Efficient Techniques for Scheduling DAG Applications in Distributed Environments

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
Debabrata Senapati

Under the guidance of
Dr. Arnab Sarkar and Dr. Chandan Karfa

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
July, 2023

Abstract

A Cyber-Physical System (*CPS*) integrates two sub-systems, a cyber sub-system and a physical sub-system. The cyber sub-system is often a heterogeneous distributed computing system that executes applications for regulating mechanisms associated with the physical sub-system, typically consisting of electro-mechanical components. Real-Time Cyber-Physical Systems (*RT-CPSs*) are characterized by their ability to respond to events that may happen in their operating environment within stipulated time bounds. The accuracy of these systems depends not only on the delivered results but also on their completion times. Applications in today's RT-CPSs are often represented by Directed Acyclic Graphs (*DAGs*) due to their distributed nature and complex interactions among component functionalities. In such DAGs, nodes represent tasks associated with the application, while edges denote inter dependencies among tasks. To meet functionality specific high-performance demands, these DAGs are often implemented on heterogeneous RT-CPS platforms where, (i) the same task may exhibit different execution time requirements on different processors, and (ii) inter-task messages containing the same amount of data may incur distinct transmission times on the different communication channels, due to variations in channel bandwidths. The RT-CPS applications may be aperiodically triggered by an external event or may execute in infinite loops, periodically acquiring data from the environment through sensors at a particular frequency, processing the same, and then producing processed data via actuators.

This dissertation deals with the design of resource allocation mechanisms for DAG-structured applications on heterogeneous distributed RT-CPSs. The thesis which unfolds through the dissertation is as follows: For the mentioned system scenarios, list-based design philosophy is effective towards obtaining low-overhead but efficient scheduling mechanisms for satisfying diverse objectives/constraints related to resource usage efficiency, timeliness, energy, security, temperature, etc. All list scheduling heuristics typically consist of two phases, (i) *Task Prioritization*: for listing tasks in a specific priority order, and (ii) *Task-to-processor mapping*: for allocating the tasks in the order of their priorities on suitable processors and associating with them appropriate execution start times. The contributions of this thesis are categorized into five phases as follows: (i) The first phase focuses on the development of an efficient real-time DAG-scheduling framework which attempts to minimize a *generic penalty function*. The designed penalty function can be amicably adopted towards its deployment in various application domains such as real-time cyber-physical systems like

automotive and avionic systems, cloud computing, smart grids, etc. (ii) In the second phase, we develop a state-space search guided heuristic scheduling algorithm called *HMDS*, whose objective is to minimize *schedule length*. By controlling the nature and extent of state-space exploration, *HMDS* can adapt itself to deliver the best possible solution within a given time bound. (iii) A mechanism for co-scheduling *multiple independent periodic DAG applications* has been devised in the third phase. The objective of the scheduling algorithm is to minimize dissipated energy. (iv) Subsequently, in the fourth phase, a security-aware real-time DAG scheduling strategy has been designed. The scheme maximizes total security utility for a given application having known minimum security strength specifications for its messages. (v) Finally, in the last phase of the dissertation, we have developed a mechanism to construct minimum *makespan* schedules for precedence-constrained task graph applications with known thermal characteristics on a heterogeneous processing platform. The efficacy of the developed scheduling schemes has been extensively evaluated through simulation-based experiments using real-world benchmark task graphs. Prototype real-platform implementations as well as real-world case studies have also been presented to exhibit the practical applicability of the proposed algorithms.

Keywords: Cyber-physical systems (CPS), directed acyclic graph (DAG), real-time systems, distributed systems, heterogeneous platforms, list scheduling, makespan minimization, energy optimization, security-aware, temperature-aware

Declaration

I certify that:

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

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.

Debabrata Senapati

Copyright

Debabrata Senapati

Certificate

This is to certify that this thesis entitled, **"Efficient Techniques for Scheduling DAG Applications in Distributed Environments"**, being submitted by **Debabrata Senapati**, 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 our supervision and guidance. The thesis, in our 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 our knowledge, it has not been submitted elsewhere for the award of the degree.

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

Dr. Chandan Karfa
Associate Professor
Department of Computer
Science and Engineering
IIT Guwahati

Dedicated to

my teachers, parents, wife and daughter

whose knowledge, blessing, love and inspiration paved my path of success

Acknowledgements

My Ph.D. has been a long journey, full of exciting events and at times difficult and exhausting. This thesis would not be completed without the support and goodwill of several people. The following is an attempt at acknowledging them.

I wish to express my deepest gratitude to my supervisors, Dr. Arnab Sarkar and Dr. Chandan Karfa, for their valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from and work with them. Their constant guidance and support paved the way for my development as a research scientist and changed my personality, ability, and nature in many ways. As a researcher, I am continually amazed at their clarity of thought and ability to get a fundamental idea behind a given problem with just a few questions. I benefited immensely from their unique advising style, which combines the correct balance between allowing students to perform independent research versus guiding them to ensure they pick the right topics of interest. It has been an honor to work with them.

Besides my supervisors, I would like to thank other members of my doctoral committee, Prof. Purandar Bhaduri, Prof. Hemangee K. Kapoor, and Dr. John Jose, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives. I want to express my heartful gratitude to the director, the deans, and other management of IIT Guwahati, whose collective efforts have made this institute a place for world-class studies and education. I am thankful to all faculty and staff of the Department of Computer Science and Engineering for extending their cooperation in terms of technical and official support to complete my research work successfully. I am thankful to the anonymous reviewers of my papers for their thoughtful suggestions and feedback.

I would also like to use this opportunity to thank all my colleagues in the Research Scholars' lab. for making the group such a cohesive unit and the lab.

such a cozy place to be in through all these years. Thanks to my friends Swagat, Shakeel, Siva, Gyanendro, Prasen, Aswathy, Sumita and Priyanka for supporting and motivating me to overcome any problems at work or otherwise. The countless discussions and sharing of ideas have improved our research. I am also grateful to all my seniors and juniors, especially Piyoosh, Basina, Hema, Ujjwal, Subrata, Rakesh, Ramanuj, Moustafa, Pradeep, Arijit, Pawan, Dipojjal, Sisir, Chinmaya, Abhijit, Swarup, Anasua, Palas, Surajit, Alakesh, Manju, Nilotpal, Maithilee and many others, for their innumerable technical and non-technical supports. They all made my life at IIT Guwahati a memorable experience. Thanks to Rajesh, Sanjit, Kousik, and Puja for collaborating with me in research during my Ph.D. tenure. I am thankful to my dearest friend Jayanta, whose belief in me, continuous support, and encouragement gave me strength and confidence throughout this journey.

Most importantly, none of this would have been possible without the love and patience of my family. I am grateful to my parents for their unconditional love and consistent support in every event of my life. I am also very fortunate to have wonderful in-laws who gave me confidence and support at every step of my Ph.D. journey. I am forever indebted to my wife, Sasmita Rout. Without her love, sacrifice, constant encouragement and faith in me, none of this would have been possible. She has been incredibly supportive to me through the ups and downs over the past few years. I cannot thank her enough for being such an excellent life partner and friend. Finally, my wholehearted thanks to my daughter Miss Purvi Senapati, who is the ultimate inspiration source.

Before concluding, once again thanks to you all.

Debabrata Senapati

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

ACC	Adaptive Cruise Controller
ACS	Automotive Control System
AFT	Actual Finish Time
AST	Actual Start Time
BCR	Benefit-to-Cost Ratio
CCR	Communication-to-Computation Ratio
CP	Critical Path
CPS	Cyber-Physical System
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DO	Deadline Overshoot
DPM	Dynamic Power Management
DPMRS	DVFS-enabled Periodic Multi-DAG Real-time Scheduler for heterogeneous systems
DTM	Dynamic Thermal Management
DVFS	Dynamic Voltage and Frequency Scaling

EDO	Estimated Deadline Overshoot
EFT	Effective Finish Time
EPS	Electric Power Steering
ERR	Expected Relative Residual-workload
EST	Effective Start Time
FOV	Fields of View
GDES	Global DVFS-enabled Energy-efficient Scheduling
HEFT	Heterogeneous Earliest Finish Time
HMDS	Heterogeneous Makespan-minimizing DAG Scheduler
ISA	Intelligent Surveillance Application
LCM	Least Common Multiple
MEE	Minimum Estimated Energy
MEP	Minimum Estimated Penalty
NDES	Non-DVFS Energy-efficient Scheduling
NE	Normalized Energy-dissipation
NM	Normalized Makespan
NP	Nondeterministic Polynomial time
NPR	Normalized Penalty Ratios
NRT	Normalized Running Time
NSU	Normalised Security Utility
$\mathbf{O_{EFT}}$	Optimistic Effective Finish Time
OFT	Optimistic Finish Time

OV	Overhead
PEFT	Predict Earliest Finish Time
PFT	Predicted Finish Time
PPTS	Predict Priority Task Scheduling
PRESTO	Penalty-aware Real-time Scheduler for Task Graphs on Heterogeneous Platforms
PSLS	Pre-Scheduling-based List Scheduling
PTZ	Pan-Tilt-Zoom
RT-CPS	Real-Time Cyber-Physical System
SCT	Schedule Completion Time
SHIELD	Security-aware Scheduling for Real-time DAGs on Heterogeneous Systems
SLR	Schedule Length Ratio
TC	Traction Controller
TETT	Task Execution Time and final Temperature
TF	Throttling Factor
TMDS	Temperature-aware Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems
VLSI	Very-Large-Scale Integration
VM	Virtual Machine
WCET	Worst Case Execution Time

List of Symbols

$G(V, E)$	A DAG
V	Set of tasks in a DAG
E	Set of edges in a DAG
P	Set of processors in a platform
F_n	Set of frequencies of a processor p_n
$\|V\|$	Number of tasks
$\|E\|$	Number of edges
$\|P\|$	Number of processors
$e_{i,j}$	An edge $e_{i,j} = (\tau_i, \tau_j) \in E$; Dependency between tasks τ_i and τ_j
$data_{i,j}$	Data transmission time between task pairs τ_i and τ_j
D	Deadline of an application
τ_j	j^{th} task
τ_{entry}	Source task node of a DAG
τ_{exit}	Sink task node of a DAG
p_n	n^{th} processor

f_{n,α_n}	Maximum available frequency of p_n
$\omega_{j,n}$	Worst case execution time of task τ_j on processor p_n
$c_{i,j}^{m,n}$	Communication time between task pairs τ_i (executing on p_m) and τ_j (executing on p_n)
$pred(\tau_j)$	Predecessor nodes of τ_j
$succ(\tau_j)$	Successor nodes of τ_j
B	A matrix of size $\|P\| \times \|P\|$; Bandwidths between all pairs of processors
$b_{m,n}$	An element of B; Data transfer rate between processors p_m and p_n
ρe	$\rho e = \{\rho e_1, \rho e_2, \ldots, \rho e_{\|P\|}\}$; Penalization rates associated with task executions on different processors
ρc	$\rho c = \{\rho c_{1,1}, \ldots, \rho c_{1,\|P\|}, \ldots, \rho c_{\|P\|,1}, \ldots, \rho c_{\|P\|,\|P\|}\}$; Penalization rates associated with data transmissions over different communication links
ν	Size of the matrix in Gaussian Elimination
ϑ	Number of parallel branches in Epigenomics
υ	SeismogramSynthesis tasks of CyberShake
$\overline{\omega_{DAG}}$	Average execution time over all tasks in a DAG
$\overline{\omega_j}$	Average execution time over all processors for each task τ_j in a DAG
$\overline{c_{DAG}}$	Average communication workload
$\overline{data_{DAG}}$	Average inter-task message size
$\overline{B}$	Average communication bandwidth
$taskList$	Task priority order list
β	Heterogeneity factor
$initT$	Stores the current CPU clock time

$capT$	Allowable run-time bound
ops	Bound on the number of processor choices
p'	An array of size ops
$FSTime$	solution generation time of HMDS-Bl
msl	Best schedule length generated so far by HMDS
pow_n^s	Static power of processor p_n
pow_n^d	Dynamic power of processor p_n
pow_n^{ind}	Independent power of processor p_n
C_n^{ef}	Switching capacitance of processor p_n
m_n	Dynamic power exponent of processor p_n
f_y^{cr}	Critical operating frequency of processor p_n
$SU_{i,j}$	Security utility of $e_{i,j}$
T_{SU}	Total security utility
so_j^n	Total security overhead of task τ_j on processor p_n
$s_{i,j}^{x,Y_x}$	Maximum available security strength of service type x on an edge $e_{i,j}$
t	Current time
T	Upper bound on possible schedule length
Γ^a	Ambient room temperature
Γ^0	Initial temperature
Γ^q	Final temperature
$\Gamma_{j,n}^{ss}$	Steady state temperature of task τ_j on processor p_n
$\Gamma_{j,n}^{st}$	Starting temperature of task τ_j on processor p_n

$\mathcal{B}_n$	Thermal characteristics of processor p_n
Γ_n^{lim}	Upper bound on temperature associated with p_n
$\Gamma_{j,n}^{c}$	Cutoff temperature of τ_j on p_n
Γ_n^t	Temperature of a processor p_n at any time step t
$\Gamma_{j,n}^t$	p_n's temperature at time t when τ_j is mapped on it
$\xi_{j,n}$	Duration in which task τ_j remains allocated on p_n
S_j	Start time of task τ_j

Chapter 1

Introduction

1.1 Background

Cyber-physical systems (CPS) are transforming the way people interact with engineered systems. 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 [6]. Real-time Cyber-Physical Systems (*RT-CPSs*) must respond to external stimuli within stipulated upper bounds on time referred to as deadline [15]. Thus, the correctness of these systems depends not only on the value of the computation but also on the deadline by which the results are produced [83]. Applications in many computing systems ranging from avionic, automotive and industrial control to telecommunication systems, health care and even a significant class of consumer electronic systems, are real-time in nature. In general, an RT-CPS application is often represented by a Directed Acyclic Graph (DAG) (alternatively referred to as task graph in the rest of the report) whose nodes denote application tasks and edges denote inter-task dependencies [85]. To meet functionality specific high-performance demands, these DAGs are often implemented on heterogeneous platforms [51] where the same task may exhibit different timing/performance characteristics on the different processing elements. *Given DAG-structured applications and a heterogeneous processing platform, successful execution of application tasks while satisfying all resource, timing and precedence related specifications, is essentially a scheduling problem.*

Task scheduling problems are broadly classified as either *offline* (*static*) and *online* (*dynamic*) [5]. In offline scheduling, the decisions such as task-to-processor mappings, start times for task execution, etc., are taken at design time. Such offline scheduling decisions

are typically based on complete or partial design-time knowledge, such as the worst-case execution time of each task on every processor, precedence relationships and communication costs between task pairs, etc. In dynamic scheduling, such partial or full information about a task graph is not available before execution, and the scheduling decisions are exclusively taken at run-time. Dynamic scheduling schemes usually achieve higher average performance gains and are more flexible compared to static schemes. These relatively superior capabilities associated with dynamic strategies are achieved through the use of additional run-time information such as actual execution times of recently completed tasks, instantaneously available channel bandwidths, actual energy dissipation, etc. [72]. However, these superior capabilities are achieved at the cost of reduced predictability (in terms of reasonably accurate bounds on minimum performance guarantees) which static schemes are able to deliver. For safety-critical hard real-time cyber-physical systems, predictability, as well as performance and timeliness, is usually deemed to be more important than performance [15]. Hence, for safety-critical RT-CPSs such as automotive/avionic systems, static scheduling is often the desired choice [40].

In general, the problem of DAG scheduling falls in the NP-complete class [70, 87]. Computation of optimal schedules for DAGs on heterogeneous distributed computing systems requires exhaustive enumeration of an exponential state-space and is often prohibitively expensive, even for moderately large problem sizes. Therefore, research in this domain has often focused on designing low-complexity heuristics that produce quick but satisfactory schedules [5, 85]. Traditionally, list-based scheduling heuristics are known to generate efficient schedules for task graphs [26, 79]. A majority of list scheduling policies attempt to build a static schedule to minimize the overall *schedule length*, also known as *makespan*. Task graph application schedules with lower *makespans* are often beneficial in many ways, especially in RT-CPSs. As an example, given the temporal constraints (deadlines) as required for stable operation of an RT-CPS, the additional slack time acquired due to a lower *makespan* may be used to optimize one or more performance metrics such as expenditure on resources [2, 93], energy [62, 95], temperature [76], security [25], etc.

This thesis primarily aims at the development of a few low overhead list-based heuristic scheduling strategies for RT-CPSs consisting of both single DAG or multiple independent DAG applications on a distributed platform consisting of a set of fully connected heterogeneous processors. The '*fully connected processors model*' is widely being used by researchers and employed in many practical scenarios such as ZigBee-based wireless sensor networks,

WiFi networks, etc. In the scenario where the physical network is not fully connected, it can be modeled as a fully connected overlay network structure. An overlay is a logical network that uses virtualization to build connectivity on top of a physical infrastructure using tunneling encapsulations via Multi-Protocol Label Switching (MPLS), Generic Routing Encapsulation (GRE), Virtual Extensible LAN protocol (VXLAN), Virtual Routing and Forwarding (VRF), or other technologies [65]. The bandwidth of a link connecting a given pair of nodes in the overlay network is determined by the effective bandwidth of the path connecting the two nodes in the underlying physical network. This '*overlay-based network model*' allows us to circumvent the problem of communication contention existing in a shared network and helps in the design of a focused and efficient computation resource allocation scheme on a given set of distributed heterogeneous processing elements. This approach is practically useful and effective, especially in RT-CPSs which are usually dominated by computation workloads with short inter-task messages leading to comparatively low communication workloads.

The rest of the Chapter is organized as follows. The motivation and objectives of our work are specified in Section 1.2. We present a summary of the work done by providing a brief description of each algorithm developed by us in Section 1.3. The chapter ends by providing the organization of this thesis in Section 1.4.

1.2 Motivation and Objectives

This thesis deals with RT-CPS as its target domain. RT-CPS applications are usually dedicated towards controlling physical plants. The applications may be aperiodically triggered by an external event or may execute persistently in infinite loops, periodically acquiring data from the plant/environment through sensors, processing the same, and then generating appropriate actuation data. Today, there is an increasing trend toward the execution of RT-CPS applications over distributed network platforms, in diverse domains ranging from consumer electronics, robotics, automotive and avionics etc., to large manufacturing plants, smart grids and networked satellites. The control applications in these systems are often complex and executed on distributed high-capacity processing elements at locations which may potentially be geographically distant from the plant. On the other hand, the sensors and actuators are generally co-located with the plant. The complex control applications are usually modeled as DAGs with multiple possibly interdependent component

functionalities (tasks). Different component functionalities or tasks of an application may execute on distinct geographically distributed processing elements. In this scenario, data exchange between tasks executing on distinct processing elements as well as sensory/actuation data, must be transmitted as messages in a timely fashion over networks. The overall problem therefore involves two distinct real-time scheduling issues: that of *managing computation resources for application execution* and *managing communication resources for message transmission*.

Large systems such as those in today's distributed manufacturing units may constitute multiple control sub-systems. The associated control applications, each modeled as a DAG, often execute in a federated fashion on a dedicated (separate) sub-group of processing elements, in order to keep the design methodology simple while meeting specifications related to timing, energy, reliability, etc. However, it may be noted that such federated execution generally leads to poor utilization of resource capacities due to a lack of resource sharing among tasks and/or messages associated with the different DAG-structured applications. 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 processing platforms can reduce design costs, although it can cause significantly increased design complexity due to a higher degree of contention for shared resources.

In order to synthesize RT-CPSs composed of single or multiple DAG structured components on a consolidated platform of network-interconnected processing elements, industries have often adopted a hierarchical two-step design approach. Here, the first step 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 adoption of simpler design methodologies and hence have often been employed in practice. Thus, in the literature, we find a few studies dealing with parallel real-time task graphs [84,95,99] which focus towards precedence-aware scheduling of tasks, but completely ignore inter-task message transmission overheads. The schedule of tasks provides information about the messages to be delivered along with latency requirements associated with such delivery. There are separate streams of work which solely deal with the scheduling of messages on a given communication platform [9, 33]. 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. Although there is a significant amount of work on the integrated scheduling of DAG tasks and messages, a majority of them are oriented towards makespan minimization as their scheduling objective [3, 5, 12, 20, 24, 36, 38, 44, 85, 112]. On the other hand, there exist a few works which deal with the integrated real-time scheduling of DAG applications, they are primarily targeted towards homogeneous platforms [47, 48].

On the basis of the analysis presented here, we have envisaged the following broad scopes of work for this Ph.D. dissertation. Firstly, although there are a significant number of makespan minimization-based scheduling strategies for DAG structured applications on heterogeneous platforms, there seems to be ample scope for designing more systematic and efficient strategies by using the principles of any time heuristic search approaches. Secondly, our analysis also shows that by employing an efficient makespan minimization strategy as a core, effective task-message co-scheduling schemes for single/multiple real-time DAG applications can be designed. Next, we present an overview of important state-of-the-art works and discuss the objectives of this research work.

1.2.1 Makespan Minimization-based Scheduling Strategies

Researchers have made significant efforts to minimize the *makespan* of a task graph and developed many *list* scheduling static heuristics [5, 12, 20, 36–38, 85]. Topcuoglu et al. in [85] presented a list scheduling policy called *HEFT* which determines the priority of each task via *upward rank*. The *upward rank* of task τ_j is an estimate of the computation overhead of the path from τ_j to the sink task τ_{exit} of a task graph (including the execution overhead of task τ_j). The processor selection phase of *HEFT* picks a task having the highest priority and allocates it to a processor that minimizes its finish time. Canon et al. in [16] compared 20 list scheduling algorithms and concluded that *HEFT* outperforms the others for schedule length. Later, a few more list scheduling algorithms: *HPS* [37], *PETS* [36], *LDCP* [20] and *Lookahead* [12] are proposed, and these policies outperform *HEFT* with respect to schedule length, albeit at the cost of higher computational overhead. In [5], Arabnejad et al. proposed *PEFT* which outperforms *HEFT*, *HPS*, *PETS*, *LDCP* and *Lookahead* over a majority of test cases while incurring quadratic computational overhead in terms of the number of tasks. Authors in [24] and [112] proposed *PPTS* and *PSLS* as enhancements

over the *PEFT* algorithm.

