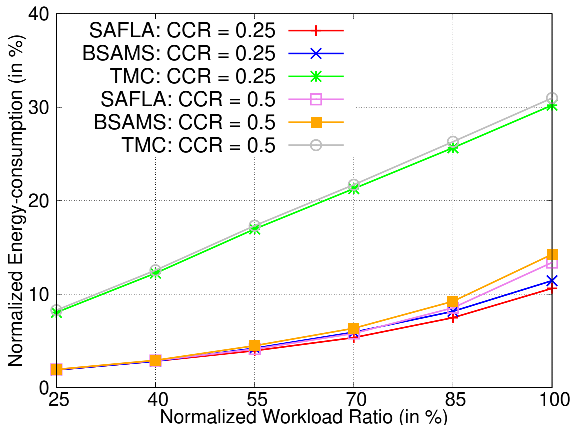
(a) *GE, Epigenomics* **(b)** *GE, Epigenomics, Laplace* **(c)** *GE, Epigenomics, Laplace, Stencil*

Figure 7.7: *Variation in #buses*

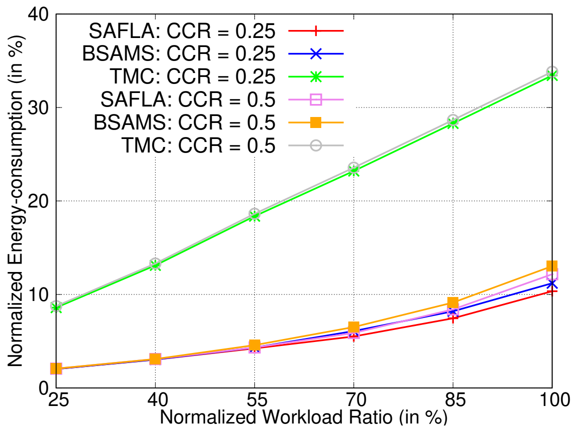
Experiment-3: Effect of variation in CCR: The results of this experiment is depicted in Figure 7.8. Here, CCR is varied from 0.25 to 0.5, while fixing p to 8 and b to 4. It can be observed that the normalized energy-consumption *NE increases with increase in CCR, for any given NWR value.* This is because, with the increase in CCR, the overall contention for message transmission increases, resulting in an overall increase in makespan. This increase in makespan reduces the overall slack time which is used by the system to reduce processor voltages/frequencies. For example, in *Gaussian Elimination* and *Epigenomics* (Figure 7.8a) with $NWR = 85\%$, the NEs returned by SAFLA for $CCR = 0.25$ and $CCR = 0.5$ are ∼7% and ∼9%, respectively.



(a) *GE, Epigenomics*



(b) *GE, Epigenomics, Laplace*



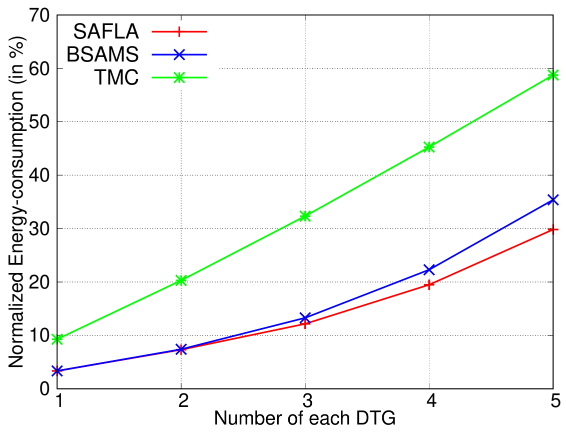
(c) *GE, Epigenomics, Laplace, Stencil*

Figure 7.8: *Effect of variation in CCR*

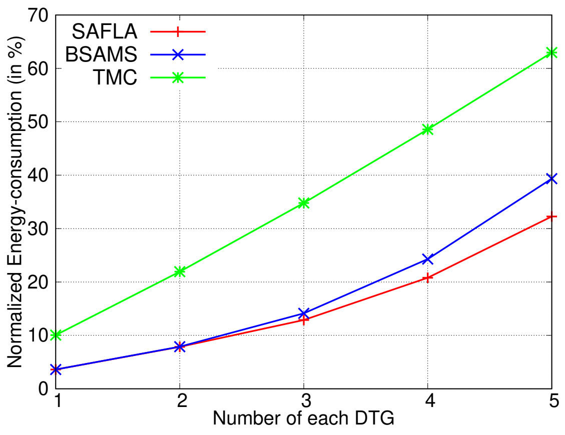
From Experiments-1, 2 and 3, it can be seen that *NE increases with the increase in NWR*. This can be attributed to the fact that with the increase in NWR, the overall execution times also increases. This in turn reduces the overall residual capacity of the system. This decrease in residual capacity restricts processor level enhancement, thus adversely affecting NE. Again, it can be observed that when NWR values are lower than a specific threshold, the NE values for both SAFLA and BSAMS drop to their minimum values (i.e., the processors always operate at their highest level). For example, in *Gaussian Elimination, Epigenomics* (Figure 7.6a) with $NWR = 25\%$ and $p = 4$, the NE value of SAFLA is ∼3%. Further in all scenarios, the proposed algorithm SAFLA outperforms the baseline algorithm BSAMS.

Experiment-4: Variation in the number of PTGs: Figures 7.9a, 7.9b, and 7.9c, show the results of this experiment considering two, three, and four types of application PTGs, respectively. For example in Figure 7.9a, we consider two types of applications, *GE* and *Epigenomics*. The values in the x-axis correspond to the number of applications of each type considered in a given experiment. For example in Figure 7.9a, all points corresponding to the value 4 on the x-axis show results of experiments conducted with four *GE* and four *Epigenomics* applications. It may be noted that for a given figure (say, Figure 7.9a), the individual deadlines and the hyperperiod remain same for all experiments, as the number of application types considered are fixed. For these experiments, the workloads are generated such that it is low when the number of applications are low. The workload imposed on the system increases as the number of applications considered in an experiment becomes higher. In order to generate low workloads for cases when the number of applications are low, we keep deadlines relaxed for each application type. For this purpose, we have fixed the values of t_1 and t_2 to 1.5 and 0.25 (in Equation 7.28), instead of 0.4 and 0.2, as used for the other experiments. Here, we set p to 8, b to 4 and CCR to 0.5. In Figure 7.9, it may be observed that the normalized energy-consumption *NE increases with increase in workload as the number of applications become higher.* For example in *Gaussian Elimination* and *Epigenomics* (Figure 7.9a), with 1 and 5 applications each for *GE* and *Epigenomics*, the NEs returned by SAFLA are ∼3% and ∼30%, respectively.