Objective 1: *Although many makespan minimization-based scheduling strategies exist for DAG structured applications on distributed heterogeneous platforms, there is ample scope for designing more efficient strategies using the principles of any time heuristic search approaches, which allow the designer to obtain a judicious balance between performance (makespan) and solution generation times. Hence, the first important objective of this Ph.D. dissertation is to devise a new approach for obtaining makespan-minimizing DAG schedulers which can opportunistically use the principles of systematic heuristic search and deliver better performance than what the current state-of-the-art schedulers provide. A brief discussion of this work is presented in Section 1.3.2.*

1.2.2 Energy-aware Real-time Scheduling Strategies

The problem of minimizing energy dissipation of a real-time application with precedence-constrained tasks on homogeneous processors has been solved in several studies [47, 48]. For real-time DAG applications on heterogeneous processors, Huang et al. [35] proposed a strategy for reclaiming slack times of tasks on their pre-assigned processors, in order to reduce energy dissipation. Xie et al. [99] proposed two schedulers: *NDES* and *GDES*. Given a heterogeneous platform with different task-to-processor energy consumption affinities, *NDES* solved the problem of energy-aware processor mapping for each task within a given deadline. *GDES* solved the same problem for a DVFS-enabled system. Similarly, the work in [95] has also developed DVFS-enabled scheduling heuristics to perform energy minimization. However, these works are targeted towards parallel DAG-structured applications where inter-task communication times are ignored.

Objective 2: *As discussed earlier in this section, there is a severe dearth of research works on task-message co-scheduling of real-time precedence-constrained task graphs. As a second objective, we have envisaged the design of a generic real-time DAG scheduling framework which can be amicably adapted towards its deployment in various application domains such as automotive and avionic systems, cloud computing, smart grids, industrial automation, etc. Section 1.3.1 presents a brief discussion of this work.*

1.2.3 Scheduling Multiple Independent DAG Applications

A detailed survey on multiple independent task graph scheduling schemes for distributed systems may be found in [31]. Authors in [111] proposed two policies to address the problem of scheduling multiple independent DAGs on heterogeneous systems. While the first policy attempts to minimize the overall makespan, the second policy aims at reducing unfairness in the slowdowns experienced by each individual DAG within the given set of DAG. Hsu et al. in [32] extended the first policy and proposed an algorithm called Online Workflow Management (OWM), targeted towards dynamically arriving task graph applications. Similarly, Arabnejad et al. in [4] extended the second policy in [111] and presented an algorithm called Fairness Dynamic Workflow Scheduling ($FDWS$) for the online scheduling of dynamic task graphs. However, none of the above works are applicable to task graphs having real-time constraints. Hu et al. in [33] presented an approach for scheduling a set of periodic real-time DAG applications on safety-critical time-triggered systems. In this system, they assumed that task-to-processor allocations were known beforehand. The objective is to generate a schedule which satisfies the deadlines of all instances of every application. However, this work focuses only on meeting the deadlines of all application instances and does not endeavor to optimize additional performance metrics related to energy, cost, temperature, reliability, security, etc.

Objective 3: *As discussed, large systems which constitute multiple control subsystems typically follow a federated resource allocation policy as it allows simpler design, albeit at the cost of significantly lower resource utilizations in many cases. In order to improve the usage efficiency of available processing and network resources, our real-time DAG scheduling framework can be extended to enable the co-scheduling of multiple independent periodic real-time applications. This forms the third objective of the thesis. A brief discussion of the same is presented in Section 1.3.3.*

1.2.4 Security-aware Real-time Scheduling

In the last two decades, researchers have made significant efforts towards developing security-aware scheduling strategies using for both *dynamic* [101, 104, 105] as well as *static* [94, 102, 103] approaches. Authors in [101, 104, 105] proposed a family of *dynamic security-aware scheduling strategies* for platforms ranging from uni-processor systems [105] and multi-core homogeneous systems [101, 104], to heterogeneous distributed systems [103, 104]. While

some of these studies focus on independent tasks set [101, 103–105], the others have dealt with applications modeled as precedence-constrained task graphs [104]. On the other hand, Xie and Qin in [102] proposed a *static scheduling* strategy *SASES* for independent periodic tasks on uni-processor systems, which accounts for both timing and security requirements. Given the risk probabilities associated with the execution of tasks on a set of available processors, Tang et al. in [94] proposed a security-driven *static list scheduling strategy* for performance-sensitive non real-time DAG-structured applications, whose objective is to minimize schedule length.

Objective 4: *The above discussion shows that there does not currently exist any significant security-aware static real-time DAG scheduling algorithms for heterogeneous platforms. Thus, the generic real-time DAG scheduling framework (discussed in the second objective) can be extended with customized design strategies to maximize system security as the scheduling goal. This forms the fourth objective of the thesis. Section 1.3.4 provides a brief explanation of the security-aware real-time scheduler.*

1.2.5 Temperature-aware Scheduling Strategies

A survey of a few important temperature-aware scheduling techniques for multi-core processing systems may be found in [78]. Authors in [10] addressed a thermal-aware global multiprocessor scheduling algorithm to increase the number of finished tasks. They also examine the competitive ratios of a large class of scheduling techniques as a function of the processors' cooling factors. In [113], Zhou et al. presented a heuristic strategy for scheduling independent tasks with reliability and temperature constraints on heterogeneous platforms to minimize schedule length. A few recent scheduling techniques on this topic which target independent tasks on homogeneous and heterogeneous systems may be found in [57, 64] and [52, 75], respectively. Sheikh et al. in [76] proposed a temperature-aware scheduling algorithm to minimize the makespan of DAG-structured applications. In [77], the same authors also proposed a DAG scheduling technique for simultaneously optimizing makespan, energy and temperature. However, these works are focused on homogeneous multi-core systems.

Objective 5: *In spite of the above efforts, there does not exist any temperature-aware makespan minimizing DAG scheduler over distributed heterogeneous processing platforms. Hence, the last objective of this dissertation is to develop a temperature-aware makespan minimizing DAG scheduling policy. A brief elaboration of this work is presented in Sec-*

tion 1.3.5.

1.3 Summary of Contributions

As part of the Ph.D. work, we have developed different list-based scheduling schemes for DAG-structured applications on heterogeneous distributed RT-CPSs. The entire thesis work comprises multiple contributions categorized into five contributory chapters, each of which is targeted to achieve different scheduling objectives. The contributions of this thesis are summarized below.

1.3.1 PRESTO: A Penalty-aware Real-Time DAG Scheduler for Heterogeneous Distributed Systems

PRESTO is an efficient real-time scheduling framework for DAG-structured applications. The objective of *PRESTO* is to minimize a *generic penalty function* that can be amicably adopted towards its deployment in various application domains. The basic intuition of PRESTO is explained below.

At a given iteration, tasks in τ_S (tasks priority list) are allocated sequentially in order. The task (say, τ_j) to be scheduled next, maybe potentially allocated to any processor p_n on the heterogeneous platform. However, selection of the actual processor is made by judiciously considering the following two parameters, for each such possible allocation: (i) $MEP[\tau_j, p_n]$: *Minimum Estimated Penalty* of τ_j on a potential processor p_n, and (ii) $ESL[\tau_j, p_n]$: *Estimated Schedule Laxity* of τ_j on p_n. $ESL[\tau_j, p_n]$ is a throttled estimate of the spare time that should remain before deadline D and after completion of the sink node τ_{exit}, for the case when all tasks in τ_S before τ_j, have already been scheduled and τ_j is restricted to execute on p_n. Among available allocation options, a task τ_j is first considered for allotment within that subset of processors say, $P_S = \{p_{s1}, p_{s2}, \ldots, p_{sr}\}$, for which $ESL[\tau_j, p_n]$ is non-negative. It may be noted that, as the number of iterations grow and the value of ESL gets increasingly throttled, the number of processors which remain eligible towards inclusion in P_S, reduces progressively. Now, τ_j is actually allocated to that processor p_{si} in P_S for which $MEP[\tau_j, p_{si}]$ is minimal. Hence, higher the iteration number larger is the penalty associated with the schedule generated in that iteration. However, if $ESL[\tau_j, p_n]$ is negative for all available processors (i.e., $P_S = NULL$), τ_j is allocated on that processor for which $ESL[\tau_j, p_n]$ is minimally negative.

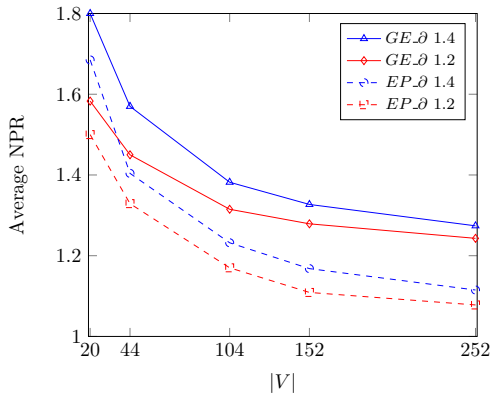


Figure 1.1: *NPRs w.r.t varying #tasks for Gaussian Elimination and Epigenomics.*

Experimental analysis using two benchmark task graphs, Gaussian Elimination (GE) and Epigenomics (EP) reveals that *PRESTO* performs appreciably over extensive sets of test scenarios, pointing to the practical effectiveness of the scheme. It can be observed from Fig. 1.1 that for any given number of tasks, the average Normalized Penalty Ratio (NPR) increases as the deadline extension rate (∂) increases, indicating that *PRESTO* is able to efficiently harness higher available slack to achieve better penalty reductions. As is obvious, NPR values decrease as the workload becomes higher with an increase in the number of tasks.

1.3.2 HMDS: A Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems

This work presents a low-overhead depth-first branch and bound-based search approach, called *HMDS*. *HMDS* has been equipped with a set of novel tunable pruning mechanisms, which allow the designer to obtain a judicious balance between performance (*makespan*) and solution generation times, depending on the specific scenario at hand.

Given a fixed task priority order *taskList*, *HMDS* generates a static schedule by determining: (i) a *processor allocation*, (ii) an *actual start time*, and (iii) an *actual finish time*, such that the *makespan* is minimized. The task (say, τ_j) in *taskList* to be scheduled next may be potentially allocated to any processor p_n on the heterogeneous platform. Each such recursive task-processor allocation procedure (say, $\langle \tau_j, p_n \rangle$) is associated with three attributes: (i) *Effective Start Time* (EST), (ii) *Effective Finish Time* (EFT), and (iii) *Optimistic Effective Finish Time* (O_{EFT}). $O_{EFT}[\tau_j, p_n]$ provides an estimate of the sink task's completion time relative to the effective execution finish time of the current task τ_j on p_n.

The recursive procedure first tries to extend the schedule with τ_j allocated to p'_1, the processor on which τ_j's O_{EFT} is minimum. Subsequently, when the schedule backtracks, it extends with τ_j on p'_2, the processor for which τ_j's O_{EFT} is second best, and this process continues likewise. After a complete schedule is generated, the currently known best schedule and its associated *makespan* are returned.

Table 1.1: *Pair-wise makespan comparison of the scheduling algorithms*

	HEFT			*PEFT*			*PPTS*			*PSLS*		
	better	equal	worse	better	equal	worse	better	equal	worse	better	equal	worse
HMDS	85.2%	5.5%	9.3%	82.9%	14.3%	2.8%	86.0%	8.0%	6.0%	80.8%	12.0%	7.3%

Experimental analyses using two real-world benchmarks show that *HMDS* is able to comprehensively outperform state-of-the-art algorithms such as *HEFT*, *PEFT*, etc., in terms of archived *makespans*. It can be observed from Table 1.1 that *HMDS* performs better, equal and worse in 85.2%, 5.5% and 9.3% test cases respectively, compared to *HEFT*.

1.3.3 DPMRS: Energy-aware Real-time Scheduling of Multiple Periodic DAGs on Heterogeneous Distributed Systems

The third contribution of our work presents an energy-aware static scheduler called *DPMRS* for a set of real-time control applications co-executing in a heterogeneous distributed environment. The objective of *DPMRS* is to minimize dissipated energy.

DPMRS consists of *three* functions: (i) *DPMRS()*, (ii) *periodicMerge()*, and (iii) *ERRRank()*. The main function *DPMRS()*, which conducts the overall scheduling, and calls function *periodicMerge()* to merge all independent application DAGs into a single DAG at the first step. Given the merged DAG, *DPMRS()* next calls *ERRRank()* to compute two different parameters namely, (1) *Expected Relative Residual-workload* for each task-processor pair and (2) A rank value for each task which is used to generate a task priority order. *DPMRS* extends our real-time DAG scheduling framework *PRESTO* and is equipped with the necessary critical features that are required to handle multi-application merged DAGs. These features include:

(a) An enhanced task prioritization mechanism, where *ranks* of the source nodes of all application instances (within the merged DAG) are designed to be aware of the relative

arrival times (within the hyperperiod) of the respective application instances. Similarly, *ranks* of the sink nodes of all application instances must be made aware of application instance deadlines. Finally, the *ranks* of all other intermediate nodes within each application instance (within the merged DAG) must be sensitive to the instance's relative deadline.

(b) In addition to such an enhanced task prioritization mechanism which is necessary for improved task ranking, the allocation phase during partial schedule generation must be able to recognize and take corrective actions in situations when: (i) the scheduler attempts to assign to a source task node a start time which is earlier than stipulated relative arrival time. (ii) the scheduler attempts to assign to a sink task node a start time which causes the application instance's relative deadline to be missed.

Table 1.2: *Normalized Energy-dissipation (NE; in W) w.r.t. varying processors.*

Processors	8		16		32	
Algorithms	*DPMRS*	*NDES&GDES*	*DPMRS*	*NDES&GDES*	*DPMRS*	*NDES&GDES*
Average NE	13.66	159.17	3.69	70.45	1.51	23.97

Compared to state-of-the-art energy-aware single DAG scheduler *NDES* and *GDES*, the global nature of *DPMRS* allows it to harness significantly improved processor/communication resource sharing among different benchmark DAGs, in addition to better exploitation of task-processor affinities in the heterogeneous environment. Due to such efficient sharing and affinity awareness, *DPMRS* is able to achieve considerably lower energy dissipation as compared to *NDES* and *GDES*. For example, in the scenario consisting of 16 processors in Table 1.2, it may be observed that the *Normalized Energy-dissipation* suffered by *DPMRS* is 3.69 W. In comparison, *NDES* and *GDES* suffer significantly higher dissipation − 70.45 W.

1.3.4 SHIELD: Security-aware Scheduling for Real-time DAGs on Heterogeneous Systems

SHIELD is a real-time static scheduler whose objective is to maximize total security utility for a given task graph application having known minimum security strength specifications for its messages. Salient facets incorporated within the design of *SHIELD* include:

(a) *An enhanced task prioritization mechanism* which not only is task-precedence aware but also possesses the following important properties: *rank* of each task τ_j is an estimate of

the overall relative remaining workload (combining task execution, message transmission as well as security overhead) associated with the sub-graph rooted at τ_j (up to the sink task node of the DAG).

(b) *A processor allocation mechanism* in which, for each task (chosen in reverse chronological order of the rank values) an appropriate processor is selected from the pool of available heterogeneous processors and a suitable start time is assigned for the task on this chosen processor. The overall aim of this mechanism is to generate a *makespan* minimizing schedule in which a chosen set of security service protocols are applied on the messages in accordance with their minimum security performance demands.

(c) *SHIELD* returns with *failure*, if the minimum *makespan* schedule generated by the previous step overshoots the given deadline. Otherwise, *SHIELD* runs a "*security enhancement procedure*" over the schedule generated in the previous step. Specifically, keeping the task-to-processor assignment and task scheduling order undisturbed, *SHIELD* attempts to utilize the inter-task as well as overall slacks available before the deadline, to maximize aggregate security performance of the system.

Experimental analysis using two benchmark task graphs: *Gaussian Elimination* and *Cybershake*, reveal that *SHIELD* significantly outperforms two greedy baseline strategies *SHIELDb* in terms of solution generation times (run-times) and *SHIELDf* in terms of achieved security utility, over various input test scenarios.

1.3.5 TMDS: A Temperature-aware Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems

The final contribution of this dissertation presents a low-overhead temperature-aware algorithm called *TMDS* for DAG-structured applications. The objective of *TMDS* is to generate a *makespan* minimizing schedule while satisfying, (i) execution and communication demands of application tasks, (ii) processor capacity and communication bandwidth constraints of the platform, and (iii) temperature threshold bound associated with all processors, over the entire schedule.

The *TMDS* comprises three phases, namely *task prioritization*, *processor allocation* and *temperature-aware scheduler*. (a) The fast phase decides a fixed task priority order among the ready tasks that ensure the satisfaction of all precedence constraints associated with the task graph. (b) The second phase determines a suitable processor for the highest priority

unallocated task, at a given intermediate point during partial schedule generation. In order to obtain this processor, a *temperature-aware evaluation* on the allocation-goodness of the selected task on all available processors, is conducted. (c) This *temperature-aware* schedule generation strategy adheres to two design principles for minimizing the total execution length of τ_j on p_n in the presence of an *active-to-idle* transition overhead. (i) An interval in which task τ_j actually executes on processor p_n is never interrupted/terminated until the threshold temperature of p_n is reached, (ii) Whenever the temperature of p_n reaches its threshold value, it is switched to idle mode. Thereafter, p_n continuously cools off in this idle state for a minimum duration which either allows: (a) continuous execution of τ_j up to completion (while not violating p_n's threshold) in the execution interval which follows this cool-off interval, or (b) cools down up to the designed cut-off temperature.

Table 1.3: *Pair-wise comparison of algorithms using Gaussian Elimination*

	TPSLS			*TPPTS*			*TPEFT*			*THEFT*		
	better	equal	worse	better	equal	worse	better	equal	worse	better	equal	worse
TMDS	66.2%	6.7%	27.1%	61.1%	1.4%	37.5%	53.9%	14.4%	31.7%	65.5%	1.4%	33.1%

The proposed temperature management strategy is a generic approach that can be easily used to adapt existing *makespan* minimizing DAG schedulers (for example, *HEFT*, *PEFT*, etc.), so that the delivered schedules never violate threshold temperature bounds of processors. This generic approach is important because as shown in Table 1.3, although *TMDS* is better or comparable in performance to the other state-of-the-art algorithms in a majority of considered test cases, there still are a significant number of test case scenarios in which one or more existing algorithms deliver slightly better results than *TMDS*.

1.4 Organization of the Thesis

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

- **Chapter 2**: This chapter introduces the essential terminology and background of real-time systems, which are needed to understand the following chapters and discuss important state-of-the-art scheduling strategies that are related to this dissertation.
- **Chapter 3**: This chapter presents *PRESTO*, a real-time static scheduler for DAG-structured applications which attempts to optimize a generic penalty function of the processor and bus allocations corresponding to a particular schedule. Theoretical analysis and experimental results of *PRESTO* have been presented and discussed.
- **Chapter 4**: In this chapter, we discuss a low-overhead depth-first branch and bound based search approach called *HMDS* for task graphs, whose objective is to minimize the overall *makespan*. *HMDS* is adaptable and can be tuned to provide approximate solutions having varied solution qualities with associated differences in solution generation times, by controlling the nature and extent of solution space exploration.
- **Chapter 5**: This chapter deals with a real-time static list scheduler for *multiple independent periodic DAG applications*. The objective of the scheduler is to minimize energy dissipated by the system during execution of a given set of DAGs.
- **Chapter 6**: Here, we present a real-time policy *SHIELD* for heterogeneous platforms, whose objective is to maximize total security utility for a given DAG-structured application having known minimum security strength specifications for its messages.
- **Chapter 7**: This chapter deals with the design of a low-overhead temperature-aware heuristic scheduler called *TMDS*, for DAG-structured applications. The objective of *TMDS* is to minimize the overall schedule *makespan* of task graphs by deciding task-processor allocations on a heterogeneous distributed platform while ensuring that processor temperatures never overshoot their respective threshold *cutoff* values.
- **Chapter 8**: The thesis concludes with this chapter. We discuss the possible extensions and future works that can be done in this area.

Chapter 2

Background on Real-time Systems and Literature Survey

2.1 Introduction

Real-time systems are characterized by their ability to respond to events that may happen in their operating environment within stipulated time bounds. Thus, the correctness of these systems depends not only on the value of the computation but also on the deadline by which the results are produced [15,83]. Real-time systems span a wide range of domains, including industrial control systems, multimedia systems, automotive and aviation systems, consumer electronics, etc. A typical example of a real-time system is provided by a temperature controller in a chemical plant that is required to switch off the heater within 30 milliseconds when the temperature reaches 250°, to avoid an explosion. Ensuring temporal correctness is ultimately a scheduling problem. The chapter starts with a brief overview on the structure of real-time systems and then discusses important state-of-the-art works related to this dissertation.

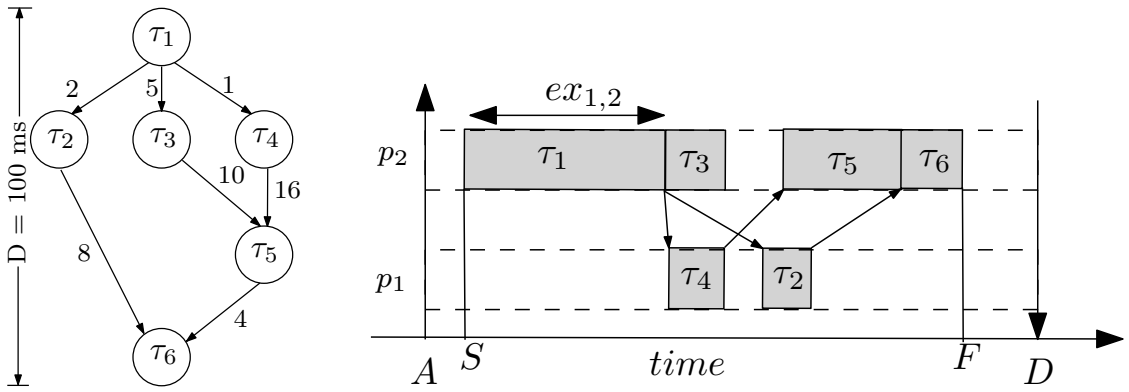
2.2 Real-time Systems

Typically, real-time systems are composed of the *Application Layer*, *Real-time Scheduler* and *Hardware Platform* [61]. *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, which will in turn allow the design of efficient scheduling strategies.