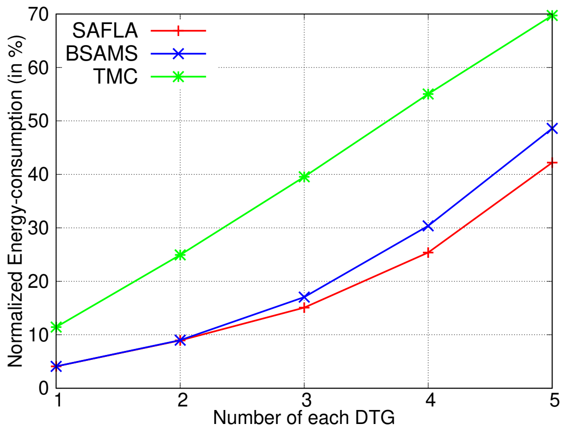
Experiment-5: Running Time Comparison: This experiment compares the



(a) *GE, Epigenomics*



(b) *GE, Epigenomics, Laplace*



(c) *GE, Epigenomics, Laplace, Stencil*

Figure 7.9: *Variation in the number of PTGs*

normalized running times (NRT) of TMC, SAFLA and BSAMS. The results of this experiment are depicted in Table 7.7. Here, p is varied from 4 to 8 while fixing b to 4, CCR to 0.5 and NWR to 100%. It can be observed that *the normalized running times increase as the number of processors becomes higher*. For example, in *GE* (Gaussian Elimination) and *Epigenomics*, the NRTs of SAFLA (3^{rd} column of Table 7.7) at $p = 4$ and $p = 8$ are 11.77 and 17.82 microseconds, respectively. It can also be observed that *the normalized running times increase with increase in the number of PTGs*. For example with $p = 4$, the NRTs of SAFLA for two PTGs (GE and Epigenomic) and four PTGs (GE, Epigenomics, Laplace and Stencil) are 11.77 and 117.4 microseconds, respectively.

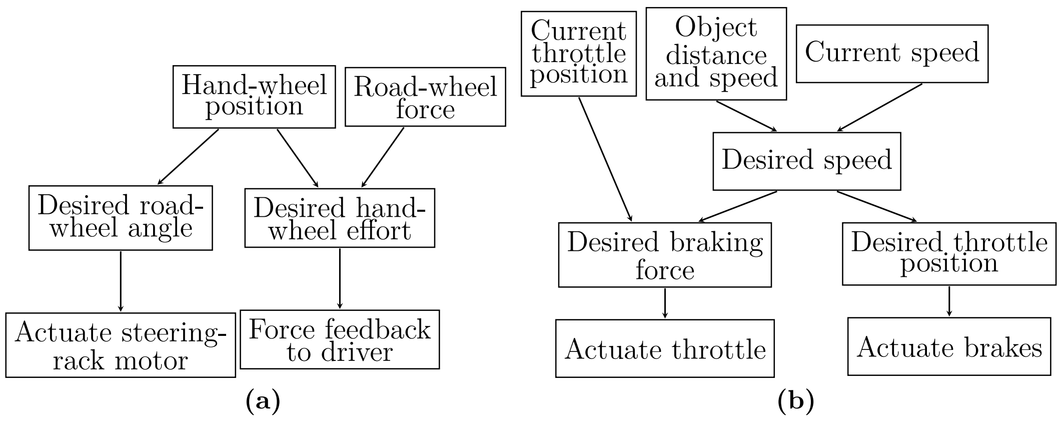
On the other hand, BSAMS having lower time-complexity compared to SAFLA, incurs lower solution generation times in all cases. Similar trends are observed while varying the number of buses.

#Processors	GE, Epigenomics			GE, Epigenomics, Laplace			GE, Epigenomics, Laplace, Stencil		
	TMC	SAFLA	BSAMS	TMC	SAFLA	BSAMS	TMC	SAFLA	BSAMS
4	1.48	11.77	7.44	1.31	36.07	20.87	1.36	117.4	100.44
8	1.86	17.82	8.46	1.72	56.82	20.97	1.73	268.25	100.81

Table 7.7: *Normalized running time (in microseconds)*

7.6 Case Study

To exhibit the practical applicability of the presented strategies to actual designs, we discuss a case study using three automotive control applications, *Electric Power Steering* (EPS), *Adaptive Cruise Controller* (ACC) and *Traction Controller* (TC) [5]. ACC automatically maintains a safe distance between two cars, while EPS provides necessary steering assistance to the driver using an electric motor. The TC helps in actively stabilizing the vehicle so that it can continue in its stipulated path even when road conditions are slippery. Figures 7.10a, 7.10b, and 7.10c depict the block diagrams of EPS, ACC and TC, respectively. The corresponding PTG representations are shown in Figures 7.11a, 7.11b and 7.11c, respectively. For the purpose of this case study, the PTGs are assumed to be executed on a three processor ($P = \{P_1, P_2, P_3\}$) heterogeneous distributed platform interconnected via two heterogeneous buses ($B = \{B_1, B_2\}$). Table 7.1 shows the voltages and frequencies of the processors at different levels. Table 7.8, 7.9 and 7.10 lists the task execution times associated with the processors in P at their highest frequencies and communication times of messages associated with the buses in B. The execution/communication times of the dummy task/message nodes on all processors/buses are set to 0.



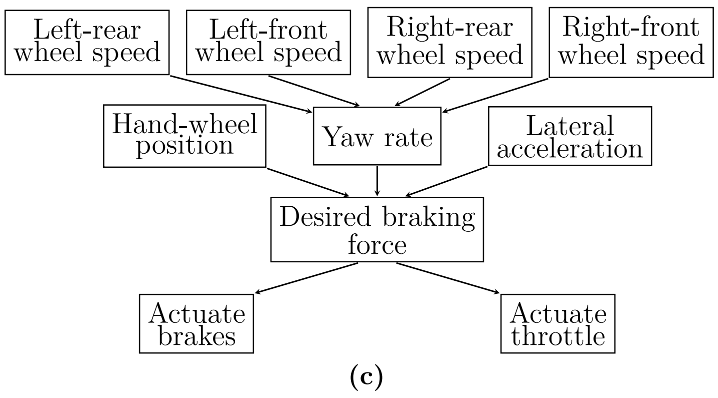


Figure 7.10: *Block diagram: (a) Electric Power Steering (EPS), (b) Adaptive Cruise Controller (ACC), (c) Traction Controller (TC)*