2.2.1 Application Layer

The application layer comprises all the applications that the system needs to execute. In general, a real-time application is represented as a Directed Acyclic Graph (DAG). In a DAG, each node represents a piece of code (or program) called task, while edges denote inter-task dependencies [85]. Fig. 2.1a depicts an example DAG, $G(V, E)$, where the set of vertices $V = \{\tau_1, \tau_2, \ldots, \tau_{|V|}\}$ represents tasks and the set of edges E represents precedence constraints between task pairs. For example, an edge $e_{i,j} = (\tau_i, \tau_j) \in E$ denotes the dependency between tasks τ_i and τ_j; that is, τ_j can only start after τ_i completes execution and its output arrives τ_j. Edge $e_{i,j}$ is assigned with a positive weight $data_{i,j}$ (e.g., $data_{4,5} = 16$) which represents the amount of data to be transferred from task τ_i to task τ_j. The DAG has an end-to-end deadline of 100 milliseconds. We assume that the task graph has a single source/entry task τ_{entry} and a single sink/exit task τ_{exit}. Scenarios in which a task graph has multiple sources (sinks) are handled by adding a dummy source (sink) task. We now discuss the set of definitions related to a real-time application.



(a) *A sample DAG* **(b)** *Typical parameters of a real-time DAG application*

Figure 2.1: *(a) A DAG $G(V, E)$, $|V|$=6, $|E|$=7; (b) Parameters of a real-time DAG application.*

2.2.1.1 Real-time Application Model

Formally, a real-time DAG-structured application G (refer Figure 2.1) can be characterized by the following parameters:

- **Arrival time** (A) is the time at which an application becomes ready for execution. It is also referred as *release time* or *request time* of the application.
- **Start time** (S) is the time at which the source task τ_{entry} (i.e., τ_1 of G; refer Figure 2.1a) of the application G starts its execution.
- **Execution time** or *Computation time* ($ex_{j,n}$) is the time taken by the processor p_n to finish the computation of τ_j without interruption.
- **Finishing time** or *Completion time* (F) refers to the time at which the sink task τ_{exit} (i.e., τ_6 of G; refer Figure 2.1a) finishes its execution.
- **Deadline** (D) is the time before which the application's sink task τ_{exit} should complete its execution without causing any damage to the system. If a deadline is specified with respect to the application's source task arrival time, is called *relative deadline*, whereas if it is specified with respect to time zero, it is called an *absolute deadline*.
- **Worst-case execution time** ($\omega_{j,n}$) is the largest computation time of a task τ_j among all of its possible executions on processor p_n (i.e., $ex_{j,n} \leqslant \omega_{j,n}$).
- **Slack time** or *Laxity* (*slack*) is the maximum amount of time by which execution of an application can be delayed after its activation/arrival, to complete within its deadline ($slack = D - ex_{exit,n}$; $ex_{exit,n}$ is the execution time of sink task τ_{exit} of G on processor p_n).
- **Priority** is the importance given to the application's tasks in context of a schedule at hand.
- **Criticality** is a parameter related to the consequences of missing a deadline (typically, it can be *hard*, *firm*, or *soft*).

A real-time application G can be classified as periodic, aperiodic or sporadic based on regularity of its activations [15, 22].

- **A Periodic** application arrives strictly periodically, separated by a fixed time interval (say, π). For example, the track correction application of a rocket recurs periodically every 50 milliseconds.

- **An Aperiodic** application arrives randomly and is not known a priori. Examples are provided by interactive applications such as in railway reservation systems, it is difficult to predict the number of clients that will be accessing the system at a particular instant.

- **A Sporadic** application that may arrive at any time once a minimum interarrival time has elapsed since the arrival of the previous instance of the same application. For example, tasks encountered by a fire prevention system may be considered sporadic because there is usually a minimum duration between the occurrence of two consecutive fires.

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

- **Implicit Deadlines:** All application deadlines are equal to their periods ($D = \pi$).

- **Constrained Deadlines:** All application deadlines are less than or equal to their periods ($D \leqslant \pi$).

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

- **Hyperperiod ($\mathcal{H}$)**: Given a static application system, $\mathcal{H}$ represents the minimum time interval after which the schedule repeats itself. For a set of periodic applications, $G = \{G_1, G_2, \ldots, G_{|G|}\}$ with periods $\{\pi_1, \pi_2, \ldots, \pi_{|G|}\}$, hyperperiod is given by the LCM of the periods ($\mathcal{H} = LCM(\pi_1, \pi_2, \ldots, \pi_{|G|})$).

This dissertation deals with both periodic as well as aperiodic applications .

2.2.2 Real-time Scheduler

A real-time scheduler acts as an interface between applications and the hardware platform. The scheduler generates a schedule by sequentially determining: a *processor allocation* and

an *execution start time* for each task of the application, in the order prescribed by a predetermined task priority order such that the scheduling objective is optimized. A set of rules that, at any time, determines the order in which tasks are executed is called a *task priority order*. In our system, a generated schedule is referred to as *feasible/valid*, if the sink task τ_{exit} finishes its execution on or before the stipulated deadline D. An application 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.

2.2.3 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.

- **Uniprocessor** systems can execute only one task at a time and must switch between tasks.
- **A Multiprocessor** system may range from several separate uniprocessors tightly coupled using high-speed networks to multi-core. It can be classified as follows:
  - **Homogeneous:** The processors are identical; hence the rate of execution of all tasks is the same on all processors.
  - **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.
  - **Heterogeneous:** The processors are different; hence the rate of execution of a task depends on both the processor and the task. That is, the same task may exhibit different execution time requirements on different processors.

In this dissertation, we have considered heterogeneous multiprocessor systems. Memory in multiprocessor systems can either be distributed or shared. In distributed memory, 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. In contrast, a shared memory multiprocessor offers a single memory space used by all processors. *This thesis deals with distributed memory multiprocessor systems.* In distributed systems, processors communicate with each other through message passing using the underlying communication network. Processors in the network can either be fully connected or connected through a shared bus-based network.

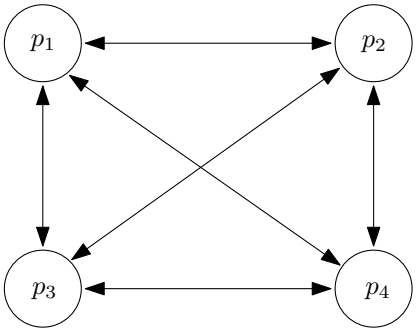


Figure 2.2: *Fully connected multiprocessor system.*

1. **Fully-connected multiprocessor system:** In a fully connected system, each pair of processors have a dedicated communication channel to send/receive data or messages. Figure 2.2 shows a typical fully connected multiprocessor system where all processors are connected to each other through dedicated bidirectional communication links. For example, processor p_1 has separate communication links to processors p_2, p_3 and p_4.

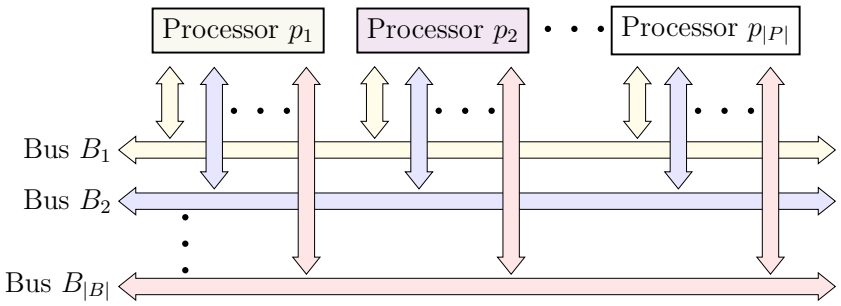


Figure 2.3: *Shared bus-based multiprocessor system.*

2. **Shared bus-based multiprocessor system:** In this system, processors are connected through a 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, processor p_1 is connected to all buses $B_1, B_2, \ldots, B_{|B|}$.

In this dissertation, we have considered heterogeneous distributed multiprocessor systems where processors are fully connected through heterogeneous communication links. The fully connected processors model is widely being used by researchers and employed in many practical situations using ZigBee-based wireless sensor networks, WiFi networks, Z-Wave, etc. WiFi and Zigbee are currently the most popular protocols for smart homes, smart grids, healthcare, robotic industrial systems, etc. Even if the physical network is not fully connected, it can be modeled as a logically fully connected overlay network structure. An overlay is a virtual network built on top of an underlay physical network infrastructure. It creates a tunnel that is designed to provide the services necessary to implement any standard point-to-point encapsulation scheme via Multi-Protocol Label Switching (MPLS), Generic Routing Encapsulation (GRE), Virtual Extensible LAN protocol (VXLAN), Virtual Routing and Forwarding (VRF), or other tunneling protocols. A tunnel looks like a single-hop and tunneling protocols prefer a tunnel over a multi-hop physical path. For example, in the topology depicted in Figure 2.4, messages from processor p_1 will appear to travel across network segments N1, T and N5 to get to processor p_2 instead of taking the path N1, N2, N3, N4 and N5. In fact, the messages going through the tunnel will still be traveling across switches SW1, SW2, SW3 and SW4. In the figure, processor p_1 (p_2) is connected to switch SW1 (SW2) by a low-bandwidth communication link and link delays are given as edge weights.

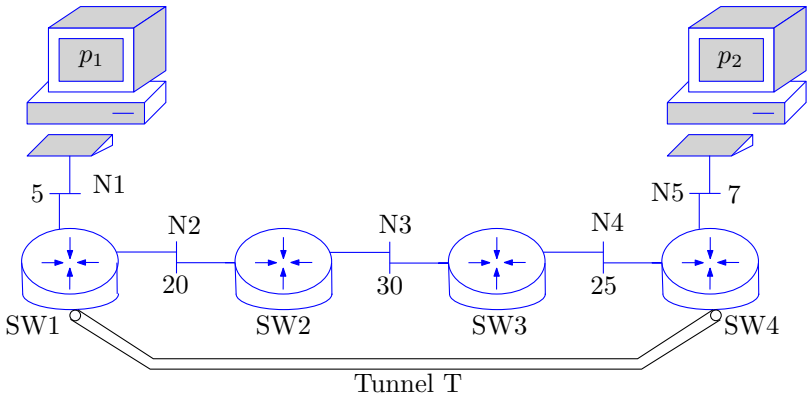


Figure 2.4: *Tunneling in an overlay network.*

The bandwidth of a tunnel/link connecting a given pair of processors in the overlay network is determined by the effective bandwidth of the path connecting the two processors in the underlying physical network. This overlay-based network model allows us to circumvent the problem of communication contention existing in a shared network and allows us to focus our attention on the design of efficient computation resource allocation schemes for given heterogeneous execution platform scenarios. This approach is practically useful and effective, especially in distributed cyber-physical systems where messages typically tend to be short and are heavily dominated by computation workloads.

2.3 Types of Real-time Application Constraints

Designing efficient real-time DAG-structured application scheduling strategies for RT-CPSs must encounter several challenges. We now enumerate a few such critical challenges and discuss them [15].

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 into three categories:
   - **Hard:** A real-time application is said to be hard if missing its deadline may cause catastrophic consequences on the system under control. Examples: autopilot systems, planetary rovers, anti-missile systems, etc.
   - **Firm:** A real-time application is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value. Examples: satellite-based tracking applications, financial forecast systems, video conferencing applications, etc.
   - **Soft:** A real-time application is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation. Examples: railway ticket reservation, online transaction processing systems, weather monitoring systems, etc.
2. **Precedence Constraints:** In DAG-structured applications, computational activities cannot be executed in an arbitrary order but have to respect some precedence relations

defined at the design stage. The set of all direct predecessor tasks (successor tasks) for a task τ_j is represented as $pred(\tau_j)$ ($succ(\tau_j)$). Figure 2.1a illustrates a DAG that describes the precedence constraints among six tasks. From the figure, it can be observed that task τ_1 do not have any predecessor and can immediately start its execution. Tasks with no predecessors are called source task nodes. It can also be seen that τ_2, τ_3 and τ_4 can start executing only after the predecessor task τ_1 completes its execution. Tasks with no successors, as τ_6 in the figure, is called sink task.

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.

4. **Energy Minimization**: Energy dissipation in real-time systems has become an important issue with the increase in the number of processing elements. Effective energy management is essential for battery-powered embedded systems, such as those deployed in hand-held devices, space missions, industrial controllers, pacemakers in health care, etc. Replacing or recharging batteries in such systems is not always feasible or practical. 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 financial and environmental gains [7]. *Scheduling schemes for real-time systems must minimize energy consumption while satisfying other constraints like timing, resource, precedence, etc.*

5. **Security-awareness**: As discussed earlier, DAG-structured applications are often executed on distributed heterogeneous platforms. Further, messages which deliver data between tasks are often transmitted over public networks, and are hence susceptible to multiple security threats such as *snooping*, *alteration* and *spoofing*. Several alternative security protocols having varying security strengths and associated implementation overheads (particularly at source and destination tasks of messages) are available in the market, for incorporating *confidentiality*, *integrity* and *authentication* on the transmitted messages. *Hence, given a resource-constrained computation*

platform, a security-aware RT-CPS scheduler should be able to judiciously choose appropriate schemes for the three types of security services, so that overall security of the system is maximized while adhering to stipulated timeliness constraints.

6. **Thermal Constraints**: Real-time systems implemented on heterogeneous platforms need to satisfy *Thermal Design Power (TDP)* thresholds employed by processor manufacturers [52]. The elevation in temperature beyond TDP may trigger Dynamic Thermal Management (DTM) to ensure the thermal stability of the system. However, the application of DTM makes the system susceptible to higher unpredictability and performance degradations for real-time applications [45, 58]. *This necessitates the development of schedulers that can guarantee adherence to a system-level peak thermal constraint.*

Given a hard real-time application modeled as a DAG and a heterogeneous processing platform, successful execution of application tasks while satisfying all timing, precedence and resource-related specifications, is ultimately a scheduling problem.

We now provide a brief description of a few common classifications of the different scheduling policies relevant to real-time systems.

2.4 A Classification of Real-Time Scheduling Policies

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

- **Preemptive Vs. Non-preemptive Scheduling:** In preemptive strategies, a running task of an application 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 reallocated to the same processor or to a different processor [27]. On the contrary, in non-preemptive strategies, a task of an application, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.

- **Offline Vs. Online Scheduling:** In *offline* scheduling, the scheduler has a priori knowledge of the application's tasks 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 offline in nature. On the other hand, *online* scheduling algorithms make their scheduling decisions at runtime based on information about the application that has arrived so far. Although they are often flexible and adaptive, they may incur significant overheads because of runtime processing. However, online schedulers are essential in systems that do not have enough information before run-time, to design the schedule statically. Online scheduling is also referred to as dynamic or runtime scheduling.

- **Optimal vs. Heuristic:** An algorithm is said to be optimal if it minimizes or maximizes some given cost function defined over all the tasks of an application. 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. On the other hand, an algorithm is said to be heuristic if it is guided by a heuristic function in making its scheduling decisions. A heuristic algorithm tends toward the optimal schedule but does not guarantee finding it. In general, scheduling of an application modeled as DAG has been shown to be NP-complete [18, 28, 87]. Computation of optimal schedules for DAGs on heterogeneous platforms requires exhaustive enumeration of an exponential state-space and are often prohibitively expensive even for moderately large problem sizes. Therefore, research in this domain often focuses towards the design of low-complexity heuristics that produce quick and satisfactory schedules which are generally sub-optimal in nature [38, 80].

This thesis deals with non-preemptive task execution of hard real-time DAG-structured applications which are scheduled using heuristic-based offline/online strategies on heterogeneous distributed systems. We have added a taxonomy in Figure 2.5 and the highlighted boxes represent scope of this thesis.

Next, we present a few important state-of-the-art works related to this dissertation.

2.5 Survey of Scheduling Algorithms

DAG scheduling can be conducted *statically*, based on information available about the application and platform at design time (before putting the system into operation), or dynamically, at run-time. In the last two-three decades, researchers have made significant

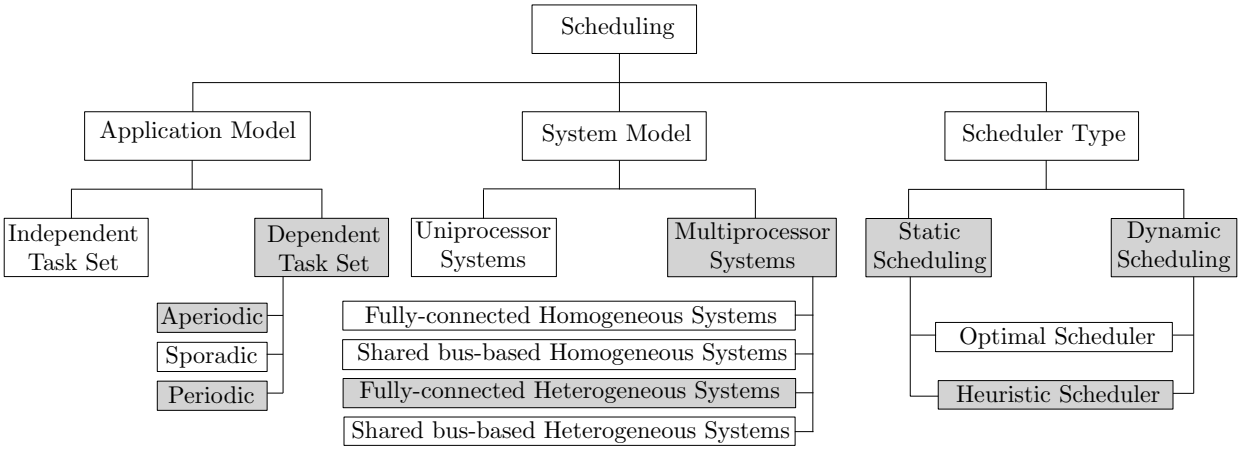


Figure 2.5: *Taxonomy; highlighted boxes represent the scope of this thesis.*

efforts towards developing effective RT-CPSs scheduling strategies using both *static* [5, 20, 85, 93–95, 102, 103] as well as *dynamic* [30, 57, 101, 104, 105] approaches. We now present a category-wise discussion of state-of-the-art scheduling strategies relating to this dissertation.

2.5.1 Makespan Minimization Scheduling Strategies

Researchers have made significant efforts to minimize the *makespan* of a task graph and developed many *list* scheduling heuristics [5, 12, 20, 36, 38, 85]. In [85], Topcuoglu et al. proposed two list scheduling strategies called *HEFT* and *CPOP*. These two algorithms differ in both their task prioritization and processor selection phases. The *HEFT* algorithm computes the priority of every task using a parameter called *upward rank*. The *upward rank* of a task τ_j is an estimate of the computation cost of the path from τ_j to the sink task (exit node of a DAG) including the computation time of τ_j. In the processor selection phase, *HEFT* selects the highest priority task and assigns it to the processor which minimizes the task's finish time. On the other hand, *CPOP* prioritizes tasks based on the summation of *upward* and *downward rank* values and schedules all critical tasks (that is, tasks on the critical path of the DAG) onto a single processor, in an attempt to minimize the total execution time of the critical tasks. The non-critical tasks are assigned to the other processors such that the *makespan* is minimized. The *HEFT* algorithm is faster and produces lower schedule lengths than *CPOP*. The authors in [16] compared 20 heuristic schedulers using *randomly* generated DAGs and *montage* task graphs as inputs and concluded that *HEFT* outperforms the others for both *makespan* and *robustness*. In [92], the authors have introduced the notion of *successor-tree consistent deadline* to compute task priorities. Compared to the other

prioritization schemes, the *successor-tree-consistent deadline* approach attempts to consider both precedence constraints and resource constraints, in a more accurate manner. However, the computation of successor-tree-consistent deadlines has been found to be computationally intensive [87]. Later, Ilavarasan et al. in [37], Ilavarasan et al. in [36], Daoud et al. in [20] and Bittencount et al. in [12] presented the list-based heuristic schemes named *High-Performance Task Scheduling* (*HPS*), *Performance Effective Task scheduling* (*PETS*), *Longest Dynamic Critical Path* (*LDCP*) and *Lookahead*, respectively. All these schemes outperform *HEFT* with respect to *makespan*, albeit at the cost of higher time complexity. In 2014, Arabnejad et al. [5] proposed *Predict Earliest Finish Time* (*PEFT*). 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. *PEFT* outperforms *HEFT*, *CPOP*, *HPS*, *PETS*, *LDCP*, *Lookahead* and others [54] over a majority of test cases while incurring quadratic time complexity in terms of the number of tasks, similar to *HEFT*. Since then, *PEFT* has been popularly used and referred by researchers in different domains such as Cloud Computing [25], Grid Computing [42], and Embedded Systems [112]. Recently, the strategies *PPTS* [24] and *PSLS* [112] have been proposed as enhancements over the *PEFT* algorithm. Authors in [3] proposed PALG, an extension to the *HEFT* algorithm.

2.5.2 Monetary Cost Minimization Real-time Scheduling Policies

In recent years, monetary cost-aware scheduling schemes have been extensively studied especially with service-oriented computing models like cloud computing. A comprehensive survey of task graph scheduling schemes in a cloud environment may be found in [91]. The authors in [50,93] studied the monetary cost-aware real-time task graph scheduling problem for cloud computing platforms. In [93], two algorithms namely *L-ACO* and *ProLis* have been proposed. While the first algorithm *L-ACO* is based on ant colony optimization, the second scheme *ProLis* uses list scheduling. At each task node, *ProLis* estimates the computational and communication workload for the remaining task graph and uses these

estimates to distribute an application's total available slack to individual task nodes by assigning an appropriate sub-deadline to each of them. Each task is then mapped to the cheapest Virtual Machine (VM) while taking care that its sub-deadline is not violated. If no sub-deadline satisfying assignment is found, the task is allocated to the fastest available VM. A noteworthy aspect of this algorithm is that, although it takes inter-VM data transmission times into account, prices corresponding to data transmissions have been ignored. The authors in [50] proposed an energy and monetary cost-aware scheduling scheme for real-time applications. Similar to the previous scheme, this work also first distributes the application-wide slack to individual task nodes so that a minimal monetary cost task-to-VM assignment can be realized for all tasks while not violating the overall application deadline. The next phase attempts to minimize energy consumption by trying to merge subsets of two or more tasks into a single VM such that data transmission times are reduced and the deadline is still satisfied.