	EPS (G^1; $D^1 = 1500\mu s$)										
	T_1^1	T_2^1	T_3^1	T_4^1	T_5^1	T_6^1	M_1^1	M_2^1	M_3^1	M_4^1	M_5^1
P_1	150	170	300	250	150	100	-	-	-	-	-
P_2	130	200	250	330	180	140	-	-	-	-	-
P_3	170	130	400	280	120	130	-	-	-	-	-
B_1	-	-	-	-	-	-	130	250	300	250	170
B_2	-	-	-	-	-	-	200	190	200	300	200

Table 7.8: *Execution and communication times: Electric Power Steering (EPS)*

We have employed SAFLA to generate schedules for PTGs EPS, ACC and TC with the objective of minimizing overall energy consumption. We summarize below the im-

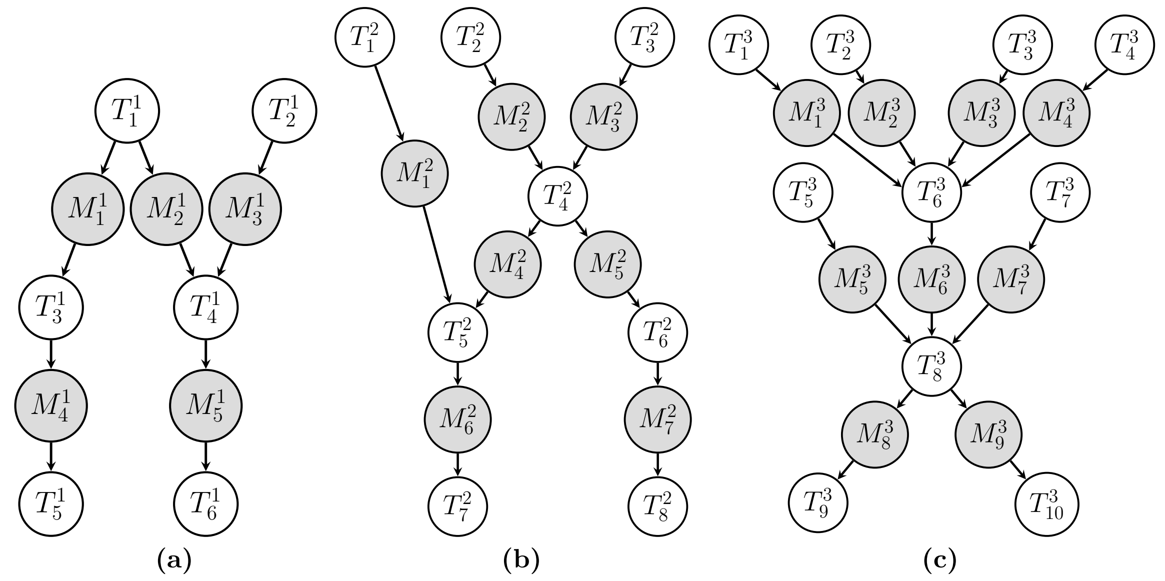


Figure 7.11: *PTG representation: (a) Electric Power Steering (EPS), (b) Adaptive Cruise Controller (ACC), (c) Traction Controller (TC)*

	ACC (G^2; $D^2 = 3000\mu s$)														
	T_1^2	T_2^2	T_3^2	T_4^2	T_5^2	T_6^2	T_7^2	T_8^2	M_1^2	M_2^2	M_3^2	M_4^2	M_5^2	M_6^2	M_7^2
P_1	170	300	150	300	250	200	150	200	-	-	-	-	-	-	-
P_2	200	150	250	320	190	240	170	300	-	-	-	-	-	-	-
P_3	150	220	170	270	270	270	160	210	-	-	-	-	-	-	-
B_1	-	-	-	-	-	-	-	-	150	330	320	220	140	200	300
B_2	-	-	-	-	-	-	-	-	200	200	350	300	210	140	180

Table 7.9: *Execution and communication times: Adaptive Cruise Controller (ACC)*

portant observations associated with the obtained schedules for SAFLA (Figure 7.12):

- *Constraint Satisfaction*: (i) For any given resources (processor/bus), no two tasks/ messages commence execution/transmission at the same time, (ii) resource constraints have always been satisfied, (iii) constraints related to inter-node dependencies have been adhered to. For example, task $T_{1,3}^{1,1}$ commences execution on processor P_2 at time 0 μs. While being processed on P_2, no task barring $T_{1,3}^{1,1}$ executes on P_1 (thus, satisfying resource usage constraint). As both predecessor and

	TC (G^3; $D^3 = 3000\mu s$)																		
	T_1^3	T_2^3	T_3^3	T_4^3	T_5^3	T_6^3	T_7^3	T_8^3	T_9^3	T_{10}^3	M_1^3	M_2^3	M_3^3	M_4^3	M_5^3	M_6^3	M_7^3	M_8^3	M_9^3
P_1	200	200	200	200	150	300	180	400	150	200	-	-	-	-	-	-	-	-	-
P_2	180	230	210	250	200	350	170	200	190	300	-	-	-	-	-	-	-	-	-
P_3	220	220	240	180	130	310	190	250	140	260	-	-	-	-	-	-	-	-	-
B_1	-	-	-	-	-	-	-	-	-	-	150	220	300	250	170	240	180	210	120
B_2	-	-	-	-	-	-	-	-	-	-	200	190	200	350	190	150	220	170	200

Table 7.10: *Execution and communication times: Traction Controller (TC)*

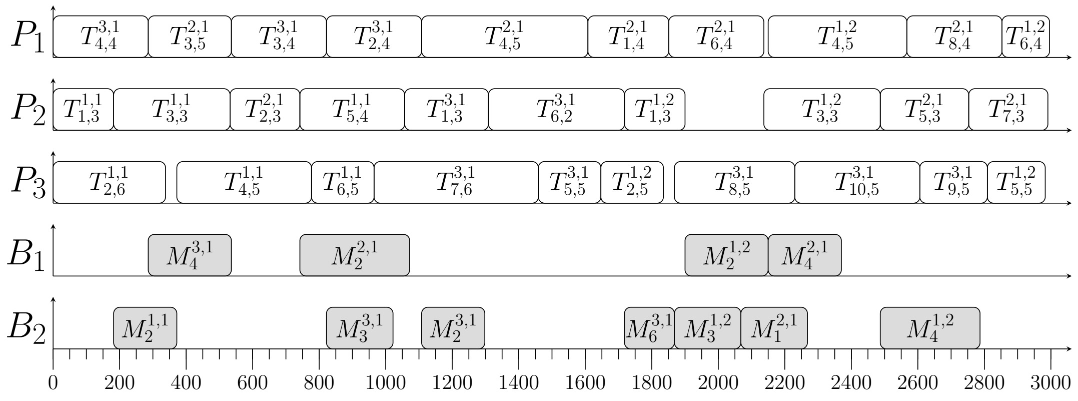


Figure 7.12: *Case study: Gantt chart representation of the schedule*

successor task nodes $T_{1,3}^{1,1}$ and $T_{3,3}^{1,1}$ of $M_1^{1,1}$ are allocated on to the same processor P_2, no communication overhead ($M_1^{1,1}$) is incurred for transmission of the output of $T_{1,3}^{1,1}$ to $T_{3,3}^{1,1}$. The message node $M_2^{1,1}$ start its transmission at time 182 μs on bus B_2 after the parent task node $T_{1,3}^{1,1}$ complete its execution, satisfying inter-node dependencies.

- *Heterogeneity Modeling*: Each node of a PTG at each iteration consumes an appropriate execution/transmission time according to the processor/bus allocated to it. For example, task $T_{4,5}^{1,1}$ of PTG G^1 (EPS) at iteration $q = 1$ is scheduled on processor P_3 and execute for 405 μs ($= \left\lceil \frac{e^1_{4,3,1} \times f_{3,1}}{f_{3,5}} \right\rceil = \left\lceil \frac{280 \times 2.6}{1.8} \right\rceil$).
- Energy consumed by the schedule generated by SAFLA to execute all the DTGs (in Figure 7.11) over the length of the hyperperiod is 15686.2 units. This value

corresponds to a reduction of 7016.89 compared to the TMC schedule (that SAFLA takes as input), which consumes 22703.09 units of energy, and the NE becomes 73.44%.

7.7 Summary

This chapter considered the problem of designing heterogeneous processor-shared bus co-scheduling strategies for a given set of independent periodic applications, each of which is modelled as a PTG, with the objective to minimize system level energy dissipation. An ILP based optimal scheme called ILP-ES is proposed to solve the scheduling problem at hand. However, ILP-ES is associated with very high computational complexity and is not scalable even for small problem sizes. Therefore, we proposed an efficient but low-overhead heuristic strategy called SAFLA which consumes drastically lower time and space complexities while generating good and acceptable solutions. The presented experimental results show the effectiveness and practical applicability of the proposed heuristic SAFLA. Finally, we presented a case study using a set of control applications from automotive systems. The next chapter summarizes the contributions of this dissertation and discusses a few possible extensions to this research.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the contributions of this dissertation and outline some possible directions for future work.

8.1 Discussion and Summarization

This dissertation presents a few novel real-time optimal/heuristic offline task-message co-scheduling strategies for safety-critical CPSs consisting of various types of task and execution platform scenarios. We now present brief summaries of these works in more detail.

CPSs, including those in the automotive domain, are often designed by assigning to each task an appropriate criticality-based reward value that is acquired by the system on its successful execution. Additionally, each task may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher rewards for the system. In our first contributory chapter (Chapter 3), we have presented co-scheduling strategies for a set of independent periodic tasks executing on a bus-based homogeneous multiprocessor system, with the objective of maximizing system level QoS. The problem is posed as a *Multi-dimensional Multiple-Choice Knapsack formulation* (MMCKP). We present a *Dynamic Programming* (DP) solution (called MMCKP-DP) for this problem. Although DP delivers optimal solutions, it suffers from significantly high overheads (in terms of running time and main memory consumption) which steeply increase as the number of tasks, service-levels, processors and buses in the system grows, and severely restricts the scalability of the strategy. Such large time and space overheads are often not affordable, especially when multiple quick design iterations are needed during design space exploration and/or powerful server systems are not available at the designer's disposal. Therefore, in addition to the optimal solution approach, we have proposed an efficient, low-overhead, greedy but balanced heuristic strategy called ALOLA which consumes drastically lower time and space complexities while generating good and acceptable solutions that do not significantly deviate from the optimal solutions. Our simulation based experimental evaluation shows that even on moderately large systems consisting of 90 tasks with 5 service-levels each, 16 processors and 4 buses, while MMCKP-DP incurs a run-time of more than 1 hour 20 minutes and approximately 68 GB main memory, ALOLA takes only about 196 μs (speedup of the order of 10^6 times) and less than 1 MB of memory. Moreover, while being fast, ALOLA is also efficient being able to control performance degradations to at most 13% compared to the optimal results produced by MMCKP-DP.

In Chapter 3, we have considered the problem of scheduling real-time independent task sets running on homogeneous multiprocessor systems. However, Continuous demands for higher performance and reliability within stringent resource budgets is driving a shift from homogeneous to heterogeneous processing platforms for the implementation of today's CPSs. These CPSs are often distributed in nature and typically represented as PTGs due to the complex interactions between their functional components. In Chapters 4 and 5, we have considered the problem of scheduling a real-time system modeled as a PTG, where tasks may have multiple implementations designated as service-levels, with higher service-levels producing more accurate results and contributing to higher *rewards*/QoS for the system. In this work, we have proposed the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for scheduling a real-time PTG executing on a distributed platform consisting of a set of fully-connected heterogeneous processing elements. First, we have developed an ILP based optimal solution strategy namely, ILP-SATC, which follows an intuitive design flow and represents all specifications related to resource, timing and dependency, through a systematic set of constraints. However, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In order to improve scalability, a second strategy namely, ILP-SANC has been designed. ILP-SANC is based on the *non-overlapping approach* [9] which sets constraints and variables in such a way that no two tasks executing on the same processor overlap in time. Further in ILP-SANC, the total number of constraints required to compute a schedule for a PTG becomes independent of the deadline of a given PTG, which helps to control complexity of the proposed scheme.

Though ILP-SANC shows appreciable improvements in terms of scalability over the ILP-SATC, it still suffers from high computational overheads (in terms of running time) as the number of nodes in a PTG and/or the number of resources, increase. Therefore, we have proposed two low-overhead heuristics (i) G-SAQA and (ii) T-SAQA. Both G-SAQA and T-SAQA internally make use of PEFT [1], a well known PTG scheduling algorithm on heterogeneous multiprocessor systems, to compute a baseline schedule which assumes all task nodes to be at their base service-levels. Since PEFT attempts to minimize schedule length, the resulting schedule length may be marked by unutilized slack time before deadline. G-SAQA only considers global slack ($= Deadline - PEFT\ makespan$) to upgrade service-levels of tasks in the PTG. However, a closer look at the PEFT schedule reveals that there exists gaps within the scheduled nodes of the PTG which could be used along with the global slack to achieve better performance in terms of service-levels and delivered rewards, compared to G-SAQA. It may also be possible to consolidate multiple small gaps within the PEFT schedule into larger consolidated slacks which may be used to further improve performance in terms of achieved rewards. Therefore, the *total slack* available with a task at any given time comprises of the global slack along with the maximum consolidated inter-node gap between the task and its successor on its assigned processor in the PEFT schedule. With the above insights on the total task-level slacks available in a PTG, another heuristic namely, T-SAQA is proposed with the objective of achieving better performance compared to G-SAQA. Our simulation based experimental results show that both the heuristic schemes (G-SAQA and T-SAQA) are about ∼10^6 times faster on an average than the optimal strategy ILP-SANC. The results show that both T-SAQA and G-SAQA returns at most ∼30% and ∼45% less rewards than ILP-SANC, respectively. In all cases, T-SAQA outperforms G-SAQA in terms of reward maximization while T-SAQA consumes more running time than G-SAQA.