2.5.3 Energy-aware Real-time Scheduling Strategies

A detailed survey on different energy-aware scheduling schemes for real-time systems may be found in [7]. Weiser et al. in [90] first introduced an energy-saving strategy by using fine-grain control of processor speed by an operating system scheduler. The primary idea is to monitor processor idle time to reduce energy consumption by lowering clock speed and idle time to a minimum. Subsequently, Yao et al. in [106] analyze offline and online policies to schedule tasks with arrival times and deadlines on a uniprocessor system with minimum energy consumption. These works have been extended further in [49, 108] to minimize processor energy consumption while still satisfying the deadline for task execution. The problem of minimizing energy consumption of a real-time application with precedence-constrained non-preemptive tasks has been solved recently in a number of studies [47, 48]. However, these studies mostly focused on homogeneous multiprocessors with shared memory. The authors in [35, 84, 95, 98, 99] studied the problem of minimizing energy consumption of real-time systems assuming a heterogeneous computation platform. However, these works ignored inter-processor data communication costs. The authors in [99] proposed two algorithms *NDES* and *GDES*. Given a heterogeneous platform with distinct task-to-processor energy dissipation affinities, *NDES* solved the problem of energy-aware processor allocation for each task within a stipulated deadline. *GDES* solved the same problem for a DVFS-

enabled (Dynamic Voltage and Frequency Scaling) processor platform. In [95], the authors have presented an algorithm called *DECM* which utilizes the slack available between the finish time of the sink task and the deadline of a DAG, in order to minimize processor operating frequencies. Further, they extended *DECM* to a new policy called *DUECM*. *DUECM* improves on *DECM* by utilizing the slack between adjacent tasks assigned onto the same processor, while still meeting deadline of the DAG. Similarly, the work in [35] has also developed DVFS-enabled scheduling algorithms to perform energy optimization. The task graph scheduling strategies in [84, 98] used a combination of both processor slowdown (DVFS) and opportunistic processor low-powering (DPM; Dynamic Power Management) for overall system energy minimization. In spite of the above efforts, there still exists a severe dearth of research works which have attempted to address the problem of energy optimization for precedence-constrained real-time task graphs in a heterogeneous and distributed setting where inter-processor data communication costs cannot be ignored.

2.5.4 Scheduling Multiple Independent DAG Applications

A detailed survey on multiple independent task graph scheduling schemes for distributed systems may be found in [31]. The authors in [111] proposed two policies to address the problem of scheduling multiple independent DAGs on heterogeneous computing systems. While the first policy attempts to minimize the overall schedule length, the second policy aims at reducing unfairness in the slowdowns experienced by each individual DAG within the given set of DAG. Hsu et al. in [32] extended the first policy and proposed an algorithm called Online Workflow Management (*OWM*), targeted towards dynamically arriving task graph applications. Similarly, Arabnejad et al. in [4] extended the second policy in [111] and presented an algorithm called Fairness Dynamic Workflow Scheduling (*FDWS*) for the online scheduling of dynamic task graphs. However, none of the above works are applicable to task graphs having real-time constraints. Hu et al. in [33] presented an approach for scheduling a set of periodic precedence-constrained real-time task graph applications on safety-critical time-triggered systems. In this system, they assumed that task-to-processor allocations were known beforehand. The objective is to generate a schedule which satisfies the deadlines of all instances of every application. However, this work focuses only on meeting the deadlines of all application instances and does not endeavor to optimize additional performance metrics related to energy, cost, temperature, reliability, security, etc.

2.5.5 Security-aware Real-time Scheduling Strategies

In the last two decades, researchers have made significant efforts towards developing security-aware scheduling strategies using for both *dynamic* [101, 104, 105] as well as *static* [94, 102, 103] approaches. Authors in [101, 104, 105] proposed a family of *dynamic security-aware scheduling strategies* for platforms ranging from uni-processor systems [105] and multi-core homogeneous systems [101,104], to heterogeneous distributed systems [103,104]. While some of these studies focus on independent tasks set [101,103–105], the others have dealt with applications modeled as precedence-constrained task graphs [104]. Xie et al. in [105] proposed a real-time scheduling policy with security awareness for independent tasks on uni-processor systems. In this, they proposed a security overhead model to estimate the computation time overhead of commonly used security services such as confidentiality, integrity, and authentication. Authors in [101] proposed a scheduling algorithm named *SAREC* for independent tasks on multi-core homogeneous systems. The scheduler accepts a new dynamically arriving task only if its deadline and minimum security requirements can be guaranteed. Once a task is accepted, *SAREC* attempts to strengthen its security strength while ensuring that this operation does not lead to deadline violations of any task in the system including the newly accepted task. Targeting independent task sets and heterogeneous platforms, a scheduling strategy named *SATS* has been presented in [103]. The goal of *SATS* is to decide a task allocation while satisfying security requirements such that average response time is minimized. Authors in [104] proposed two *dynamic* security-aware soft real-time resource allocation schemes called *TAPADS* and *SHARP*. *TAPADS* is designed for parallel DAG-structured applications (where inter-task communication times are ignored) on homogeneous multi-processor systems, whereas *SHARP* is targeted for independent tasks on heterogeneous systems. The objective of these two schemes is to maximize the quality of security and the probability of meeting application deadlines.

The dynamic run-time scheduling schemes as mentioned above, usually achieve higher average performance gains and are more flexible compared to static schemes (which perform scheduling at design time). These relatively superior capabilities associated with dynamic strategies are achieved through the use of additional run-time information such as actual execution times of recently completed tasks, instantaneously available channel bandwidths, actual energy dissipation etc [72]. However, these superior capabilities are achieved at the cost of reduced predictability (in terms of reasonably accurate bounds on minimum perfor-

mance guarantees) which static schemes are able to deliver. For safety-critical hard real-time cyber-physical systems (as is the focus of this thesis), predictability in terms of security guarantees as well as performance and timeliness, is usually deemed to be more important than performance [69]. Hence, for safety-critical RT-CPSs such as automotive/avionic systems, static scheduling is often the desired choice [73].

Xie and Qin in [102] proposed a *static scheduling* strategy *SASES* for independent periodic tasks on uni-processor systems, which accounts for both timing and security requirements. Given the risk probabilities associated with the execution of tasks on a set of available processors, Tang et al. in [94] proposed a security-driven *static list scheduling strategy* for performance-sensitive non real-time DAG-structured applications. The strategy attempts to minimize schedule length while ensuring that the overall risk probability related to the execution of tasks in the DAG remains within a stipulated bound. From the above discussion, it may be highlighted that there exists a severe dearth of security-aware static real-time DAG scheduling algorithms.

2.5.6 Temperature-aware Scheduling Strategies

The introduction of sub-micron VLSI advancements have led to highly dense multi-million gates per chip, where power dissipation rates and thermal management have become critical design issues [82]. Unconfined temperature surges may increase cooling costs and decline system performance and life expectancy. Therefore, the processor's temperature management has become a topic of great concern for researchers and practitioners over the past few years. A survey of a few important temperature-aware scheduling techniques for multi-core processing systems may be found in [78]. Huang et al. in [34] proposed two throughput management policies for uni-processor systems with given temperature limits. The first policy shuffles the processor between sleep and active states to meet the goal. The second policy recommends a scheduling technique considering the task's thermal characteristics and the processor's sleep/active levels. Bampis et al. in [8] presented the makespan minimization problem under a given temperature bound. They studied the temperature-aware scheduling problem for systems consisting of unit-length independent tasks and homogeneous processors with known thermal characteristics. Authors in [10] addressed a thermal-aware global multiprocessor scheduling algorithm to increase the number of finished tasks. They also examine the competitive ratios of a large class of scheduling techniques as a function of the

processors' cooling factors. In [113], Zhou et al. presented a heuristic strategy for scheduling independent tasks with reliability and temperature constraints on heterogeneous platforms to minimize schedule length. A few recent scheduling techniques on this topic which target independent tasks on homogeneous and heterogeneous systems may be found in [57,64] and [52,75], respectively.

Research works on the temperature-aware scheduling of precedence-constrained task graphs are relatively fewer. Sheikh et al. in [76] proposed a temperature-aware scheduling algorithm to minimize the makespan of DAG-structured applications. In [77], the same authors also proposed a DAG scheduling technique for simultaneously optimizing makespan, energy and temperature. However, these works are focused towards homogeneous multi-cores.

2.6 Summary

This chapter started with a brief overview of the basic terms and definitions of real-time systems, followed by a literature survey of various scheduling algorithms for real-time 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 an efficient real-time DAG-scheduling framework that attempts to minimize a *generic penalty function*. The designed penalty function can be amicably adapted towards its deployment in various application domains such as real-time cyber-physical systems.

Chapter 3

PRESTO: A Penalty-aware Real-time Scheduler for Task Graphs on Heterogeneous Platform

3.1 Introduction

In the last chapter, we discussed various scheduling algorithms for real-time systems with the consideration of different design parameters. Most real-time scheduling algorithms typically attempt to optimize performance of the system with respect to a set of one or more chosen resource parameters while satisfying deadline constraints. Resource optimization objectives related to many commonly used parameters like energy, monetary cost, reliability, temperature, power, etc. can many-a-times be defined as functions of the specific task-to-processor and message-to-communication channel mappings, associated with a given schedule. Thus, for a given task or message we could say that a task (message) consumes lower energy or monetary cost on a processor (communication channel) compared to another. Inspired through these observations, we felt the usefulness of the design of real-time scheduling strategies which attempt to optimize generic objective functions of the above nature, because such objective functions can generally be adapted to model many common real-world resource parameters like energy, monetary cost, security, etc., with moderate effort. In this chapter, we propose an efficient scheduling framework for executing a real-time task graph on a distributed platform consisting of a set of fully connected heterogeneous processors. The objective of the scheduling framework is to minimize a *generic penalty function* which

can be amicably adapted towards its deployment in various application domains such as real-time embedded systems, cloud/fog computing, industrial automation and IoTs, smart grids, automotive and avionic systems, etc.

The main contributions of this work are summarized as follows:

1. We have first encoded the problem as a constraint satisfaction problem and then developed an efficient list-based heuristic scheduling algorithm called *Penalty-aware REal-time Scheduler for Task graphs on heterOgeneous platforms* (*PRESTO*), to generate a minimal penalty deadline-meeting static schedule. The proposed static heuristic solution *PRESTO* is scalable to large real-world scenarios. Theoretical analysis reveals that *PRESTO* incurs comparatively low, polynomial time scheduling overheads. This analysis has been supported through experimental evaluations which show that *PRESTO* consumes less than 25 *ms* of execution time for benchmark task graphs with up to 250 task nodes on platforms consisting of 32 heterogeneous processors.

2. Experiments have been conducted using real-world benchmark task graphs namely, Gaussian elimination and Epigenomics. Results show that the proposed algorithm works equally efficiently for both benchmarks, over diverse variations in different parameters such as number of tasks, Communication-to-Computation Ratio (CCR), number of processors and degree of heterogeneity.

3. The practical applicability of *PRESTO* in diverse scenarios has further been exhibited by using the scheme in two different real-world case studies. In the first case study, *PRESTO* has been used to schedule and map an (automotive) *adaptive cruise control* application such that energy consumption is minimized. In the second case study, *PRESTO* has been used to minimize the total monetary cost involved in the execution of an *intelligent surveillance application* running in a fog environment.

4. *PRESTO* has been designed to converge to a penalty oblivious *makespan* minimizing scheduling strategy called *Minimum Makespan Scheduler for Heterogeneous platforms* (*MMSH*) in worst-case scenarios when deadline satisfaction becomes more important than the incurred penalty. Experimental evaluation and comparison has shown that *MMSH* is able to outperform state-of-the-art algorithms *HEFT* [85], *PEFT* [5], *PPTS* [24], *PSLS* [112] and *PALG* [3], in terms of *makespan*.

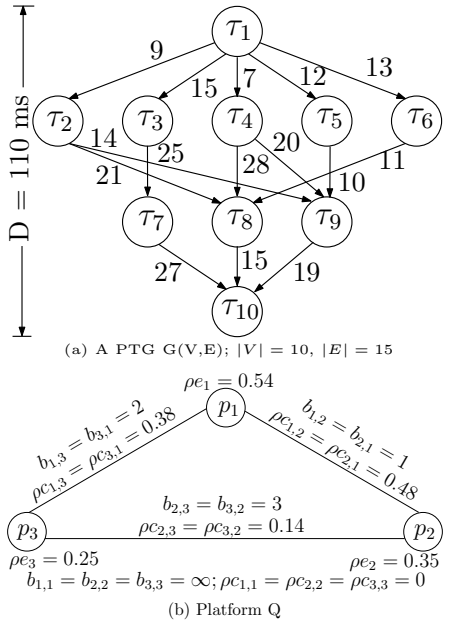
5. In addition to the simulation-based experiments, a simple proof-of-concept implementation of the proposed work has been conducted on a real platform consisting of two heterogeneous processors *ATmega328p* (*Arduino Uno*) and *ATmega2560* (*Arduino Mega*) inter-connected through their serial ports.

This chapter is organized as follows. In the next section, we present the system model and problem statement, while Section 3.3 describes the problem formulation. In Section 3.4, we present the proposed scheduling policy. Section 3.5 provides the experimental results. Section 3.6 presents a prototype real-platform implementation. The generic applicability of our proposed methodology is illustrated using two real-world case studies in Section 3.7. Finally, we conclude in Section 3.8.

3.2 System Model

The system under consideration consists of a real-time application having a deadline D, modeled as a Precedence-constrained Task Graph (PTG), to be scheduled on a platform consisting of a set of fully connected heterogeneous processors. Fig. 3.1a depicts an example PTG represented as a DAG $G(V, E)$, where the set of vertices $V = \{\tau_1, \tau_2, \ldots, \tau_{|V|}\}$ represents task nodes and the set of edges E represents precedence constraints between task pairs. Edge $e_{i,j}$ is labeled with a positive weight $data_{i,j}$ (e.g., $data_{1,4} = 7$) representing the size of this output.

The platform $P = \{p_1, p_2, \ldots, p_{|P|}\}$ consists of $|P|$ heterogeneous processors. The processors are fully interconnected through a set of $(|P| \times (|P|-1)/2)$ bidirectional communication links having heterogeneous (potentially distinct) bandwidths (refer Fig. 3.1b). A matrix B of size $|P| \times |P|$ is used to capture the bandwidths between all pairs of processors. An element $b_{m,n}$ of B denotes the data transfer rate between processors p_m and p_n. Obviously, as the links are bidirectional, $b_{m,n} = b_{n,m}$ (e.g., $b_{2,3} = b_{3,2} = 3$). The processors being heterogeneous, each task may have possibly distinct Worst Case Execution Times (*WCETs*) on the different processors. The *WCETs* of the tasks on different processors is stored in a matrix $\mathcal{W}$ of size $|V| \times |P|$, as shown in Fig. 3.1c. An element $\omega_{j,n} \in \mathcal{W}$ denotes the *WCET* of task τ_j on processor p_n (e.g., $\omega_{5,2} = 19$). Given $data_{i,j}$ and $b_{m,n}$, the data transmission/communication time ($c_{i,j}^{m,n}$) between task pairs τ_i (executing on processor p_m) and τ_j (executing on p_n; $\tau_j \in succ(\tau_i)$) may be calculated as:



(a) A PTG G(V,E); $|V| = 10$, $|E| = 15$

(b) Platform Q

Tasks	p_1	p_2	p_3
τ_1	16	16	23
τ_2	14	9	12
τ_3	20	43	17
τ_4	18	23	5
τ_5	8	19	27
τ_6	17	24	19
τ_7	15	24	31
τ_8	12	35	26
τ_9	29	34	10
τ_{10}	21	16	22

(c) Matrix $\mathcal{W}$ ($|V| \times |Q|$)

Figure 3.1: *(a) A sample DAG with* 10 *tasks; (b) A heterogeneous platform with* 3 *processors; (c) Execution time of each task on each processor.*

$$c_{i,j}^{m,n} = \begin{cases} 0, & if\ m = n \\ data_{i,j}/b_{m,n}, & otherwise \end{cases} \tag{3.1}$$

Equation 3.1 reveals that data transmission overhead is assumed to be negligible (i.e., $c_{i,j}^{m,n} = 0$) when τ_i and τ_j are mapped on to the same processor.

In this work, we assume that the execution of tasks on processors and data transmissions over communication links are associated with penalties which are levied by the system on completion of a task execution or data transmission. As both processors and communication links are heterogeneous, the penalization rates associated with the different processors/communication links are also heterogeneous. A vector $\rho e = \{\rho e_1, \rho e_2, \ldots, \rho e_{|P|}\}$ is used to represent the penalization rates associated with task executions on different processors and a matrix $\rho c = \{\rho c_{1,1}, \ldots, \rho c_{1,|P|}, \ldots, \rho c_{|P|,1}, \ldots, \rho c_{|P|,|P|}\}$ is used to represent the penalization rates associated with data transmissions over different communication links. The exact values of the penalization rates will vary depending on the actual parameters that these penalization rates measure in a real-life problem setting to which the proposed solution procedure is applied. For example, in a service-oriented computing environment such as fog computing, *penalties* may relate to the actual *monetary expenditure* incurred in, (i) transmitting the data and/or code associated with a task onto a particular fog node,

and (ii) executing the task on that node. Similarly, in a problem setting such as execution of real-time tasks in a heterogeneous multiprocessor environment, *penalties* may be equated to distinct *energy dissipation* overheads associated with the transmission and execution of tasks on different processors.

In the example system depicted in Fig 3.1, we assume the application deadline to be 110. The penalty per unit processing time spent on the processors p_1, p_2, and p_3 are 0.54, 0.35, and 0.25 respectively, while the penalty for each unit of data transfer through the communication links $l_{1,2}$, $l_{1,3}$, and $l_{2,3}$ are 0.48, 0.38, and 0.14, respectively. For simplicity, we have ignored the units of all the parameters.

Based on the above system model, we define our problem statement as follows:

Problem Statement: *Given a real-time application graph G and a fully connected heterogeneous distributed processing platform P, determine a feasible static schedule which minimizes aggregate penalty while satisfying resource, deadline and dependency constraints.*

3.3 Constraint Satisfaction Problem Formulation

In order to present a formal model of the problem and its objective, we have formulated it as a constraint optimization problem. First, let us consider a set of binary decision variables: $x = \{x_{j,n,t} : j = 1, 2, \ldots, |V|; n = 1, 2, \ldots, |P|; t = 0, 1, \ldots, D\}$. $x_{j,n,t} = 1$ denotes that task τ_j is allocated for execution on processor p_n starting from the t^{th} time step; $x_{j,n,t} = 0$, otherwise. We now present the objective function and the required constraints on the binary variables to model the scheduling problem.

Objective Function: Our objective is to minimize the overall scheduling penalty associated with the execution of tasks and data communication, by selecting appropriate task-processor mappings and execution start times. The objective function can be written as:

$$\min \sum_{j=1}^{|V|} \sum_{n=1}^{|P|} \sum_{t=0}^{D} \Big[x_{j,n,t} \times \omega_{j,n} \times \rho e_n + \sum_{i \in pred(j)} \sum_{m=1}^{|P|} \sum_{t'=0}^{t-1} x_{j,n,t} \times x_{i,m,t'} \times c_{i,j}^{m,n} \times \rho c_{m,n} \Big] \quad (3.2)$$

It may be noted that, if both tasks τ_i and τ_j are mapped to the same processor then the communication time is assumed to incur negligible overhead, i.e., $m = n | c_{i,j}^{m,n} = 0$.
Next, we present the constraints in equations 3.3 - 3.6.

Unique Start Time Constraints: The start time of each task should be unique; that is,

$$\forall j \in [1, |V|], \sum_{n=1}^{|P|} \sum_{t=0}^{D} x_{j,n,t} = 1 \tag{3.3}$$

The above constraint indicates that each task τ_j must start its execution at a unique time step t on a distinct processor p_n.

Dependency Constraints: The dependency constraints enforce satisfaction of the precedence relationships among task nodes of a task graph. That is, the start time of a task τ_j must be greater or equal to the latest time of arrival of an output from one of the predecessors of τ_j. For any given predecessor say τ_i, this arrival time is given by the summation of the finish time of τ_i and the communication time from τ_i to τ_j.

$$\forall(\tau_i, \tau_j) \in E, \sum_{n=1}^{|P|} \sum_{t=0}^{D} t \times x_{j,n,t} \geqslant \sum_{n=1}^{|P|} \sum_{t=0}^{D} \sum_{m=1}^{|P|} \sum_{t'=0}^{D} x_{i,m,t'} \times x_{j,n,t}(t' + \omega_{i,m} + c_{i,j}^{m,n}) \tag{3.4}$$

Resource Constraints: Processor usage bounds must be satisfied at each time step. Any processor p_n can execute at most one task at a given time. In this regard, it may be noted that a task τ_j can only be executing on processor p_n at time t, if it has started at most $t - \omega_{j,n} + 1$ time steps earlier. Thus, the expression $\sum_{l=t-\omega_{j,n}+1}^{t} x_{j,n,l}$ assumes a value of '1' when τ_j starts on p_n between time steps $[t - \omega_{j,n} + 1, t]$. This constraint can therefore be written as:

$$\forall n \in [1, |P|], \forall t \in [0, D], \sum_{j=1}^{|V|} \sum_{l=t-\omega_{j,n}+1}^{t} x_{j,n,l} \leqslant 1 \tag{3.5}$$

Deadline Constraint: The deadline constraint enforces the exit task node τ_{exit} to complete execution on or before the application deadline D.

$$\sum_{n=1}^{|P|} \sum_{t=0}^{D} x_{exit,n,t}(t + \omega_{exit,n}) \leqslant D \tag{3.6}$$

The scheduling problem encoded as above amicably lends itself towards its computation using standard optimization tools like CPLEX [1]. However, the existence of a large number of decision variables and constraints makes the problem computationally highly complex. Hence, solution techniques using standard optimizers often take huge amounts of time and space even for moderate problem sizes with respect to number of tasks, heterogeneous processors, latency bounds, nature of inter-task dependencies, etc.

We reiterate here that the principal motivation towards encoding of the problem as above is the clarity it lends in understanding and appreciating the structure of the scheduling problem at hand. Such an understanding proves immensely useful in the design and analysis of efficient lower overhead heuristic strategies for the problem. We now present *PRESTO*, an efficient list-based greedy heuristic algorithm for the problem discussed above.

3.4 PRESTO: The Proposed Scheduler

The proposed algorithm *PRESTO* takes a task graph $G(V, E)$, application deadline D, and a target platform P as inputs. The objective is to map all tasks to appropriate processors in the heterogeneous platform such that the overall penalty in executing the task graph is minimized while ensuring that the sink node (and thereby, all other task nodes) completes execution before deadline D. *PRESTO* is an efficient list-based heuristic scheduling algorithm and consists of *two* phases namely, *initialization phase* and *allocation phase*. We now provide an overview of these two phases before discussing the detailed algorithm.

3.4.1 Initialization Phase

The *initialization phase* computes three different parameters which are used in the subsequent *allocation phase* namely, (1) *Optimistic Finish Time* $OFT[\tau_j, p_n]$, $1 \leq j \leq |V|$; $1 \leq n \leq |P|$, for each task-processor pair, (2) *A Rank value* $Rank[\tau_j]$ for each task τ_j, which is employed to construct a priority order among tasks. In the *allocation phase*, this priority list of tasks is used to determine the actual order in which tasks are considered for processor allocation and scheduling, and (3) *Minimum Estimated Penalty* value $MEP[\tau_j, p_n]$, $1 \leq j \leq |V|$; $1 \leq n \leq |P|$, for each task-processor pair. The *OFTs* of all task-processor pairs and the *Ranks* of all tasks are determined through a function $OFT\_Rank()$ (refer Algorithm 1).

OFT$[\tau_j, p_n]$: *Optimistic Finish Times* for different task-processor pairs are represented as a table where rows indicate tasks and columns indicate processors as shown in Table 3.1, for the example system presented in Fig. 3.1. $OFT[\tau_j, p_n]$ essentially provides an estimate of the total time required to complete execution of the current task τ_j on processor p_n, along with execution of the remaining yet to be allocated task nodes. It may be observed from steps 5 and 6 of Algorithm 1 that $OFT[\tau_{exit}, p_n] = \omega_{exit,n}$, for the exit task. For all other tasks, $OFT[\tau_j, p_n]$ is computed in a two-level process as depicted in step 8. At the

first level, given a successor task τ_k (of τ_j) and a certain processor p_r (say), an estimate of the total processing time from the start of the execution of τ_j on p_n to the completion of the exit task τ_{exit}, is determined. This is a recursive step and the value is obtained as the summation of: (i) the *Optimistic Finish Time* ($OFT[\tau_k, p_r]$) corresponding to the execution of the successor τ_k on processor p_r, (ii) the *execution time* ($\omega_{j,n}$) of τ_j on processor p_n, and (iii) *communication time* ($c_{j,k}^{n,r}$) required to transmit the output of τ_j to τ_k. At the second level, $OFT[\tau_j, p_n]$ is obtained as the maximum over the minimum estimated processing times, considering the execution of τ_j and all its successor tasks as step 8 reveals. The value of $OFT[\tau_j, p_n]$ obtained through this process is potentially subject to further scaling depending on the value of $Rank[\tau_j]$ and the values of the Ranks of the successors of τ_j, as discussed below.

Algorithm 1: $OFT\_Rank(G, P)$

Input: Task graph $G(V, E)$, processor set P
Output: Determines *Optimistic Finish Time* for all task-processor pairs and *Rank* of all tasks

1 Construct a queue *openQ*; Initialize $openQ = \{\tau_{exit}\}$;
2 **while** $openQ \neq empty$ **do**
3 Dequeue a task τ_j from *openQ*;
4 **for** *each processor* p_n *in* P **do**
5 **if** $\tau_j = \tau_{exit}$ **then**
6 $OFT[\tau_j, p_n] = \omega_{j,n}$;
7 **else**
8 $OFT[\tau_j, p_n] = \max_{\tau_k \in succ(\tau_j)} \Big[\min_{p_r \in P} \{OFT[\tau_k, p_r] + \omega_{j,n} + c_{j,k}^{n,r}\} \Big]$;
9 $Rank[\tau_j] = \sum_{n=1}^{|P|} OFT[\tau_j, p_n]/|P|$;
10 $maxSuccRank = \max_{\tau_k \in succ(\tau_j)} Rank[\tau_k]$;
11 **if** $Rank[\tau_j] \leqslant maxSuccRank$ **then**
12 **for** *each processor* p_n *in* P **do**
13 $OFT[\tau_j, p_n] = OFT[\tau_j, p_n] \times \frac{maxSuccRank + \delta}{Rank[\tau_j]}$;
14 $Rank[\tau_j] = maxSuccRank + \delta$;
15 Enqueue all immediate predecessors of τ_j in *openQ*;
16 **return** [*OFT*, *Rank*];

Rank[τ_j]: Processor allocation of tasks occur in non-increasing order of *Rank* values. This is intended to serve two important objectives, the first of which is mandatory while

the second is desirable. **Objective 1**: Ensuring that all ancestors of a given task are always considered for processor allocation before the task itself, so that precedence constraints associated with the task graph are never violated. **Objective 2**: Tasks having relatively higher total estimated workloads corresponding to the remaining (still to be allocated) tasks in the task graph should be considered earlier for processor allocation. Given the OFT values of a task τ_j on each processor, its average is considered as the *Rank* ($Rank[\tau_j]$) of τ_j (step 9 of Algorithm 1). This scheme naturally takes care of *objective* 2 as OFT values represent remaining estimated workloads, as discussed above. However sometimes, the initially calculated *Rank* ($Rank[\tau_j]$; calculated in step 9) of a task τ_j may possibly be smaller than the maximum of the *Ranks* of $\tau_j's$ successors. In this case, schedule generation in the non-increasing order of *Ranks* will result in violation of *objective* 1.

In order to rectify this situation, the initially calculated OFT values of τ_j are appropriately scaled (as shown in steps 12 and 13) so that the resulting *Rank* value of τ_j becomes equal to $\max_{\tau_k \in succ(\tau_j)} Rank[\tau_k] + \delta$, where δ is a small constant (in our work we have assumed $\delta = 0.01$) and $succ(\tau_j)$ denotes the set of all immediate successors of task τ_j. After obtaining the *Rank* values, a priority list $taskList = \{\tau_{s1}, \tau_{s2}, \ldots, \tau_{s|V|}\}$ of tasks is generated such that: $Rank[\tau_{s1}] \geq Rank[\tau_{s2}] \geq \cdots \geq Rank[\tau_{s|V|}]$. Table 3.1 lists the *Rank* values for the ten tasks of the example system depicted in Fig. 3.1.

Table 3.1: *OFT, Rank, and MEP values corresponding to the example system depicted in Fig. 3.1*

Tasks	OFT			Rank	MEP		
	p_1	p_2	p_3		p_1	p_2	p_3
τ_1	72	86.3	86.5	81.61	19.22	15.18	15
τ_2	53	60	55.5	56.17	18.22	11.8	11
τ_3	56	83	65.3	68.11	28.8	29.05	17.5
τ_4	60	74	52	62	21.52	16.98	9.25
τ_5	45	54.3	59	52.78	14.22	15.12	14.75
τ_6	50	68	57.5	58.5	23.27	20.91	16.75
τ_7	36	40	53	43	18.73	14	13.25
τ_8	33	51	47	43.67	14.83	17.85	12
τ_9	50	50	32	44	24.77	17.5	8
τ_{10}	21	16	22	19.67	11.34	5.6	5.5

MEP$[\tau_j, p_n]$: As the name suggests, Minimum Estimated Penalty $MEP[\tau_j, p_n]$ provides an estimate of the minimum penalty considering all tasks on every available processor over all paths, starting from the execution of τ_j on p_n to the completion of τ_{exit}. The MEP value

of τ_j on processor p_n is determined through a recursive definition by traversing the graph backward from τ_{exit} to τ_j, as depicted in equation 3.7.

$$MEP[\tau_j, p_n] = \begin{cases} \omega_{exit,n} \times \rho e_n, & \text{if } \tau_j = \tau_{exit} \\ \min\limits_{\tau_k \in succ(\tau_j)} \Big[\min\limits_{p_r \in P} \{MEP[\tau_k, p_r] + \omega_{j,n} \times \rho e_n + c_{j,k}^{n,r} \times \rho c_{n,r}\} \Big], & otherwise \end{cases} \tag{3.7}$$

It may be noted that the data transmission penalty vanishes ($\rho c_{n,r} = 0$) when τ_j and its successor τ_k are executed on the same processor (i.e., $p_n = p_r$). The $MEP[\tau_j, p_n]$ values for different task-processor pairs of the example system in Fig. 3.1, is represented in Table 3.1.

3.4.2 Allocation Phase

The objective of the *allocation phase* is to generate a real-time static schedule obtained by sequentially determining: (i) a *processor allocation* $aloc[\tau_j]$, and (ii) an *actual start time* $AST[\tau_j]$ for each task τ_j, in the order prescribed by priority list *taskList*, such that overall penalty is minimized. In our system, a generated schedule is referred to as *valid*, if the exit task τ_{exit} finishes its execution on or before the stipulated deadline D (that is, *makespan* $\leq D$). The *allocation phase* is essentially an iterative process which continues either until a *valid* schedule is successfully generated in a certain iteration, or finally exits with a minimal *makespan invalid* schedule. Each new iteration uses knowledge of the actual *makespan* obtained for the *invalid* schedule generated in the previous iteration and calculates the *Deadline Overshoot* (DO) suffered, as:

$$DO = makespan - D \tag{3.8}$$

At a given iteration, tasks in *taskList* are allocated sequentially in order. The task (say, τ_j) to be scheduled next, maybe potentially allocated to any processor p_n on the heterogeneous platform. However, selection of the actual processor is made by judiciously considering the following two parameters, for each such possible allocation: (i) $MEP[\tau_j, p_n]$: *Minimum Estimated Penalty* of τ_j on a potential processor p_n (refer equation 3.7), and (ii) $ESL[\tau_j, p_n]$: *Estimated Schedule Laxity* of τ_j on p_n. $ESL[\tau_j, p_n]$ is a throttled estimate of the spare time that should remain before deadline D and after completion of the sink node τ_{exit}, for the case when all tasks in *taskList* before τ_j, have already been scheduled and τ_j is restricted to execute on p_n. Hence, $ESL[\tau_j, p_n]$ is calculated as the throttled difference between deadline D and the current estimate of the *makespan* considering τ_j on p_n, as follows:

$$ESL[\tau_j, p_n] = D - (EST[\tau_j, p_n] + OFT[\tau_j, p_n]) \times TF^{-1} \tag{3.9}$$

where $EST[\tau_j, p_n]$ denotes the *Effective Start Time* of τ_j on p_n and TF is the Throttling Factor ($0 < TF \leq 1$). The exact mechanism for determining $EST[\tau_j, p_n]$ is discussed later; refer equation 3.12). The value of TF is initialised to 1, and is decremented after each unsuccessful iteration in proportion to the deadline overshoot (DO) corresponding to that iteration:

$$TF = \begin{cases} TF - \epsilon, & if \lceil \frac{DO}{D} \times 100 \rceil = 1 \\ TF - \epsilon \times \log_2 \lceil \frac{DO}{D} \times 100 \rceil, & otherwise \end{cases} \tag{3.10}$$

where ϵ is a constant and set to 0.01 as a default value in our experiments (for more detail, refer to Experiment 3).

Among available allocation options, a task τ_j is first considered for allotment within that subset of processors say, $P_S = \{p_{s1}, p_{s2}, \ldots, p_{sr}\}$, $(1 \leq r \leq |P|)$ for which $ESL[\tau_j, p_n]$ is non-negative. It may be noted that, as the number of iterations grow and the value of ESL gets increasingly throttled, the number of processors which remain eligible towards inclusion in P_S, reduces progressively. Now, τ_j is actually allocated to that processor p_{si} in P_S for which $MEP[\tau_j, p_{si}]$ is minimal. Hence, higher the iteration number larger is the penalty associated with the schedule generated in that iteration. However, if $ESL[\tau_j, p_n]$ is negative for all available processors (that is, $P_S = NULL$), τ_j is allocated on that processor for which $ESL[\tau_j, p_n]$ is minimally negative. Therefore, the processor finally allocated to task τ_j may be represented as:

$$aloc[\tau_j] = \begin{cases} p_n | MEP[\tau_j, p_n] = \min\limits_{p_\lambda \in P_S} MEP[\tau_j, p_\lambda], & if\ P_S \neq \phi \\ p_n | ESL[\tau_j, p_n] = \max\limits_{p_\lambda \in P} ESL[\tau_j, p_\lambda], & if\ P_S = \phi \end{cases} \tag{3.11}$$

MMSH Algorithm: It may be noted that the *minimum makespan* schedule is naturally produced in a situation where the processor set $P_S = NULL$ for all tasks in a given iteration. In such a scenario, the *PRESTO* algorithm terminates with this *minimum makespan schedule*. Further, it may be also observed that a non-real time minimum *makespan* scheduler similar to *PEFT* [5] can be easily derived from *PRESTO* as follows. In the *initialization phase*, we compute the OFT and *Rank* values and also generate the prioritized task list *taskList*. The *allocation phase* runs for a single iteration in which each task τ_j is assigned to that processor p_n for which the expression $EST[\tau_j, p_n] + OFT[\tau_j, p_n]$ is minimal. We refer to this algorithm as the *Minimum Makespan Scheduler for Heterogeneous Platforms (MMSH)*.

Effective Start Time: The calculation of $EST[\tau_j, p_n]$ is based on the assumption

that the following information is already known: (i) $aloc[\tau_i]$, the processor allocated to each predecessor task τ_i of task τ_j ($aloc[\tau_i] = p_m$, say), (ii) the actual finish time of each predecessor task τ_i ($AFT[\tau_i]$) on its allocated processor p_m, and (iii) the earliest time at which a particular processor (say, p_n) becomes available for execution ($avail[n]$). The three assumptions mentioned above are *valid* since task-to-processor allocations are conducted strictly based on *Rank* prescribed by the priority order in *taskList*. $EST[\tau_j, p_n]$ is defined as:

$$EST[\tau_j, p_n] = \begin{cases} 0, & \text{if } \tau_j = \tau_{entry} \\ \max\{avail[n], \max\limits_{\tau_i \in pred(\tau_j)} (AFT[\tau_i] + c_{i,j}^{m,n})\}, & otherwise \end{cases} \tag{3.12}$$

where $c_{i,j}^{m,n}$ is the time required to transmit the output of τ_i (executing on p_m) to τ_j (on p_n) as defined in equation 3.1. *Actual start time* of task τ_j on processor $aloc[\tau_j]$ is given by:

$$AST[\tau_j] = EST[\tau_j, aloc[\tau_j]] \tag{3.13}$$

Given the value of $AST[\tau_j]$, $AFT[\tau_j]$ is obtained as:

$$AFT[\tau_j] = AST[\tau_j] + \omega_{\tau_j, aloc[\tau_j]} \tag{3.14}$$

The allocation of tasks to processors is known from equation 3.11. Additionally, equation 3.13 provides the actual start times of the tasks. The total penalty (ρ_{presto}) can be computed by appropriately rewriting the objective function in equation 3.2 as depicted in the equation below:

$$\rho_{presto} = \sum_{j=1}^{|V|} \left\{ \omega_{j,aloc[\tau_j]} \times \rho e_{aloc[\tau_j]} + \sum_{\tau_i \in pred(\tau_j)} c_{i,j}^{aloc[\tau_i], aloc[\tau_j]} \times \rho c_{aloc[\tau_i], aloc[\tau_j]} \right\} \tag{3.15}$$

3.4.3 PRESTO Algorithm

Pseudocode of the *Penalty-aware REal-time Scheduler for Task graphs on heterOgeneous platforms* (*PRESTO*), is presented in Algorithm 2. Steps 1, 2 and 3 correspond to the *initialization phase* of *PRESTO*. As discussed in Section 3.4.1, these steps calculate OFT, MEP and *Rank* values; additionally, the tasks are stored in list *taskList* in non-increasing order of their *Ranks*. The remaining steps correspond to the *allocation phase*. The *while loop* in steps 5-7 iterates either until a minimal-penalty deadline meeting (*valid*) schedule is successfully produced by function *scheduleTasks()* in step 6 of algorithm 2, or terminates with

Algorithm 2: PRESTO (G, D, P)

Input: Task graph $G(V, E)$, deadline D and processor set P
Output: A *valid* schedule, which minimizes penalty

```
1  [OFT, Rank] = OFT_Rank(G, P);
2  Compute MEP values using equation 3.7;
3  Sort tasks in non-increasing order of Rank; store in taskList;
4  validSched = false, minLenSched = false, TF = 1;
5  while validSched = false and minLenSched = false do
6      [validSched, minLenSched, ρ_presto] = scheduleTasks(G, P, MEP, D, taskList);
7      Update TF using equation 3.10;
8  if validSched = false then
9      Output: Invalid schedule having min. makespan;
10 else
11     Output: Valid schedule having penalty = ρ_presto;
```

Algorithm 3: scheduleTasks $(G, P, MEP, D, taskList)$

Input: $G(V, E)$, P, MEP, D, $taskList$
Output: A *valid* schedule having minimal penalty or an *invalid* schedule having minimal *makespan*

```
1  minLenSched = true;
2  for each task in taskList do
3      Extract the first task (say, τ_j) from taskList;
4      for each processor p_n in P do
5          Compute ESL[τ_j, p_n] using equation 3.9;
6          P_S = Set of processors for which ESL[τ_j, p_n] ⩾ 0;
7      if P_S ≠ φ then
8          Assign τ_j on the processor p_n ∈ P_S for which MEP[τ_j, p_n] is minimal (refer
           equation 3.11);
9          minLenSched = false;
10     else
11         Assign τ_j on processor p_n ∈ P for which ESL[τ_j, p_n] is minimally negative (refer
           equation 3.11);
12 if AFT[τ_exit] ⩽ D then
13     Compute ρ_presto using equation 3.15;
14     return [true, minLenSched, ρ_presto];
15 return [false, minLenSched, φ];
```

the minimum *makespan* deadline violating (*invalid*) schedule, after atmost 100 unsuccessful iterations. This upper-bound of 100 on the number of iterations may be derived from equation 3.10. Production of a *valid* schedule at any given iteration of the *while* loop is indicated by the *truth* of the flag *validSched*. On the other hand, *truth* of the flag *minLenSched* indicates generation of the penalty-oblivious minimum *makespan MMSH* schedule.

The *scheduleTasks()* function presented in Algorithm 3, allocates an appropriate processor to each task in *taskList* considering both timing and penalty constraints as discussed in Section 3.4.2.

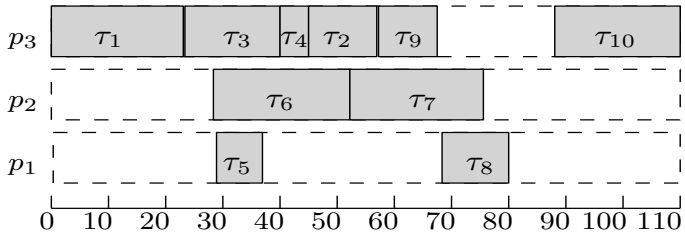
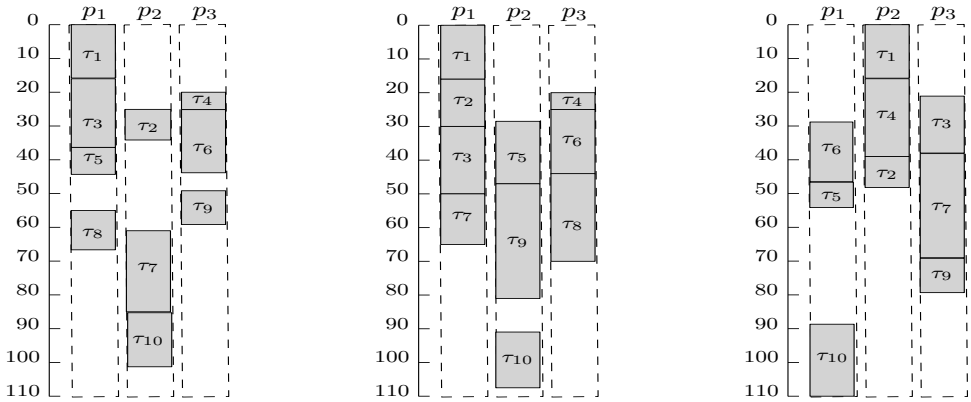


Figure 3.2: *Gantt chart of the schedule generated by PRESTO for the PTG in Fig. 3.1a: makespan* = 110, *penalty* = 74.03.



(a) *MMSH* **(b)** *PEFT* **(c)** *HEFT*

Figure 3.3: *Gantt charts depicting the schedules: (a) MMSH (makespan* = 101*); (b) PEFT (makespan* = 108*); (c) HEFT (makespan* = 110*), for the PTG in Fig. 3.1a.*

Example Continued: As mentioned above in Section 3.4.1, Table 3.1 lists the *OFT*, *MEP* and *Rank* values corresponding to the example system shown in Fig. 3.1. The priority list of tasks is obtained as: $taskList = \langle \tau_1, \tau_3, \tau_4, \tau_6, \tau_2, \tau_5, \tau_9, \tau_8, \tau_7, \tau_{10} \rangle$. The *allocation phase* for this example continues for 8 iterations and finally generates a *valid* schedule with

makespan 110 and penalty 74.03 as depicted in Fig. 3.2.

Using the same example system, we have also compared the *MMSH*, *PEFT* and *HEFT* algorithms by presenting Gantt charts for the schedules generated by them in Fig. 3.3. It may be observed that for this example, *MMSH* delivers the least *makespan* 101, while *PEFT* and *HEFT* produces schedules with *makespans* 108 and 110, respectively.

3.4.4 Complexity Analysis

The complexity of the *PRESTO* algorithm is composed of the complexities of its two constituent phases, namely the *initialization* and *allocation phases*. While steps 1-4 of the main function *PRESTO()* (refer Algorithm 2) comprises the *initialization phase*, steps 5-11 comprise the *allocation phase*. In the *initialization phase*, the computation of OFT and MEP considers each edge of the task graph exactly once and iterates over $|P|$ processors to find the minimum value. Hence, the overall complexity of generating the OFT and MEP tables require $O(|P|(|V| + |E|))$ time (steps 1 and 2). As the overhead of generating the *Rank* value for a single task is $O(|P|)$ (refer step 1), the total overhead of determining the *Ranks* for all tasks becomes $O(|V| \times |P|)$. The sorting operation in step 3 runs in $O(|V| \log |V|)$ time. Therefore, the complexity of the *initialization phase* is $O(|P|(|V|+|E|)+|V| \times |P|+|V| \log |V|)$. Assuming $|E|$ to be larger than $|V|$, $O(|V|log|V|)$ may be considered to be proportional to $O(|P| \times |E|)$ for even considerably large task graphs (containing upto a few 100 nodes). Hence, the overall complexity of the *initialization phase* may be expressed as $O(|E| \times |P|)$.