The PTG scheduling technique considered in Chapters 4 and 5 assumed a fully connected heterogeneous platform. Assumption of a fully connected platform helps to avoid the problem of resource contention, as is the case when the system is assumed to be associated with shared data transmission channels. However, it may be appreciated that shared bus networks form a very commonly used communication architecture in CPSs [41, 42]. Therefore, Chapter 6 extends the problem of scheduling PTGs on fully-connected platforms, to CPS systems where the processors are connected through a limited number of bus based shared communication channels. In this work, we have presented the design of ILP based optimal scheduling strategies as well as low-overhead heuristic schemes for the scheduling of real-time PTGs executing on a distributed platform consisting of a set of heterogeneous processing elements interconnected by heterogeneous shared buses. We present two ILP based strategies namely, ILP-ETR and ILP-NC. The design philosophies of ILP-ETR and ILP-NC are similar to the formulations ILP-SATC (Chapter 4, Section 4.3) and ILP-SANC (Chapter 4, Section 4.4) respectively, proposed in Chapter 4 for PTG scheduling on fully connected heterogeneous platform. Although ILP-ETR follows an intuitive design flow, its scalability is limited primarily due to the explicit manipulation of task mobilities between their earliest and latest start times. In comparison, ILP-NC being based on the *non-overlapping approach* [9], exhibits significantly better scalability. Experimental results show that ILP-ETR takes ∼5 hours to compute the schedule of a PTG with ∼20 nodes executing on a system with 4 processor and 2 buses with the deadline of the PTG set to the optimal makespan. On the other hand, ILP-NC takes only ∼12 secs to compute a schedule for the same. In addition to the two optimal ILP based approaches, we have designed a fast and efficient heuristic strategy namely, CC-TMS for the problem at hand. CC-TMS is based on a *list scheduling* based heuristic approach to co-schedule task and message nodes in a real-time PTG executing on a distributed system consisting of a set of heterogeneous processors interconnected by heterogeneous shared buses. To evaluate the performance of CC-TMS with respect to optimal solutions, we have defined a metric called *Makespan Ratio* as follows:

$$Makespan\ Ratio = \frac{Optimal\ Makespan}{Heuristic\ Makespan} \times 100 \tag{8.1}$$

Extensive simulation based experimental results show that CC-TMS achieves 97% and 58% (*Makespan Ratio*) in the best and worst case scenarios, respectively.

The works done in Chapters 4, 5 and 6 dealt with the co-scheduling of a single task graph on heterogeneous distributed platform. In Chapter 7, our last contributory chapter, we have endeavoured towards the design of processor-shared bus co-scheduling strategies for a given set of independent periodic applications, each of which is modelled as a PTG. In particular, we have developed an ILP based optimal and heuristic strategy for the mentioned system model, whose objective is to minimize system level dynamic energy dissipation. To achieve energy savings, the processors in the system are assumed to be DVFS enabled and thus, the operating frequencies of these processors can be dynamically reconfigured to a discrete set of alternative voltage/frequency-levels at run-time. The proposed ILP based optimal scheme called ILP-ES is associated with very high computational complexity and is not scalable even for small problem sizes. Therefore, we have proposed an efficient but low-overhead heuristic strategy called SAFLA which consumes drastically lower time and space complexities while generating good and acceptable solutions. The SAFLA algorithm starts by using an efficient co-scheduling algorithm TMC which actually extend the CC-TMS algorithm (Chapter 6, Algorithm 4) to schedule multiple periodic PTGs executing on a shared bus-based heterogeneous distributed platform. This schedule is generated assuming all processors to be running at their highest frequency for the entire duration of the schedule. Now, the available slack associated with each task node is used to enhance the tasks' assigned voltage/frequency-levels in an endeavour to minimize energy dissipation while retaining task-to-processor/message-to-bus mappings as provided by TMC. Experimental results show that SAFLA is an effective scheduling scheme and delivers handsome savings in terms of lower energy consumption in most practical scenarios.

8.2 Future Works

The work presented in this thesis leaves several open directions and there is ample scope for future research in this area. In this section, we present few such future perspectives.

- **Deployment of ALOLA on real communication frameworks:** The proposed scheme ALOLA in Chapter 3, is a generic processor-bus co-scheduling strategy which has not been designed with any particular protocol in mind. As part of our future work, we plan to conduct experiments on a real CAN-based Distributed *Cyber-Physical Systems* (CPSs) test bed. We also plan to design a co-scheduling strategy for application with futuristic communication frameworks such as Time Sensitive Networking (TSN; IEEE 802.1Qbv; [57–59]). TSN is a switched Ethernet based protocol which can support precise real-time communication. The proposed co-scheduling strategy must be significantly extended for employment with protocols such as TSN. TSN allows each message to be split into multiple Ethernet frames with each frame being possibly transmitted through distinct paths. Each such path may pass through multiple intermediate switches. For effective real-time data transmission, the necessary output ports of each such switch must be appropriately scheduled so that all frames of all messages reach their destinations before the end of their respective periods. Thus here, although the processor schedule can still be similar to ALOLA, the corresponding communication schedule may be considerably more involved and will be taken up as a future work.

- **Co-scheduling dynamic task sets in the presence of persistent tasks:** Chapter 3 addresses the problem of co-scheduling a set of independent periodic tasks with multiple service-levels, executing on a bus-based homogeneous multiprocessor system. As solution approaches, we have devised both optimal and heuristic static offline scheduling schemes. However, CPSs like automotive systems may consist of both safety critical and non-safety critical applications and these applications may be represented as real-time independent tasks or PTGs. Safety critical tasks like anti-lock braking system, fuel injection, chassis control, traction control, etc., are persistent in nature. These tasks are triggered at the start of the system and continue running periodically till the system stops. On the other hand, non-safety critical tasks like air conditioning system, power window, infotainment system, etc., are non-persistent in nature. These tasks are triggered dynamically at run time and terminate after execution of one or more instances. These tasks may have multiple QoS levels and may be executed on homogeneous or heterogeneous platform which are centralized or distributed. Here, we plan to address the following problem: *Given a set of persistent and dynamic tasks (which may arrive at any time when the system is under operation) the objective is to generate a real-time schedule which allows guaranteed execution of all persistent tasks, while maximizing the number of dynamic tasks that can be incorporated.* As in many of our earlier chapters, we plan to employ a heterogeneous shared bus-based distributed system as the execution platform.