In the *allocation phase*, the *scheduleTasks()* function is called in a *while* loop (steps 5-7). The complexity of *scheduleTasks()* is primarily governed by the computational overhead of calculating $ESL[\tau_j, p_n]$ (refer equation 3.9; in step 5 of *scheduleTasks()*) for all task-processor pairs within the nested *for* loops (outer loop: steps 2-11; inner loop: steps 4-6). The complexity of computing $ESL[\tau_j, p_n]$ in turn, is primarily dependent on the overhead of determining $EST[\tau_j, p_n]$, as the other terms required in its calculation are either known or constant. It may be observed that calculation of EST for any given task-processor pair requires constant time calculations over all predecessors of the task and hence, incurs an overhead of $O(\#predecessors)$. However, the total number of predecessors over all tasks is same as the total number of edges in the task graph. Therefore, the amortized complexity for calculating $EST[\tau_j, p_n]$ (and also $ESL[\tau_j, p_n]$) becomes $O(|P|(|V| + |E|)/(|P| \times |V|)) =$

$O((|V| + |E|)/|V|) = O(|E|/|V|)$. Given this complexity, the total overhead incurred in calculating $ESL[\tau_j, p_n]$ for all task-processor pairs become $O(|E|/|V| \times |P| \times |V|) = O(|E| \times |P|)$. Thus, in the main function *PRESTO()*, the overhead corresponding to a single call to the *scheduleTasks()* function is $O(|E| \times |P|)$.

Observing equations 3.9, 3.10 and function *scheduleTasks()*, it may be inferred that *scheduleTasks()* can return $false$ for both flags *validSched* and *minLenSched*, at most 100 times. Hence, the number of iterations of the *while* loop (in steps 5-7 of *PRESTO()*) is upper bounded by 100. However in practice, *PRESTO* is able to restrict the number of iterations to a small value (typically, < 8; this observation is also validated through our experimental results (refer Fig. 3.6)). The above phenomena may be attributed to two reasons: (i) *PRESTO* delivers a *valid* schedule (making $validSched = true$) within a few iterations in most cases, and (ii) the expression: $\lceil \frac{DO}{D} \times 100 \rceil$ is often greater than 1, making the *throttling factor* (TF) decrease at a significantly faster rate than $0.01/iteration$ (refer equation 3.10). The remaining steps of *PRESTO()* take $O(1)$ time. Hence, the overall complexity of *PRESTO()* comprising overheads related to both its *initialization* and *allocation phases*, may be expressed as $O(|E| \times |P|)$.

The Algorithm *MMSH*, being very similar to *PRESTO* in its structure, incurs an overhead of $O(|E| \times |P|)$.

3.5 Experiments and Results

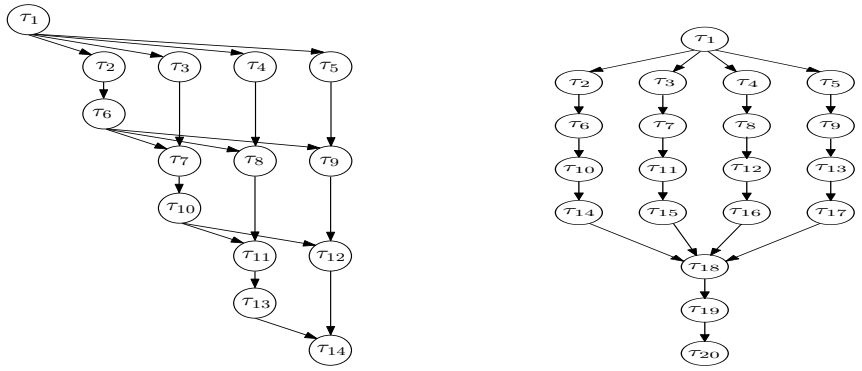
In this section, we have experimentally evaluated the performance of the proposed algorithms *PRESTO* and *MMSH* (a non-real-time makespan minimization algorithm obtained as a spin-off from *PRESTO*; refer Section 3.4.2) through an extensive set of simulation-based experiments, in two phases. Experiments 1-2 comprise the first phase. Here, the performance of *MMSH* has been compared against the well known state-of-the-art *makespan* reduction algorithms *HEFT* [85], *PEFT* [5], *PPTS* [24], *PSLS* [112] and *PALG* [3]. Experiments 3-5 comprise the second phase. Here, we measure the performance of *PRESTO* against variations in different input parameters and also provide comparative results with four slightly restricted implementations of *PRESTO*. In the next subsections, we explain the experimental setup and the performance metrics before presenting the detailed experimental results.

3.5.1 Experimental Setup

The experiments have been conducted using two real-world DAG-based parallel applications namely, Gaussian Elimination [85] and Epigenomics [39]. While Epigenomics is highly CPU-intensive, Gaussian Elimination is relatively more I/O-intensive.

Gaussian Elimination is an algorithm for solving systems of linear equations and its task graph representation is influenced by the size of the matrix (ν) corresponding to a given set of equations. The total number of task nodes and edges in a Gaussian Elimination graph is equal to $((\nu^2 + \nu - 2)/2)$ and $(\nu^2 - \nu - 1)$, respectively. As example, for the Gaussian Elimination task graph shown in Fig. 3.4a, matrix size $\nu = 5$ and therefore, the graph has 14 task nodes and 19 edges.

Epigenomics is a data processing pipeline that represents the execution of various genome sequencing operations. The size of an Epigenomics task graph is influenced by the number of parallel branches (ϑ). The total number of task nodes and edges in an Epigenomics task graph is equal to $(4\vartheta + 4)$ and $(5\vartheta + 2)$, respectively. As an example, for the Epigenomics task graph shown in Fig. 3.4b, the number of parallel branches $\vartheta = 4$ and therefore, the graph has 20 task nodes and 22 edges.



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

Figure 3.4: *Benchmark DAGs.*

Data Generation Framework: An exhaustive set of experiments have been carried out using randomly generated data sets obtained by carefully varying a set of parameters.

1. **Number of task nodes and edges**: *Matrix size* (for Gaussian Elimination) $\nu = \{6, 9, 14, 17, 22\}$; with these values of ν, five types hypothetical task graphs having distinct number of task nodes $|V|$ and edges $|E|$ are generated ($|V| = \{20, 44, 104, 152, 252\}$; $|E| =$

$\{29, 71, 181, 271, 461\}$). Similarly, *parallel branch* (for Epigenomics) $\vartheta = \{4, 10, 25, 37, 62\}$; task graphs having $|V| = \{20, 44, 104, 152, 252\}$ task nodes and $|E| = \{22, 52, 127, 187, 312\}$ edges, are generated. The values of ν and ϑ have been carefully chosen in order to be able to generate Gaussian Elimination and Epigenomics task graphs having the same number of task nodes in them. This approach helps towards better performance evaluation and comparison of the algorithms against Gaussian Elimination and Epigenomics task graphs having the same size (in terms of number of task nodes).

2. **Number of processors**: $|P| = \{4, 8, 16, 32, 64\}$.

3. **Task execution times**: Generation of execution times of all tasks on each heterogeneous processor is accomplished in three steps: (i) Selection of average execution time (denoted as $\overline{\omega_{DAG}}$) over all tasks in the task graph. Although for a given value of $\overline{\omega_{DAG}}$, individual tasks with vastly varying execution times can still possibly be generated, $\overline{\omega_{DAG}}$ helps towards providing a certain degree of control and determinism on the overall execution requirement of an application. The experiments conducted in this work use the following values of average execution time: $\overline{\omega_{DAG}} = \{40, 80, 120, 160, 200\}$. (ii) Given $\overline{\omega_{DAG}}$, the next step is to determine the average execution time (denoted as $\overline{\omega_j}$) over all processors, for each task node (τ_j) in the task graph. The $\overline{\omega_j}$ values of the task nodes are generated from normal distributions having mean $\mu = \overline{\omega_{DAG}}$ and various values of standard deviation σ ($= \{10, 20, 30\}$). Each distinct value of σ allows the generation of average task node execution times with a certain known degree of skewness among them. (iii) After determining $\overline{\omega_j}$ for each task τ_j, the final step is to generate the actual worst case execution time ($WCET$) (denoted as $\omega_{j,n}$) of τ_j, on each processor p_n. For any given task, the values of $\omega_{j,n}$ are determined from normal distributions having mean $\mu = \overline{\omega_j}$ and various values of standard deviation $\sigma = (\overline{\omega_j} \times \beta)$; here the parameter β is referred to as the heterogeneity factor and determines the degree of skewness among the execution times of a task on the different heterogeneous processors. Normal distributions with the following β values have been generated: $\beta = \{0.1, 0.25, 0.5, 0.75, 1\}$. The obtained $\omega_{j,n}$ values are then appropriately scaled so that $\sum_{j=1}^{|V|} \sum_{n=1}^{|P|} \omega_{j,n}$ becomes equal to $|V| \times |P| \times \overline{\omega_{DAG}}$.

4. **Inter-task data transmission workload**: The overall workload with respect to a task graph has two major components: (i) execution time requirements of the task nodes on processors (discussed above), and (ii) inter-task message transmission time workload on the communication links. The ratio of the overhead related to message transmission and execution, which we refer to as *Communication-to-Computation Ratio* (CCR), is an impor-

tant characterizing parameter of a task graph and may significantly vary depending on the nature of the application a task graph represents. We have evaluated the performance of the proposed algorithms *MMSH* and *PRESTO* for different values of *CCR* ($= \{0.1, 0.5, 1, 2, 5\}$). Given *CCR*, the average communication related workload $\overline{c_{DAG}}$ is obtained as:

$$\overline{c_{DAG}} = CCR \times \overline{\omega_{DAG}}$$

The average inter-task message size (denoted as $\overline{data_{DAG}}$; in Bytes) for a task graph is determined as:

$$\overline{data_{DAG}} = \overline{c_{DAG}} \times \overline{B}$$

where $\overline{B}$ is the average communication bandwidth of the considered platform. Experiments have been conducted for two different values of $\overline{B}$ ($= \{5\ Mbps, 10\ Mbps\}$). As there are $|P| \times (|P| - 1)/2$ communication links on a platform of $|P|$ processors, $\overline{B}$ can be expressed as:

$$\overline{B} = \frac{1}{|P| \times (|P| - 1)/2} \sum_{m=1}^{|P|} \sum_{n=1}^{m-1} B_{m,n}$$

where $B_{m,n}$ is the actual bandwidth of the communication link between processors p_m and p_n. The values of $B_{m,n}$ are generated from a normal distribution having mean $\mu = \overline{B}$ and various values of standard deviation σ ($= 0.2 \times \overline{B}$). The obtained $B_{m,n}$ values are then appropriately scaled so that $\sum_{m=1}^{|P|} \sum_{n=1}^{m-1} B_{m,n}$ becomes equal to $|P| \times (|P| - 1)/2 \times \overline{B}$. Similarly, the size of the output message ($data_{i,j}$) for each edge (τ_i, τ_j) in the task graph is determined from a normal distribution having mean $\mu = \overline{data_{DAG}}$ and various values of standard deviation σ ($= 0.2 \times \overline{data_{DAG}}$). The obtained $data_{i,j}$ values are then appropriately scaled so that $\sum_{i=1}^{|V|} \sum_{j=1}^{|V|} data_{i,j}$ becomes equal to $|E| \times \overline{data_{DAG}}$.

5. **Penalties**: The penalty associated with each time unit of execution on a processor p_n is denoted as ρe_n, while the penalty corresponding to each unit of data transfer on a communication link $\langle p_m, p_n \rangle$ is denoted as $\rho c_{m,n}$. The values of ρe_n and $\rho c_{m,n}$ are generated from uniform distributions within the ranges $[0.1, 1]$ and $[0.1, 0.5]$, respectively.

6. **Deadline Extension Rate**: It may be noted that the *MMSH* algorithm proposed as part of this work, typically delivers the shortest *makespan* schedule, while completely ignoring penalties that may be incurred thereof. Therefore, while the schedule length delivered by *MMSH* may be small, the corresponding penalty suffered may be high. Also, the *PRESTO* algorithm ultimately boils down to the *MMSH* algorithm in the interest of generating a *valid* schedule, when the stipulated deadline is very stringent with respect to

the workload imparted by a task graph. Thus, when the stipulated deadline is lower than *MMSH's makespan*, *PRESTO* can rarely generate a *valid* schedule. *PRESTO* is expected to generate *valid* but high penalty schedules when the deadline is larger but close to *MMSH's makespan*. Further for a given task graph, *valid* schedules with progressively lower penalties should be produced by *PRESTO* as the deadline is relaxed more and more with respect to *MMSH's makespan*. We define *deadline extension rate* (∂) as the ratio of the stipulated deadline of a task graph *'G'* with respect to the *makespan* produced by *MMSH* for *'G'*. The performance of *PRESTO* with respect to penalty minimization has also been evaluated by determining the average normalized penalties for different values of *deadline extension rate*. The different values of ∂ used in our experiments are: $\partial = \{1, 1.2, 1.4, 1.6\}$.

Simulation Framework: The simulation framework is written in *C* and executed on a system having the following configuration: (i) Intel® Core™ i5-6500 CPU @ 3.20GHz ×4, (ii) Ubuntu 16.04 LTS OS (64 bit), and (iii) 8 GB Memory.

3.5.2 Performance Metrics

The performance of the proposed methodology has been evaluated using five different parameters:

1. **Normalized Makespan**: This metric is used to compare performance of the proposed *makespan* minimization algorithm *MMSH* against the existing state-of-art algorithms *HEFT* [85], *PEFT* [5], *PPTS* [24], *PSLS* [112] and *PALG* [3]. Given a DAG, *Normalized Makespan* (NM) is defined as:

$$NM = \frac{X_{ms}}{\sum_{\tau_j \in CP_{AVG}} \overline{\omega_j}} \tag{3.16}$$

   where X_{ms} represents the *makespan* achieved by an algorithm like, *HEFT*, *PEFT*, *PPTS*, *PSLS*, *PALG* or *MMSH*. Assuming execution time of each task to be the average value over all heterogeneous processors, the denominator represents the sum of the execution times of all tasks in the critical path (CP_{AVG}) of the task graph. It may be noted that lower the value of achieved NM, better is the performance of the scheduling algorithm.

2. **Number of Occurrences of Better Solutions**: This metric exhibits pair-wise tabular comparisons between the performances of two algorithms in terms of the per-

centages of test cases in which one algorithm performs better, equal or worse than the other.

3. **Iteration Count**: This metric counts the average number of iterations of the while loop (steps 5-7) in function *PRESTO* (refer Algorithm 2). Lower the value of this metric, better is the performance of *PRESTO* towards quickly converging to a solution.

4. **Computation Time**: This metric measures the average run time (in ms) incurred by *PRESTO* to produce schedules over data generated through a fixed set of parameter values.

5. **Normalized Penalty Ratio**: This metric attempts to evaluate and compare the efficiency of *PRESTO* towards penalty minimization, against the penalty ignorant *makespan* minimization strategy *MMSH*. For a given task graph, *Normalized Penalty Ratio* (NPR) is the ratio between the average penalty suffered by *MMSH* to that suffered by *PRESTO*. That is:

$$NPR = \frac{\rho_{mmsh}}{\rho_{presto}} \tag{3.17}$$

It is easy to observe that higher the value of NPR, more is the efficiency of *PRESTO* towards penalty minimization.

3.5.3 Performance Results

In this subsection, we present detailed results corresponding to five conducted experiments. In these experiments, each data point is obtained as the average of 250 different task graph data (except Experiment 1 which uses 156250 test cases), corresponding to a fixed set of parameters.

3.5.3.1 Experiment-1: Pair-wise *makespan* comparison of algorithms

Table 3.2 shows pair-wise performance comparisons among the following algorithms *MMSH*, *PALG*, *PSLS*, *PPTS*, *PEFT* and *HEFT*. Specifically, the result corresponding to the ($row\ i$, $column\ j$)th- entry in the table depicts the percentages of test cases for which the algorithm corresponding to the i^{th} row performs better, equal or worse than the algorithm in column j. A total of 156250 test cases using Gaussian Elimination task graphs have been considered for each pair of algorithms. For example in Gaussian Elimination, the $(1, 5)^{th}$- entry in Table 3.2

shows that *MMSH* performs better, equal and worse in 75.8%, 1.8% and 22.4% test cases respectively, compared to *HEFT*. It may be observed from the first row of the table that *MMSH* performs better than all the other algorithms for a majority of the considered test cases. *MMSH* also performs at par with the other algorithms for Epigenomics, with trends of the results being similar to that of Gaussian Elimination. Hence, results for Epigenomics have not been separately shown in the chapter.

Table 3.2: *Pair-wise makespan comparison of the scheduling algorithms*

		PALG	*PSLS*	*PPTS*	*PEFT*	*HEFT*
MMSH	better	95.6%	50.7%	61.0%	55.1%	75.8%
	equal	0.5%	4.5%	3.8%	11.2%	1.8%
	worse	3.9%	44.8%	35.2%	33.7%	22.4%
HEFT	better	87.1%	26.7%	32.8%	26.4%	
	equal	3.2%	2.4%	1.2%	2.6%	
	worse	9.7%	70.9%	66.0%	71.0%	
PEFT	better	94.6%	37.0%	55.7%		
	equal	0.7%	12.8%	3.7%		
	worse	4.7%	50.2%	40.6%		
PPTS	better	92.9%	40.5%			
	equal	0.4%	2.2%			
	worse	6.7%	57.3%			
PSLS	better	95.6%				
	equal	0.6%				
	worse	3.8%				

3.5.3.2 Experiment-2: Normalized makespans vs. varying processors

This experiment measures the *Normalized Makespan* (NM) of *MMSH*, *PEFT*, *HEFT*, *PPTS*, *PSLS*, and *PALG*, for varying values of #processors ($|P|$). Obtained results for both Gaussian Elimination and Epigenomics are presented in Table 3.3. Here, the values of the parameters $|V|$, *CCR*, and β have been fixed at 104, 0.5 and 0.25, respectively. For both Gaussian Elimination and Epigenomics, *MMSH* seems to deliver slightly better results than the existing algorithms in almost all cases (44 out of 50 cases considered in Table 3.3).

Table 3.3: *Normalized Makespans for varying number of processors*

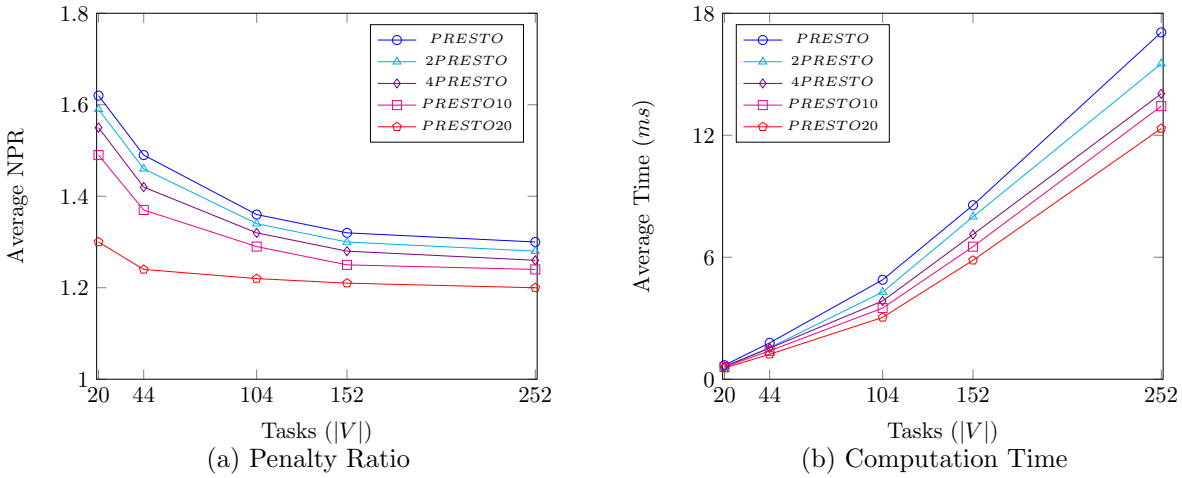
	Gaussian Elimination					Epigenomics				
	4	8	16	32	64	4	8	16	32	64
HEFT	1.16	0.95	0.90	0.86	0.81	2.97	1.65	1.06	0.91	0.84
PEFT	1.15	0.91	0.85	0.81	0.77	3.05	1.68	1.08	0.90	0.81
PPTS	1.15	0.92	0.87	0.83	0.79	2.93	1.63	1.07	0.91	0.83
PSLS	1.16	0.91	0.86	0.81	0.77	3.06	1.71	1.09	0.89	0.80
PALG	1.37	1.14	1.05	0.98	0.93	2.92	1.66	1.26	1.09	0.97
MMSH	1.14	0.90	0.84	0.80	0.75	2.99	1.67	1.06	0.89	0.80

3.5.3.3 Experiment-3: Normalized penalty ratios and run-times

One of the key facets of *PRESTO* is the interplay between the conflicting parameters *makespan* and *penalty*. Typically, lower a schedule's *makespan*, higher is the suffered penalty. Over the iterations of the *allocation phase* (refer to the *while loop* in steps 5-7 of Algorithm 2), *PRESTO* strives to converge to the largest *makespan* deadline meeting schedule, so that the lowest penalty *valid* schedule may be delivered. In this regard, the *deadline Throttling Factor (TF)* essentially acts as the tuning knob which determines the trade-off balance between *makespan* and *penalty*. Thus, the *rate of TF reduction* over iterations decides the pace at which solution *makespans* get reduced. This pace in turn plays a vital role towards controlling, (i) how quickly *PRESTO* produces a *valid* solution (run-time) and (ii) quality of the generated solution (penalty).

As revealed through equation 3.10, *PRESTO* has adopted a *deadline overshoot aware dynamic rate change policy* for the throttling factor *TF*. Additionally, the amount by which *TF* may be reduced in one step is lower bounded by a constant ϵ whose value is fixed at $\epsilon = 0.01$. In order to evaluate this policy, we have compared this with four other policies. The first two of these policies named, *PRESTO10* and *PRESTO20*, employ a *constant rate change strategy* which takes the form: $TF = TF - \epsilon'$ ($\epsilon' = 0.1$, for *PRESTO10* and $\epsilon' = 0.2$, for *PRESTO20*). The third and fourth policies named, *2PRESTO* and *4PRESTO*, employ dynamic rate change similar to the proposed work, but fixes the value of ϵ at 0.02 and 0.04, respectively.

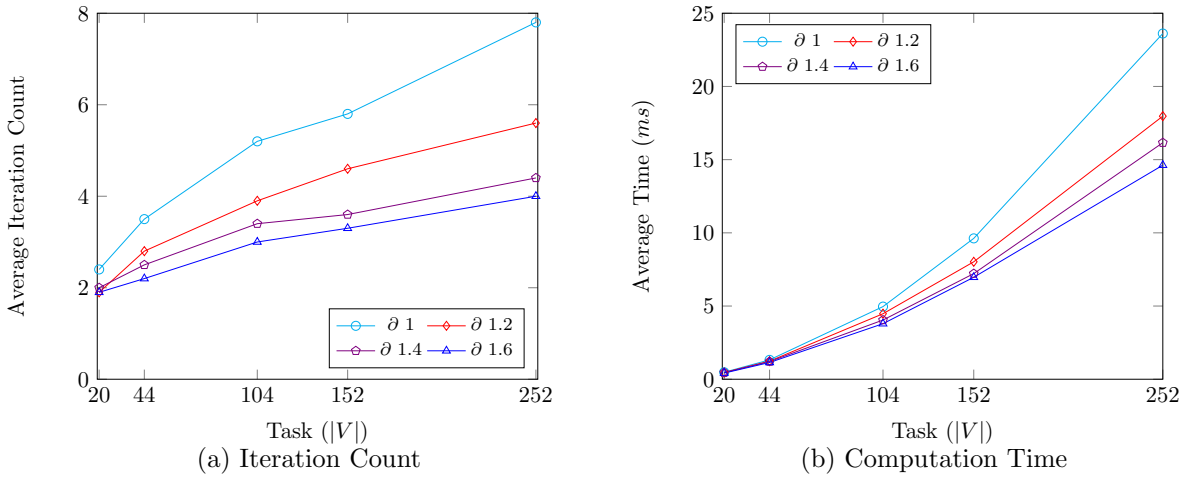
Fig. 3.5a and Fig. 3.5b depict the average *Penalty Ratio* (NPR) and average *run-time* respectively, as a function of the number of tasks ($|V|$). The parameters $|P|$, *CCR*, β and ∂ are set to 32, 0.5, 0.25 and 1.2. It may be observed from Fig. 3.5b that the different variations of *PRESTO* can be clearly ordered in terms of there average run-times from fastest to slowest as: *PRESTO20* < *PRESTO10* < *4PRESTO* < *2PRESTO* < *PRESTO*.



(a) Penalty Ratio (b) Computation Time

Figure 3.5: *Penalty Ratios and run-times of PRESTO for Gaussian Elimination.*

The first two strategies in the list are *constant rate change policies*, with the *TF reduction rate* of *PRESTO20* being double that of *PRESTO10*. A look at the *PRESTO* pseudocode (Algorithms 2 and 3) shows that *PRESTO20* and *PRESTO10* ensure convergence to a solution within 5 and 10 iterations, respectively. The other three approaches follow *dynamic rate change policies* in which the *rate of change of TF reduction* decreases as the amount of *makespan* overshoot beyond deadline becomes lower. The use of such a strategy allows *PRESTO* to make a higher number of attempts towards the generation of a *valid* solution as the deadline overshoot reduces, and thereby enhances the chances of obtaining better quality solutions. This fact may be observed from Fig. 3.5a where the *dynamic rate change policies* clearly outperform both *PRESTO20* and *PRESTO10*. However, this more intensive search for lower penalty solutions make the dynamic rate change policies slightly slower than the two constant rate change strategies, as revealed from Fig. 3.5b. Among the dynamic approaches, *PRESTO* has the finest resolution ($\epsilon = 0.01$) in the rate of change of *TF*, *4PRESTO* has the crudest resolution ($\epsilon = 0.04$), with *2PRESTO* being in the middle ($\epsilon = 0.02$). As is expected with this setting, *PRESTO* performs better than *2PRESTO* and *2PRESTO* performs better than *4PRESTO*, as may be seen from Fig. 3.5a. On the flip side, *PRESTO* typically incurs a higher number of iterations than *2PRESTO*, while *2PRESTO* incurs more iterations than *4PRESTO* for converging to a solution, and this describes the observation with respect to their run-times. Here, we have not shown the results for the Epigenomics application. This is because, the trends of the results as obtained for Epigenomics are similar to that of Gaussian Elimination.



(a) Iteration Count (b) Computation Time

Figure 3.6: *Iteration counts and run-times of PRESTO for Epigenomics.*

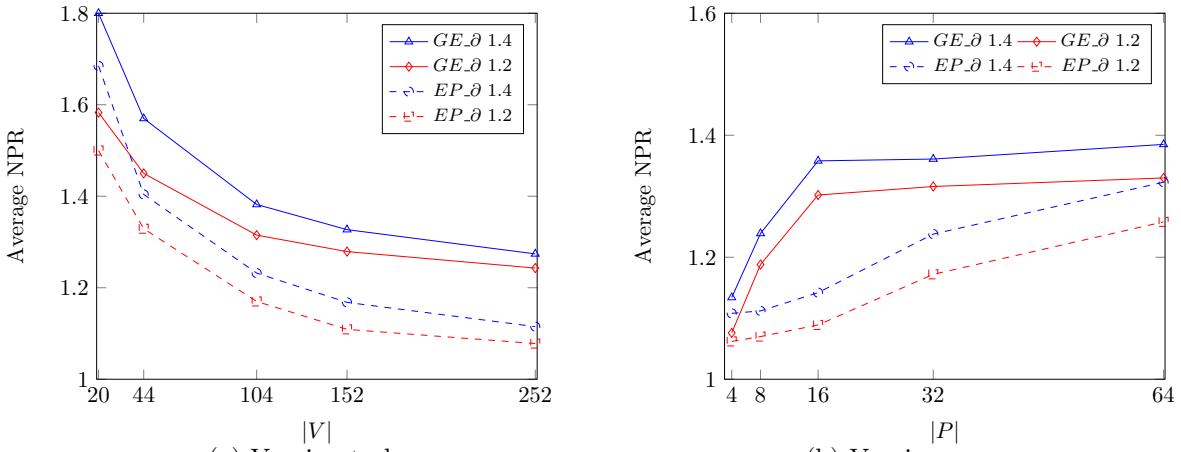
3.5.3.4 Experiment-4: Iteration counts and run-times

This experiment measures and compares the number of iterations (refer to the *while loop* in steps 5-7 of Algorithm 2) and execution times incurred by the *PRESTO* algorithm for different number of tasks ($|V|$). Values of the parameters $|P|$, *CCR* and β have been fixed at 32, 0.5 and 0.25, respectively. From Fig. 3.6a and Fig. 3.6b, it may be seen that both the average number of iterations and the average run-times of *PRESTO* decreases as deadline extension rate increases from 1 to 1.6. This is because, the deadlines get more and more relaxed for higher deadline extension rates, allowing higher slack time which *PRESTO* can use to quickly converge to a *valid* solution point. The figures show results for the Epigenomics application. The results for Gaussian Elimination have not been presented as they exhibit trends very similar to Epigenomics.

3.5.3.5 Experiment-5: Normalized penalty ratios w.r.t. tasks, processors, CCR and heterogeneity

This experiment depicts *Normalized Penalty Ratios* (NPR) which measure the ratio of the penalty suffered by *MMSH* to that suffered by *PRESTO*, for varying values of #tasks ($|V|$), #processors ($|P|$), Communication-to-Computation Ratios (*CCR*), and heterogeneity (β). Fig. 3.7 and Fig. 3.8 show the results obtained for both the Gaussian Elimination and Epigenomics applications. In each figure, solid and dashed lines represent the results of Gaussian Elimination and Epigenomics, respectively.

Fig. 3.7a depicts the obtained NPR values for different number of tasks $|V|$. The pa-



(a) Varying tasks (b) Varying processors

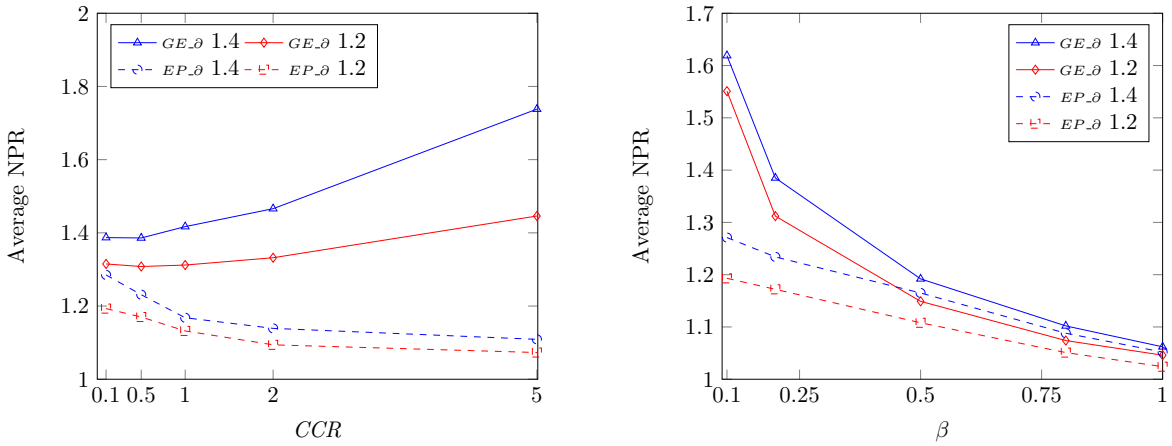
Figure 3.7: *Normalized Penalty Ratios w.r.t varying #tasks and #processors for Gaussian Elimination and Epigenomics.*

rameters $|P|$, CCR and β are set to 32, 0.5 and 0.25, respectively. It is observed that for any given number of tasks, average NPR increases as deadline extension rate (∂) increases, indicating that *PRESTO* is able to efficiently harness higher available slack to achieve better penalty reductions. As is obvious, NPR values decrease as workload becomes higher with increase in the number of tasks.

Fig. 3.7b shows NPR results as number of processors vary. In this, the values of the parameters $|V|$, CCR and β have been fixed at 104, 0.5 and 0.25, respectively. Similar to Fig. 3.7a, normalized penalty ratios seem to improve with higher deadline extension rate values for any fixed value of the number of processors. As a higher number of processors allows higher slack capacity, increase in the number of processors results in a consequent improvement in NPR.

Fig. 3.8a shows the effect of variation in CCR over obtained NPR values. The parameters $|V|$, $|P|$ and β are set to 104, 32 and 0.25, respectively. It may be seen that due to differences in structure, while NPR values show an approximately increasing trend with higher CCR for Gaussian Elimination, the trend is reverse for Epigenomics. This is because, with increase in CCR, a significantly higher number of parent-child node pairs get allocated to the same processor for Gaussian Elimination, compared to Epigenomics.

In Fig. 3.8b, we show the variation in NPR values as the degree of heterogeneity is increased from 0.1 to 1. Parameters $|P|$, CCR, and $|V|$ are fixed at 32, 0.5, and 104, respectively. As the average task-to-processor affinity reduces with increasing heterogeneity,



(a) Varying CCR (b) Varying heterogeneity

Figure 3.8: *Normalized Penalty Ratios w.r.t varying Communication-to-Computation Ratios and heterogeneity for Gaussian Elimination and Epigenomics.*

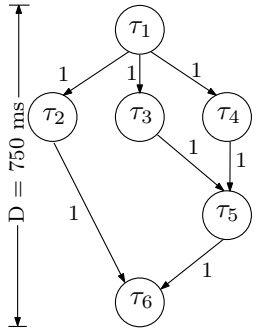
performance of *PRESTO* in terms of achieved NPR is observed to become poorer when heterogeneity becomes higher.

3.6 A Prototype Implementation

In addition to the above simulation-based experiments, a proof-of-concept implementation of the proposed work has been carried out on a real platform consisting of two processors, *ATmega*2560 (*Arduino Mega*) and *ATmega*328*p* (*Arduino Uno*) that are connected through their serial ports. We discussed this implementation by first presenting the task graph model of the application and the platform. Then, we discuss construction of the *PRESTO* schedule along with a step-by-step description of task execution on the platform.

Application Model: We have used a hypothetical application whose task graph model is shown in Fig. 3.9a. The task graph has six tasks and seven edges. The execution times of all tasks on *ATmega*328*p* and *ATmega*2560 have been determined by separately executing them in a standalone fashion on each processor. A list of these execution times may be found in Fig. 3.9b. The application is assumed to have an end-to-end deadline (D) of 750 ms. Each edge of the task graph is labeled with a positive weight '1' (in *bytes*) representing the size of the data to be transmitted between associated task pairs.

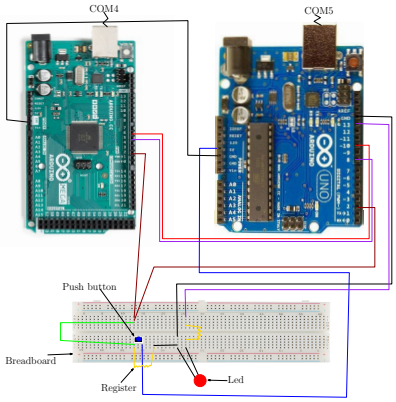
The Platform: The platform is created using two *Arduino* processors (*ATmega*2560 (*Arduino Mega*) and *ATmega*328*p* (*Arduino Uno*)) and establishing a serial communication channel (with 9600 *baud rate*) between them by connecting pins 10 (RX) and 8 (TX) of



(a) *A DAG* $G(V, E)$

Tasks	*ATmega328p* (Uno)	*ATmega2560* (Mega)
τ_1	332	335
τ_2	63	63
τ_3	108	109
τ_4	57	57
τ_5	181	183
τ_6	115	116

(b) *Matrix* W $(|V| \times |E|)$



(c) *Platform Layout*

Figure 3.9: *(a) A DAG* $G(V, E)$*; (b) Execution time (in ms) of each task on two processors; (c) The connection layout of real platform using ATmega2560 (Arduino Mega) and ATmega328p (Arduino Uno) processor.*

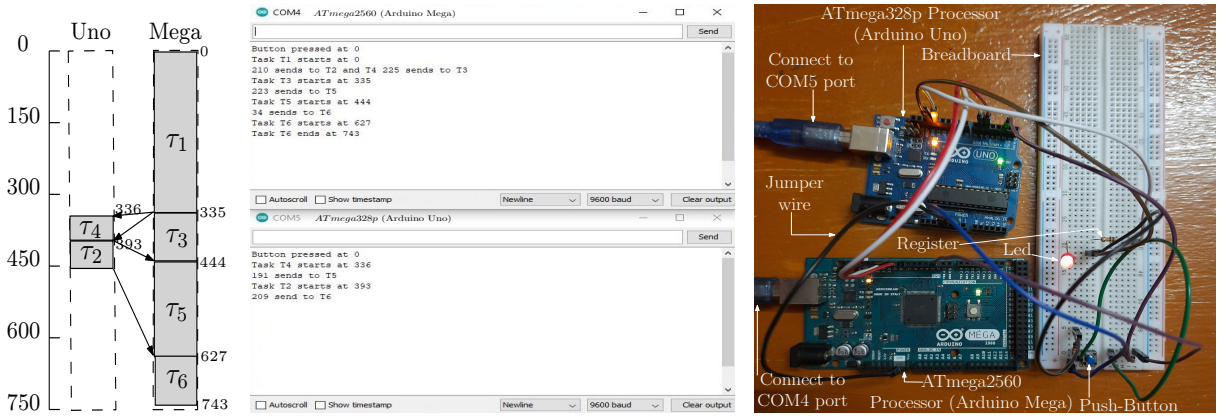
Arduino Mega to pins 6 (TX) and 5 (RX) of *Arduino Uno*, respectively. A few additional pieces of equipment such as: a *Push-button*, 2 *Resistors* (220 Ohm), a *Breadboard*, 10 *Jumper wires* and an *LED* has also been used. Fig. 3.9c shows the interconnection layout for the designed heterogeneous platform. The commencement of task executions on both processors have been synchronized by attaching a single *Push-button* which is connected to digital pin 2 of both *Arduinos*. We have assumed the penalty per unit processing time spent on the processors *ATmega328p*, and *ATmega2560* to be 0.85 and 0.80, respectively. The penalty for each unit of data transfer through the communication links is taken to be 0.5.

Table 3.4: *OFT, Rank and MEP values corresponding to the example system depicted in Fig. 3.9*

Tasks	OFT		Rank	MEP	
	ATmega328p	*ATmega2560*		*ATmega328p*	*ATmega2560*
τ_1	736	740	738.0	425.90	411.20
τ_2	178	179	178.5	146.85	143.20
τ_3	404	406	405.0	331.50	326.40
τ_4	353	354	353.5	288.15	284.80
τ_5	296	299	297.5	247.15	239.20
τ_6	115	116	115.5	97.75	92.80

Schedule Generation: Table 3.4 displays the *OFT*, *MEP* and *Rank* values corresponding to the *initialization phase* of *PRESTO* for the considered example system. The priority list of tasks is obtained as: $taskList = \langle \tau_1, \tau_3, \tau_4, \tau_5, \tau_2, \tau_6 \rangle$. The *allocation phase* for this example continues for *one* iteration and generates a *valid* schedule with *makespan*

743. The associated penalty is obtained as 698.40. Fig. 3.10a depicts the Gantt chart of the obtained schedule. The generated schedule indicates the processor allocation and actual execution start time of each task. For example, it may be seen that τ_3 starts on $ATmega2560$ (*Arduino Mega*) processor at time 335 *ms* relative to the start of the schedule.



(a) *PRESTO* **(b)** *Snapshorts of Schedule* **(c)** *Photograph of the Developed Setup*

Figure 3.10: *(a) Gantt chart of PRESTO: makespan* = 743 *ms, penalty* = 698.40*; (b) Snapshots of the monitor displays representing the execution of tasks on processors ATmega*2560 *and ATmega*328*p, for the DAG shown in Fig. 3.9a; (c) Real platform using ATmega*2560 *(Arduino Mega) and ATmega*328*p (Arduino Uno) processors.*

Implementation Procedure: To setup the execution environment, two separate code pieces (which include the task schedulers) have been written for the $ATmega328p$ (*Arduino Uno*) and $ATmega2560$ (*Arduino Mega*) processors. The structure of the code for $ATmega328p$ is displayed in Listing 3.1.

Lines 1 and 2 of Listing 3.1 include the *SchedTask* and *SoftwareSerial* libraries to allow multitasking in the system and serial communication between the two processors, respectively. Lines 3-5 declares the functions $\tau_4()$, $\tau_2()$ and $scheduler()$. Lines 6-8 initializes objects of *SchedTask* and *SoftwareSerial* classes.

The *setup()* function in Lines 9-15 runs only once when the *Arduino* board is switch *on*, followed by the *loop()* which contains instructions that get repeated until the board is turned *off*. The *setup()* function first initializes hardware pin 2 for *Push-button* (Line 11), the COM port for monitor display (Line 12), and the serial communication channel (Line 13) of *Arduino Uno*. After this, *setup()* finally calls the *scheduler()* function in Line 14. The value 9600 in line numbers 12 and 13 indicates bandwidth (*baud rate*) of the communication channels.

```
1  #include<SchedTask.h>
2  #include<SoftwareSerial.h>
3  void τ4();
4  void τ2();
5  void scheduler();
6  SchedTask Task4(-1, 0, τ4);
7  SchedTask Task2(-1, 0, τ2);
8  SoftwareSerial s(5,6);        // RX,TX for Uno
9  void setup()
10 {
11     pinMode(2, INPUT);        // Initialize hardware pin 2 for Push-button
12     Serial.begin(9600);       // Initialize COM port for monitor display
13     s.begin(9600);            // Initialize serial communication channel
14     scheduler();              // Call task scheduler
15 }
16 void loop()
17 {
18     SchedBase::dispatcher(); // Dispatch ready tasks for execution
19 }
20 void scheduler()
21 {
22     while(digitalRead(2)==LOW){}  // Wait until Push-button is pressed
23     Task4.setNext(336);       // Set Task 4 to start after 336 ms from
                                    the current time
24     Task2.setNext(393);       // Set Task 2 to start after 393 ms from
                                    the current time
25 }
```

Listing 3.1: *The code Structure of ATmega328p*

The *scheduler*() function is displayed in Lines 20-25 of Listing 3.1. The scheduler first waits in a loop until the *Push-button* is pressed (Line 22). Then it sets tasks τ_4 and τ_2 to enable the commencement of their executions after 336 ms and 393 ms (respectively), from the time at which the *Push-button* is pressed (refer Lines 23 and 24).

The *loop*() function in Lines 16-19 (of Listing 3.1) goes on executing the *dispatcher*() function repetitively. The *dispatcher*() correctly dispatches τ_4 and τ_2 at times 336 and 393 (relative to *Push-button* press), as may be observed from the screenshot of the monitor display in Fig. 3.10b. It may be noted that both tasks τ_4 and τ_2 commence by reading the serial communication channel ($s.read()$) in order to receive the output data transmitted at the completion of execution of task τ_1 on the $ATmega2560$ processor.

A similar code piece as displayed in Listing 3.2 is written for the $ATmega2560$ (*Arduino Mega*) processor. From the screenshot of the monitor display (Fig. 3.10b) corresponding to the execution of tasks on *Arduino Mega*, it may be observed that Listing 3.2 is also able to

currently govern the execution of tasks as prescribed by the *PRESTO* schedule in Fig. 3.10a. An annotated photograph of the actual setup built for this prototype implementation is shown in Fig. 3.10c.

```
#include<SchedTask.h>
#include<SoftwareSerial.h>
void τ1();
void τ3();
void τ5();
void τ6();
void scheduler();
SchedTask Task1(-1, 0, τ1);
SchedTask Task3(-1, 0, τ3);
SchedTask Task5(-1, 0, τ5);
SchedTask Task6(-1, 0, τ6);
SoftwareSerial s(10,8);       // RX,TX for Mega
void setup()
{
    pinMode(2, INPUT);        // Initialize hardware pin 2 for Push-button
    Serial.begin(9600);       // Initialize COM port for monitor display
    s.begin(9600);            // Initialize serial communication channel
    scheduler();              // Call task scheduler
}
void loop()
{
    SchedBase::dispatcher(); // Dispatch ready tasks for execution
}
void scheduler()
{
    while(digitalRead(2)==LOW){}  // Wait until Push-button is pressed
    Task1.setNext(0);         // Set Task 1 to start immediately
    Task3.setNext(335);       // Set Task 3 to start after 335 ms from
                                 the current time
    Task5.setNext(444);       // Set Task 5 to start after 444 ms from
                                 the current time
    Task6.setNext(627);       // Set Task 6 to start after 627 ms from
                                 the current time
}
```

Listing 3.2: *The code Structure of ATmega*2560

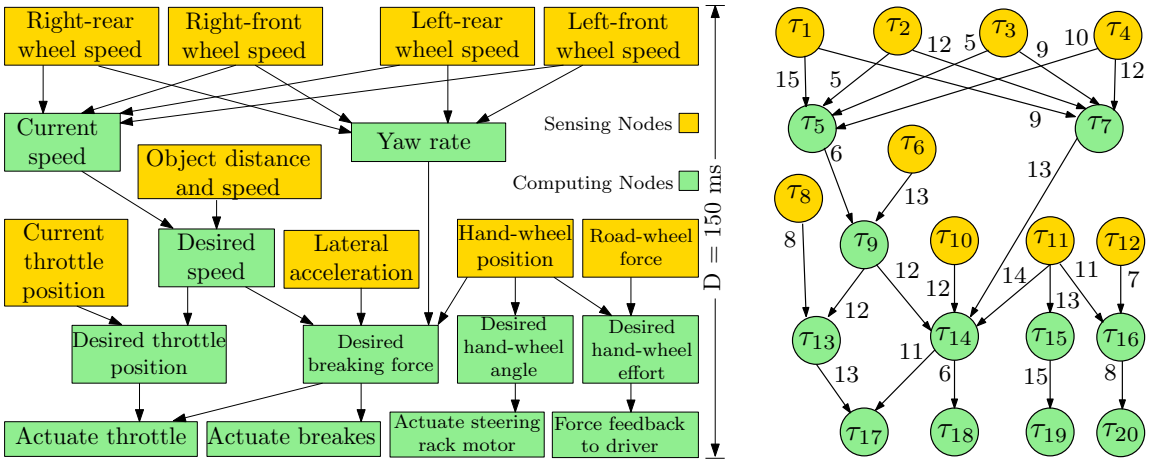
3.7 Case Studies

To illustrate the generic applicability of our proposed scheduling strategy in real-world designs, we present two case studies using (i) an *Adaptive Cruise Controller* (ACC) [13] in automotive systems, (ii) *Intelligent Surveillance Application* (ISA) [29] in a fog computing

environment.