- **Design strategies to enhance run-time resource utilization:** The static resource allocation strategies developed as part of this thesis are based on wort-case resource usage estimates of applications. These schemes may be prone to significant performance degradations when actual resource usage of the applications are significantly less than their worst-case estimates. Research in the last few years has revealed that the mapping and scheduling mechanisms in these scenarios may need to be both static and dynamic. The static part first provides a cost-optimal constraint-driven scheduling, allocation and assignment of various functional components of all the available resources; this step should not be intended to generate a single solution, but to generate an execution plan consisting of a set of optional solutions which the dynamic part can use to take decisions according to different run-time conditions. The dynamic part should be fast and must efficiently do a

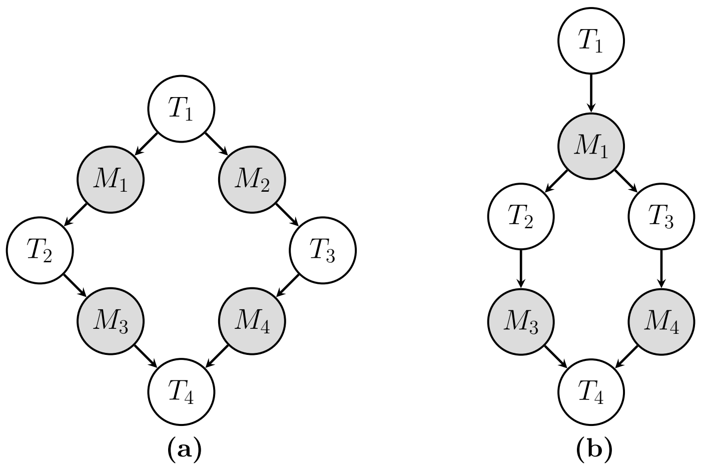


Figure 8.1: *(a) PTG with unicast, (b) PTG with multicast*

combined architecture load and power-aware run-time scheduling according to the execution plan provided by the static part, such that real-time constraints are met. However, determination of the exact offline-online strategies to be employed for specific system scenarios at hand is non-trivial and demands considerable research.

- **Multicast bus messaging in PTG scheduling scenarios for improved resource utilization:** Chapters 6 and 7 addressed the problem of co-scheduling real-time PTGs running on shared bus-based heterogeneous multiprocessor systems. These works used *unicast* communication to send a message from one task to another. However, a task may need to send the same message to multiple task nodes. For example in Figure 8.1a, task T_1 sends messages M_1 and M_2 to task nodes T_2 and T_3, respectively. However, it may happen that both the message nodes M_1 and M_2 contain the same data. In this scenario, the task T_1 can send only one multicast message (M_1) as shown in Figure 8.1b instead of two separate messages. Needless to say that this will reduce the overall communication load leading to potentially improved scheduled performance. However, multicast messaging within a PTG scheduling scenario is a non-trivial extension and requires considerable research. We intended to design both optimal and heuristic solution strategies for this problem.

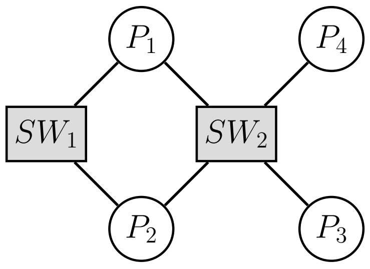


Figure 8.2: *Example of an ad hoc network; Here, P_1, P_2, P_3, P_4 are processors and SW_1, SW_2 are switches*

- **Adaptation of PTG scheduling to more generalized ad hoc point-to-point networks:** The PTG scheduling problems presented in Chapters 6 and 7 considered shared bus-based heterogeneous multiprocessor systems. In those works, the assumption was that all processors are connected to all buses. In general, processors are connected through a network, and a processor may be connected to a subset of buses. To schedule real-time PTGs on multiprocessor systems connected through an ad hoc point-to-point network, the transmission of a message have to complete within a bounded time. A message may have multiple paths to reach the destination from a given source and each such path may take a different amount of time. The co-scheduling of tasks and messages for PTGs on multiprocessor systems interconnected to an ad hoc point-to-point network, requires ample research and will be taken up as a future work.

8.3 Disseminations out of this Work

Journal Papers

Published/Submitted

1. **Sanjit Kumar Roy**, Rajesh Devaraj, Arnab Sarkar, Sayani Sinha and Kankana Maji. "Contention-aware optimal scheduling of real-time precedence-constrained task graphs on heterogeneous distributed systems." *Elsevier Journal of Systems Architecture (JSA)*, Volume 105, 2020.

2. **Sanjit Kumar Roy**, Rajesh Devaraj and Arnab Sarkar. "Contention Cognizant Scheduling of Task Graphs on Shared Bus based Heterogeneous Platforms." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)*, Volume 41, Pages 281-293, 2021.

3. **Sanjit Kumar Roy**, Rajesh Devaraj, Arnab Sarkar and Debabrata Senapati. "SLAQA: Quality-level Aware Scheduling of Task Graphs on Heterogeneous Distributed Systems." *ACM Transactions on Embedded Computing Systems (ACM TECS)*, Volume 20, Pages 1-31, 2021.

4. **Sanjit Kumar Roy**, Rajesh Devaraj and Arnab Sarkar. "SAFLA: Scheduling Multiple Real-time Periodic Task Graphs on Heterogeneous Systems." *IEEE Transactions on Computers* (IEEE TC), (Accepted).

Conference Papers

1. **Sanjit Kumar Roy**, Rajesh Devaraj, Arnab Sarkar, Sayani Sinha and Kankana Maji. "Optimal scheduling of precedence-constrained task graphs on heterogeneous distributed systems with shared buses." *IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, Pages 185-192, 2019.

2. **Sanjit Kumar Roy**, Rajesh Devaraj and Arnab Sarkar. "Optimal scheduling of PTGs with multiple service levels on heterogeneous distributed systems." *American Control Conference (ACC)*, Pages 157-162, 2019.

3. **Sanjit Kumar Roy**, Arnab Sarkar and Rahul Gangopadhyay. "Processor and Bus Co-scheduling Strategies for Real-time Tasks with Multiple Service-levels." *IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Pages 21-30, 2021.

References

[1] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014. [Pg.xxii], [Pg.xxviii], [Pg.4], [Pg.6], [Pg.11], [Pg.29], [Pg.35], [Pg.42], [Pg.44], [Pg.70], [Pg.82], [Pg.83], [Pg.86], [Pg.90], [Pg.94], [Pg.96], [Pg.103], [Pg.105], [Pg.112], [Pg.138], [Pg.141], [Pg.152], [Pg.175], [Pg.203]

[2] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013. [Pg.xxii], [Pg.xxiii], [Pg.xxiv], [Pg.82], [Pg.83], [Pg.103], [Pg.143], [Pg.187], [Pg.188]

[3] C. Bolchini and A. Miele, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2489–2502, 2013. [Pg.xxii], [Pg.87], [Pg.88]

[4] A. Benoit, M. Hakem, and Y. Robert, "Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems," *Parallel Computing*, vol. 35, no. 2, pp. 83–108, 2009. [Pg.xxii], [Pg.xxiii], [Pg.xxiv], [Pg.103], [Pg.143], [Pg.187], [Pg.188]

[5] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Dependable communication synthesis for distributed embedded systems," *Reliability Engineering & System Safety*, vol. 89, no. 1, pp. 81–92, 2005. [Pg.xxiii], [Pg.87], [Pg.116], [Pg.155], [Pg.156], [Pg.196]

[6] H. Topcuoglu *et al.*, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002. [Pg.xxiii], [Pg.xxiv], [Pg.4], [Pg.6], [Pg.29], [Pg.35], [Pg.42], [Pg.43], [Pg.44], [Pg.94], [Pg.138], [Pg.141], [Pg.143], [Pg.152], [Pg.175], [Pg.187], [Pg.188]

[7] T. Abdelzaher, E. M. Atkins, and K. G. Shin, “QoS negotiation in real-time systems and its application to automated flight control,” *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1170–1183, 2000. [Pg.xxvii], [Pg.64], [Pg.65]

[8] J. Jiang, Y. Lin, G. Xie, L. Fu, and J. Yang, “Time and energy optimization algorithms for the static scheduling of multiple workflows in heterogeneous computing system,” *Journal of Grid Computing*, vol. 15, no. 4, pp. 435–456, 2017. [Pg.xxviii], [Pg.42], [Pg.43], [Pg.44], [Pg.162], [Pg.163], [Pg.166], [Pg.190]

[9] S. Venugopalan and O. Sinnen, “ILP formulations for optimal task scheduling with communication delays on parallel systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 142–151, 2015. [Pg.3], [Pg.5], [Pg.10], [Pg.14], [Pg.29], [Pg.35], [Pg.42], [Pg.44], [Pg.77], [Pg.131], [Pg.203], [Pg.204]

[10] “CPLEX Optimizer: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer.” [Online]. Available: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer [Pg.4], [Pg.76], [Pg.84], [Pg.105], [Pg.130], [Pg.145]

[11] G. Xie, R. Li, and K. Li, “Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems,” *Journal of Parallel and Distributed Computing*, vol. 83, pp. 1–12, 2015. [Pg.4], [Pg.6], [Pg.29], [Pg.35], [Pg.42], [Pg.44], [Pg.94], [Pg.138], [Pg.175]

[12] R. Bajaj and D. P. Agrawal, “Improving scheduling of tasks in a heterogeneous environment,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 107–118, 2004. [Pg.4], [Pg.6], [Pg.29], [Pg.35], [Pg.42]

[13] S. Bansal, P. Kumar, and K. Singh, “Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs,” *Journal of Parallel*

and Distributed Computing, vol. 65, no. 4, pp. 479–491, 2005. [Pg.4], [Pg.6], [Pg.29], [Pg.35], [Pg.42]

[14] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011. [Pg.4], [Pg.21], [Pg.40]

[15] H. Baek, J. Lee, and I. Shin, "Multi-level contention-free policy for real-time multiprocessor scheduling," *Journal of Systems and Software*, vol. 137, pp. 36–49, 2018. [Pg.4], [Pg.40]

[16] H. S. Chwa, H. Back, J. Lee, K.-M. Phan, and I. Shin, "Capturing urgency and parallelism using quasi-deadlines for real-time multiprocessor scheduling," *Journal of Systems and Software*, vol. 101, pp. 15–29, 2015. [Pg.4], [Pg.40]

[17] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973. [Pg.5], [Pg.40]

[18] B. Andersson and J. Jonsson, "The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%," in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on.* IEEE, 2003, pp. 33–40. [Pg.5], [Pg.40]

[19] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996. [Pg.5], [Pg.40], [Pg.44]

[20] J. H. Anderson and A. Srinivasan, "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks," in *Real-Time Systems, 13th Euromicro Conference on, 2001.* IEEE, 2001, pp. 76–85. [Pg.5], [Pg.40]

[21] ——, "Early-release fair scheduling," in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000.* IEEE, 2000, pp. 35–43. [Pg.5], [Pg.40], [Pg.44]

[22] D. Zhu, D. Mossé, and R. Melhem, "Multiple-resource periodic scheduling problem: how much fairness is necessary?" in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003.* IEEE, 2003, pp. 142–151. [Pg.5], [Pg.40]

[23] A. Khemka and R. Shyamasundar, "An optimal multiprocessor real-time scheduling algorithm," *Journal of parallel and distributed computing*, vol. 43, no. 1, pp. 37–45, 1997. [Pg.5], [Pg.40]

[24] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06).* IEEE, 2006, pp. 101–110. [Pg.5], [Pg.40]

[25] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *2011 IEEE 32nd Real-Time Systems Symposium.* IEEE, 2011, pp. 104–115. [Pg.5], [Pg.40]

[26] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on.* IEEE, 2010, pp. 3–13. [Pg.5], [Pg.10], [Pg.40], [Pg.41], [Pg.44]

[27] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer, 2011, vol. 24. [Pg.5], [Pg.6], [Pg.21], [Pg.41]

[28] X. Wang, Z. Li, and W. M. Wonham, "Dynamic multiple-period reconfiguration of real-time scheduling based on timed DES supervisory control," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 101–111, 2016. [Pg.5], [Pg.41]

[29] G. Srinivasa Prasanna and B. R. Musicus, "Generalised multiprocessor scheduling using optimal control," in *3rd annual symposium on Parallel algorithms and architectures.* ACM, 1991, pp. 216–228. [Pg.5], [Pg.41], [Pg.44]

[30] G. S. Prasanna and B. R. Musicus, "Generalized multiprocessor scheduling and applications to matrix computations," *IEEE Transactions on Parallel and Distributed systems*, vol. 7, no. 6, pp. 650–664, 1996. [Pg.5], [Pg.42]

[31] J. Liu, Q. Zhuge, S. Gu, J. Hu, G. Zhu, and E. H.-M. Sha, "Minimizing system cost with efficient task assignment on heterogeneous multicore processors considering time constraint," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2101–2113, 2014. [Pg.5]

[32] H. Kanemitsu, M. Hanada, and H. Nakazato, “Clustering-based task scheduling in a large number of heterogeneous processors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3144–3157, 2016. [Pg.5], [Pg.29], [Pg.35], [Pg.42], [Pg.44]

[33] P.-C. Hsiu, C.-K. Hsieh, D.-N. Lee, and T.-W. Kuo, “Multilayer bus optimization for real-time embedded systems,” *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1638–1650, 2012. [Pg.5], [Pg.42], [Pg.44]

[34] M.-Y. Wu and D. D. Gajski, “Hypertool: A programming aid for message-passing systems,” *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990. [Pg.6], [Pg.42]

[35] T. C. Hu, “Parallel sequencing and assembly line problems,” *Operations research*, vol. 9, no. 6, pp. 841–848, 1961. [Pg.6], [Pg.42]

[36] C. Krishna, “Fault-tolerant scheduling in homogeneous real-time systems,” *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 48, 2014. [Pg.7]

[37] M. Chetto, “Optimal scheduling for real-time jobs in energy harvesting computing systems,” *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 2, pp. 122–133, 2014. [Online]. Available: https://doi.org/10.1109/TETC.2013.2296537 [Pg.7]

[38] G. Raravi, B. Andersson, V. Nélis, and K. Bletsas, “Task assignment algorithms for two-type heterogeneous multiprocessors,” *Real-Time Systems*, vol. 50, no. 1, pp. 87–141, 2014. [Pg.7]

[39] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, “A survey of approaches combining safety and security for industrial control systems,” *Reliability Engineering & System Safety*, vol. 139, pp. 156–178, 2015. [Pg.7]

[40] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, “Energy-aware scheduling for real-time systems: A survey,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, pp. 1–34, 2016. [Pg.7]

[41] S. Chakraborty, M. A. Al Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu, “Automotive cyber–physical systems: A tutorial introduction,” *IEEE Design & Test*, vol. 33, no. 4, pp. 92–108, 2016. [Pg.13], [Pg.36], [Pg.121], [Pg.204]

[42] T. Mitra, J. Teich, and L. Thiele, "Time-critical systems design: A survey," *IEEE Design & Test*, vol. 35, no. 2, pp. 8–26, 2018. [Pg.13], [Pg.36], [Pg.121], [Pg.204]

[43] G. Nelissen, "Efficient optimal multiprocessor scheduling algorithms for real-time systems," Ph.D. dissertation, Université libre de Bruxelles, 2012. [Pg.19]

[44] C. Fidge, "Real-time scheduling theory," 2002. [Pg.27]

[45] D. Gislason, *Zigbee wireless networking.* Newnes, 2008. [Pg.29]

[46] B. Chattopadhyay and S. Baruah, "A Lookup-Table Driven Approach to Partitioned Scheduling," in *Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 257–265. [Pg.40]

[47] G. Xie, X. Xiao, H. Peng, R. Li, and K. Li, "A survey of low-energy parallel scheduling algorithms," *IEEE Transactions on Sustainable Computing*, 2021. [Pg.42]

[48] G. Xie, J. Jiang, Y. Liu, R. Li, and K. Li, "Minimizing energy consumption of real-time parallel applications using downward and upward approaches on heterogeneous systems," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1068–1078, 2017. [Pg.42], [Pg.44]

[49] J. Huang, R. Li, J. An, H. Zeng, and W. Chang, "A dvfs-weakly-dependent energy-efficient scheduling approach for deadline-constrained parallel applications on heterogeneous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. [Pg.43]

[50] G. Xie, G. Zeng, J. Jiang, C. Fan, R. Li, and K. Li, "Energy management for multiple real-time workflows on cyber–physical cloud systems," *Future Generation Computer Systems*, vol. 105, pp. 916–931, 2020. [Pg.43], [Pg.44]

[51] S. Liden, "The evolution of flight management systems," in *Digital Avionics Systems Conference, 1994. 13th DASC., AIAA/IEEE.* IEEE, 1994, pp. 157–169. [Pg.63]

[52] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll, "Task period selection to minimize hyperperiod," in *ETFA2011.* IEEE, 2011, pp. 1–4. [Pg.65]

[53] T. Chantem, X. Wang, M. D. Lemmon, and X. S. Hu, “Period and deadline selection for schedulability in real-time systems,” in *2008 Euromicro Conference on Real-Time Systems*, 2008, pp. 168–177. [Pg.65]

[54] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, “Period optimization for hard real-time distributed automotive systems,” in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 278–283. [Pg.65]

[55] I. Ripoll and R. Ballester-Ripoll, “Period selection for minimal hyperperiod in periodic task systems,” *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1813–1822, 2012. [Pg.65]

[56] R. van Glabbeek and P. Höfner, “Split, send, reassemble: A formal specification of a can bus protocol stack,” *arXiv preprint arXiv:1703.06569*, 2017. [Pg.66]

[57] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, “Scheduling real-time communication in ieee 802.1 qbv time sensitive networks,” in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192. [Pg.66], [Pg.206]

[58] R. S. Oliver, S. S. Craciunas, and W. Steiner, “Ieee 802.1 qbv gate control list synthesis using array theory encoding,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 13–24. [Pg.66], [Pg.206]

[59] TSN, “Time-Sensitive Networking (TSN),” *https://en.wikipedia.org/wiki/Time-Sensitive_Networking*, 2012. [Pg.66], [Pg.206]

[60] G. Xie, J. Jiang, Y. Liu, R. Li, and K. Li, “Minimizing energy consumption of real-time parallel applications using downward and upward approaches on heterogeneous systems,” *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1068–1078, 2017. [Pg.82], [Pg.103]

[61] K. Li, “Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers,” *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1668–1681, 2012. [Pg.166]

[62] Y. C. Lee and A. Y. Zomaya, “Energy conscious scheduling for distributed computing systems under different operating conditions,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1374–1381, 2010. [Pg.166]

[63] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li, “An energy-efficient task scheduling algorithm in dvfs-enabled cloud environment.” *J. Grid Comput.*, vol. 14, no. 1, pp. 55–74, 2016. [Pg.166]

[64] Q. Huang, S. Su, J. Li, P. Xu, K. Shuang, and X. Huang, “Enhanced energy-efficient scheduling for parallel applications in cloud,” in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 2012, pp. 781–786. [Pg.166]

[65] H. Djigal, J. Feng, J. Lu, and J. Ge, “Ippts: An efficient algorithm for scientific workflow scheduling in heterogeneous computing systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1057–1071, 2021. [Pg.175]