3.7.1 Adaptive Cruise Controller in Automotive Systems

Adaptive Cruise Controller (ACC) automatically maintains a safe distance between two cars [41]. Fig. 3.11a shows the block diagram of ACC adapted from [13] and Fig. 3.11b depicts its corresponding DAG representation. We assume that this DAG consisting of 20 task nodes $\{\tau_1, \tau_2, ..., \tau_{20}\}$ is to be scheduled on a distributed platform having two heterogeneous processors $\{p_1, p_2\}$. The end-to-end application deadline is assumed to be 150 ms.



(a) *ACC Block Diagram* **(b)** *ACC DAG*

Figure 3.11: *Adaptive Cruise Controller (ACC).*

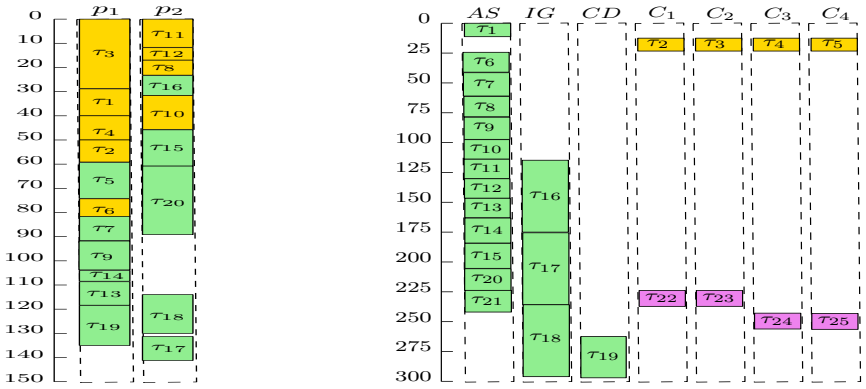
Table 3.5 depicts the computation times of the ACC application task nodes on the two heterogeneous processors. We assume a non-DVFS [99] platform with static power ratings associated with the two processors p_1 and p_2 to be: $s_{pow,1} = 0.01\ W$ and $s_{pow,2} = 0.02\ W$ respectively, while the corresponding dynamic power ratings are: $\rho e_1 = 0.62\ W$ and $\rho e_2 = 0.99\ W$, respectively. Similarly, the power rating of the link connecting the two processors in the platform is assumed to be: $\rho c_{1,2} = 0.5\ W$. The total energy consumption (E_{total}) associated with the execution of the application is given by the sum of both the static and dynamic energy dissipation [99]. That is:

$$E_{total} = \sum_{n=1}^{|P|} s_{pow,n} \times makespan + \rho_{presto}$$

where ρ_{presto} denotes the penalty for the generated execution schedule (refer equation 3.15) and gets directly mapped to the dynamic energy dissipation for the ACC application. Gantt chart representation of the *PRESTO* schedule is shown in Fig. 3.12a. It may be observed that *PRESTO* delivers a schedule which consumes an overall energy of 241.28 W while successfully completing within the deadline of 150 ms (*PRESTO* schedule length = 141 ms).

Table 3.5: *ACC: Execution time table*

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}
p_1	11	9	29	10	15	8	10	28	12	7	8	12	10	4	12	13	15	27	17	27
p_2	16	10	14	12	13	13	22	6	11	14	12	5	13	12	15	9	10	16	19	28



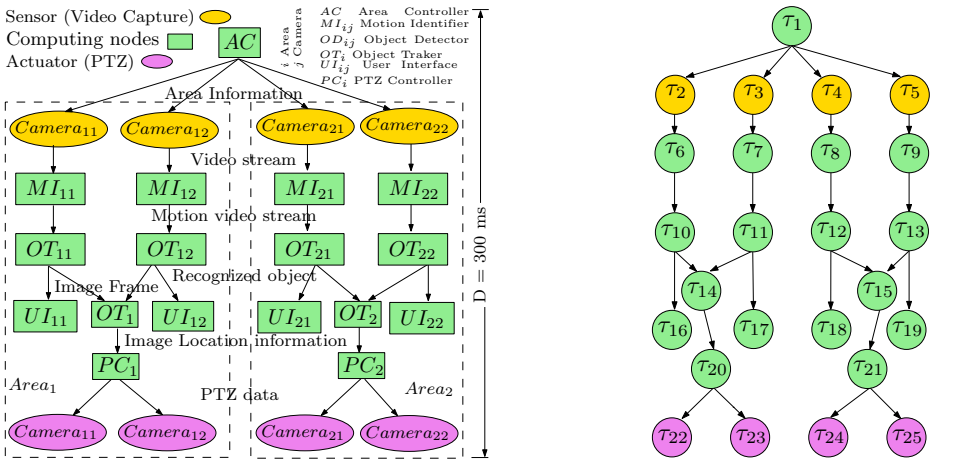
(a) *ACC* **(b)** *ISA*

Figure 3.12: *Gantt charts depicting the schedules of PRESTO on (a) ACC (makespan = 141 $ms, E_{total} = 241.28$ W), and (b) ISA (makespan = 297 ms, cost = \$145).*

3.7.2 Intelligent Surveillance in a Fog Environment

Intelligent Surveillance Application (ISA) aims at coordinating multiple cameras with different fields of view (FOVs) to surveil a vulnerable area. Coordination between cameras requires cooperative tuning of their *Pan-Tilt-Zoom* (PTZ) parameters, so that the best view of an area can be obtained. Moreover, the system sends alarms to the user in case of any unusual event which may demand the attention of security personnel for manual intervention. Here, we have considered two separate areas ($Area_1$ and $Area_2$), with each area being surveilled by two cameras. Fig. 3.13a shows the block diagram of ISA adapted from [29] and Fig. 3.13b depicts its corresponding DAG representation. This DAG consisting of 25

tasks $\{\tau_1, \tau_2, ..., \tau_{25}\}$ is to be scheduled on a Cloud-Fog environment having three processing nodes, in addition to the embedded processing elements within the four cameras. Among the processing nodes, one is a Cloud Data Center (CD), while the other two are Fog computing nodes (consisting of say, an Area Switch (AS) and an ISP Gateway (IG)). We assume that each computing device is associated with a certain geophysical location and is fully interconnected through an overlay network. The bandwidths of the bidirectional links between CD and the other devices is set to be 1 $Mbps$, bandwidths between IG and AS/cameras is 50 $Mbps$, while that between AS and cameras is 100 $Mbps$. The end-to-end application deadline is assumed to be 300 ms.



(a) *ISA Block Diagram* **(b)** *ISA DAG*

Figure 3.13: *Intelligent Surveillance Application (ISA).*

Table 3.6: *ISA: Execution time table*

	τ_1	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}
AS	10	18	18	18	18	17	17	17	17	20	20	67	67	67	67	19	19
IG	15	16	16	16	16	22	22	22	22	15	15	60	60	60	60	20	20
CD	8	11	11	11	11	14	14	14	14	13	13	34	34	34	34	15	15

Table 3.6 depicts the execution times of the ISA application tasks on three heterogeneous Cloud/Fog computing devices. The tasks τ_2-τ_5 and τ_{22}-τ_{25} in Fig. 3.13b are associated with video stream capture and PTZ actuation respectively, and are exclusively executed on the embedded processing elements in the cameras. The tasks associated with video stream capture and PTZ actuation are assumed to consume 12 ms and 11 ms of execution time,

respectively. The processing elements in the cameras do not play any role in the execution of other modules of ISA. All edges in the ISA block diagram (Fig. 3.13a) other than those from AC to cameras and PC to cameras are associated with the transmission of video stream data having size 12288 bytes. Data size corresponding to the rest of the edges (AC $\rightarrow$ cameras, PC $\rightarrow$ cameras) is considered to be 512 bytes. The monetary cost associated with each time unit of execution on the Area Switch, ISP Gateway and Cloud DC are considered as \$0.2, \$0.4 and \$0.8 respectively, while the cost of execution on a camera's processing element is assumed to be negligible. Similarly, the monetary cost corresponding to each unit of data transfer between any two devices available in the local network is set to \$0.001, while data transfer with the Cloud DC incurs \$0.002.

In this case study, we represent the *generic penalty function* (defined in equation 3.15) as the *total monetary cost* and the attempt is to minimize the overall cost incurred. Fig. 3.12b shows the Gantt chart representation of the generated *PRESTO* schedule. It may be observed that *PRESTO* delivers a schedule which incurs a total monetary cost of \$145 while successfully completing within the deadline of 300 ms (*PRESTO* schedule length = 297 ms).

3.8 Summary

In this work, we have presented a static list scheduling policy called *PRESTO*, for real-time task graphs to be executed on a system of fully-connected heterogeneous processors. The proposed scheme minimizes a *generic penalty function* while satisfying resource, precedence and timing constraints. Experimental analysis using two benchmark task graphs reveals that *PRESTO* performs appreciably over extensive sets of test scenarios, pointing to the practical effectiveness of the scheme. The practical applicability of *PRESTO* is presented using two case studies. In the first case, the objective is to minimize *energy consumption* of an adaptive cruise control application in an automotive system. In the second case, *PRESTO* generates a schedule for an intelligent surveillance application running in a fog environment, with the objective of minimizing associated *monetary cost*. A penalty ignorant *makespan* minimization algorithm called *MMSH* to which *PRESTO* converges in worst-case scenarios has also been designed. Experimental evaluation shows that *MMSH* is able to outperform the state-of-the-art *makespan* minimization strategies *HEFT*, *PEFT*, *PPTS*, *PSLS* and *PALG* in most cases. Finally, the practical adaptability of the proposed work is shown using a prototype real-platform implementation using two heterogeneous processors,

an *Arduino Uno* ($ATmega328p$) and an *Arduino Mega* ($ATmega2560$).

While designing *MMSH*, we realized that there is ample scope for improving *MMSH* as well as other state-of-the-art makespan minimizing DAG scheduling strategies mentioned above, by systematically applying the principles of any time heuristic search approaches. Based on this insight, Chapter 4 presents a low-overhead *makespan* minimizing depth-first branch and bound based search algorithm called *PRESTO*. *PRESTO* is equipped with a set of novel tunable pruning mechanisms, which allows the designer to obtain a judicious balance between performance (*makespan*) and solution generation times.

Chapter 4

HMDS: A Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems

4.1 Introduction

In the previous chapter, we used a greedy makespan minimizing resource allocation scheme called *MMSH* at the core of the proposed real-time scheduling strategy called *PRESTO*. In this chapter, we extend *MMSH* to obtain an anytime branch and bound strategy called *HMDS* which is able to comprehensively outperform *MMSH*.

The problem of scheduling DAGs to minimize *makespan*, is a challenging and computationally hard problem. Computation of *makespan* minimizing optimal schedules for DAGs on heterogeneous distributed computing systems requires exhaustive enumeration of an exponential state-space and is often prohibitively expensive even for moderately large problem sizes. Therefore, research in this domain has often focused on designing low-complexity heuristics that produce quick and satisfactory schedules [5, 85]. Heuristic-based algorithms provide approximate solutions, usually good solutions, with polynomial time complexity. In this work, we develop a list-based heuristic scheduling algorithm called *Heterogeneous Makespan-minimizing DAG Scheduler (HMDS)* for task graphs, whose objective is to minimize the overall *makespan* on a given set of heterogeneous processors. The *HMDS* algorithm is empowered with a guided state-space search technique [89]. The guided state-space search technique is typically an extension of the well-known branch and bound search approach [53]

in which only some nodes of the solution space are explored. In this search approach, at any level, only the promising nodes which are likely to deliver good solutions are retained for the branching while the remaining nodes are pruned. Through such aggressive pruning, exploration of a significant portion of the search tree can be avoided, which often helps to control solution generation times below acceptable limits.

The main contributions of this work are summarized as follows:

1. We first propose a list-based greedy static scheduling algorithm called *Heterogeneous Makespan-minimizing DAG Scheduler-Baseline (HMDS-Bl)* for the problem at hand. Experimental results show that *HMDS-Bl* is able to perform marginally better than state-of-the-art algorithms such as *HEFT*, *PEFT*, *PPTS*, etc.

2. Subsequently, *HMDS-Bl* is extended by employing a low-overhead depth-first branch and bound based search technique which is able to further deliver significant improvements in performance over *HMDS-Bl*. This enhanced scheduling technique is named *Heterogeneous Makespan-minimizing DAG Scheduler (HMDS)*.

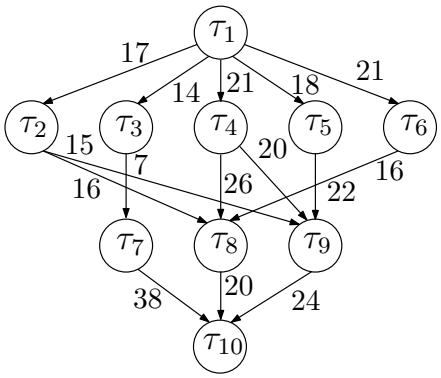
3. Experiments have been conducted using two real-world benchmark task graphs, namely Gaussian Elimination and Epigenomics. Results reveal that the proposed algorithms work equally efficiently over diverse variations in input parameters, such as number of tasks, different number of processors, Communication to Computation Ratio (CCR), and degree of heterogeneity. *HMDS* has been experimentally shown to comprehensively outperform the currently known state-of-the-art algorithms.

4. A real-world case study on traction control application has been presented to exhibit the practical applicability of the proposed work.

The rest of the chapter is organized as follows: Section 4.2 provides the system model and problem statement. In Section 4.3, we present the proposed schedulers and describe both scheduling policies, namely *HMDS-Bl* and *HMDS*. Experimental results and analysis have been provided in Section 4.4. Section 4.5 presents a real-world case study using an automotive traction control application. Finally, Section 4.6 concludes the chapter.

4.2 System Model

The system under consideration consists of an application modeled as a Directed Acyclic Graph (DAG), to be scheduled on a platform consisting of a set of heterogeneous processors. Fig. 4.1a depicts an example DAG, $G(V, E)$, where the set of vertices $V = \{\tau_1, \tau_2, \ldots, \tau_{|V|}\}$ represents tasks and the set of edges E represents precedence constraints between task pairs. Edge $e_{i,j}$ is assigned with a positive weight $data_{i,j}$ which represents the amount of data to be transferred from task τ_i to task τ_j. The set of all direct predecessor and successor tasks of a task say, τ_j are represented as $pred(\tau_j)$ and $succ(\tau_j)$, respectively.



(a) A DAG G(V,E); $|V| = 10, |E| = 15$

τ_j	p_1	p_2	p_3
τ_1	16	21	33
τ_2	13	25	43
τ_3	64	55	51
τ_4	30	18	29
τ_5	54	35	93
τ_6	48	50	42
τ_7	18	32	9
τ_8	51	27	40
τ_9	35	55	60
τ_{10}	63	81	44

(b) Matrix W ($|V| \times |P|$)

τ_j	PFT			$rank_{pft}$
	p_1	p_2	p_3	
τ_1	145	142	148	145.00
τ_2	91.7	106	98.5	98.72
τ_3	56.5	55.3	53.0	54.94
τ_4	97	107	101	101.67
τ_5	91	107	102	100
τ_6	92	77.7	83	84.22
τ_7	63	56.7	44	54.56
τ_8	54	50.7	44	49.56
τ_9	56	52	44	50.67
τ_{10}	0	0	0	0

(c) Matrix PFT and $rank_{pft}$

Figure 4.1: *An Example: (a) DAG of* 10 *tasks; (b) WCETs of* 10 *tasks on* 3 *processors; (c) PFT and $rank_{pft}$ values corresponding to the example system depicted in this Figure.*

The platform $P = \{p_1, p_2, \ldots, p_{|P|}\}$ consists of $|P|$ heterogeneous processors. These processors are fully interconnected through a set of $(|P| \times (|P| - 1)/2)$ bidirectional communication links having heterogeneous (potentially distinct) bandwidths. A matrix B of size $|P| \times |P|$ is used to store the data transfer rates between all pairs of processors. An element $b_{m,n} \in B$ denotes the bandwidth between processors p_m and p_n. Given $data_{i,j}$ and $b_{m,n}$, the data communication/transmission cost ($c_{i,j}^{m,n}$) between task pairs τ_i and τ_j ($\tau_j \in succ(\tau_i)$), when τ_i and τ_j are executed on distinct processors p_m and p_n respectively, may be determined as: $c_{i,j}^{m,n} = L_m + data_{i,j}/b_{m,n}$, where L_m is the communication startup cost of processor p_m. The L_m values of the different processors are stored in a vector L of size $|P|$. When τ_i and τ_j are mapped to the same processor, the communication overhead is assumed to be negligible. Thus, when $m = n$, $c_{i,j}^{m,n} = 0$. As processors are heterogeneous, each task may have distinct Worst-Case Execution Times ($WCETs$) on the different processors. W is a

$|V| \times |P|$ computation cost matrix (Fig. 4.1b) in which an element $\omega_{j,n}$ is the $WCET$ of task τ_j on processor p_n.

Problem Statement: *Given an application modeled as DAG G and a fully connected platform of heterogeneous processors P, the scheduling objective is to determine the execution start times and processor assignments for all tasks such that the schedule length/makespan is minimized while satisfying resource and precedence constraints.*

4.3 The Proposed Schedulers

In this section, we first describe the proposed list-based heuristic scheduling algorithm called *Heterogeneous Makespan-minimizing DAG Scheduler-Baseline (HMDS-Bl).* Subsequently in section 4.2, we present *Heterogeneous Makespan-minimizing DAG Scheduler (HMDS)*, obtained by extending *HMDS-Bl* with a novel adaptive guided search mechanism.

4.3.1 HMDS-Bl: The Baseline List Scheduler

HMDS-Bl is a list-based heuristic *makespan* minimizing scheduler having two phases: (i) *Task Prioritization*: for listing tasks in a specific *priority order* (this priority order ensures, (a) satisfaction of all *precedence constraints*, (b) construction of a schedule having low *makespan*), and (ii) *Processor Selection*: for mapping the highest priority unscheduled task to a processor that *minimizes* the sink task node's finish time.

The algorithm is critically pivoted on a function called $PFT()$ which is used to construct a matrix called *Predicted Finish Time* (PFT), containing values corresponding to each task-processor pair. This PFT matrix has two important functions: (1) Determination of a rank value for each task based on which a sorted task list as mentioned above, 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 schedule *makespan.* We now discuss the design of the $PFT()$ in more detail.

PFT$[\tau_j, p_n]$: Predicted Finish Time for different task-processor pairs are represented as a matrix where rows indicate tasks and columns indicate processors, as shown in Fig. 4.1c, for the example system presented in Figs. 4.1a and 4.1b. $PFT[\tau_j, p_n]$ essentially provides an estimate of the total cost required to complete the execution of all dependant nodes of

task τ_j, assuming τ_j to be allocated for execution on processor p_n. Formally, $PFT[\tau_j, p_n]$ is calculated as follows:

$$PFT[\tau_j, p_n] = \begin{cases} 0, & \text{if } \tau_j = \tau_{exit} \\ \max\limits_{\tau_k \in succ(\tau_j)} \Big[\min\limits_{p_r \in P} \{PFT[\tau_k, p_r] + \omega_{k,r} + c_{j,k}^{n,r}\} \Big], & \text{otherwise} \end{cases} \tag{4.1}$$

where the communication cost $c_{j,k}^{n,r} = 0$ when τ_j and its successor τ_k are scheduled on the same processor (i.e., $n = r$). It may be observed from equation 4.1 that $PFT[\tau_{exit}, p_n] = 0$ corresponding to the exit task. For all other tasks, $PFT[\tau_j, p_n]$ may be considered to be computed as a three-step process. In the first step, given a successor task τ_k (of τ_j) to be assigned to a certain processor p_r (say), an estimate of the total processing cost from the completion of τ_j on p_n to the completion of the exit task τ_{exit} is determined. This is a recursive step and the value is obtained as the summation of (i) the *predicted finish time* value corresponding to the execution of τ_k on p_r ($PFT[\tau_k, p_r]$), (ii) the execution cost of τ_k on processor p_r ($\omega_{k,r}$), and (iii) the actual communication cost required to transmit the output of τ_j to τ_k ($c_{j,k}^{n,r}$). For any given successor task τ_k (of τ_j), the second step calculates the minimum estimated processing time considering τ_k to be executed on all the different available processors in the system ($\min_{p_r \in P}\{\dots\}$). Finally at the third step, $PFT[\tau_j, p_n]$ is obtained as the maximum over the minimum estimated processing costs considering all successor tasks of τ_j ($\max_{\tau_k \in succ(\tau_j)}[\dots]$).

In the *task prioritization phase*, the average PFT for a given task τ_j over all processors is considered to determine the rank ($rank_{pft}[\tau_j]$) of τ_j. That is:

$$rank_{pft}[\tau_j] = \frac{\sum_{n=1}^{|P|} PFT[\tau_j, p_n]}{|P|} \tag{4.2}$$

The task priority order for considering them during processor selection is derived by generating a sorted vector called *taskList* arranged in non-increasing order of ranks. The priority order of tasks is responsible for maintaining the precedence constraints in the task graph and determining the task's execution order. However, due to the nature of the structural relationship between $rank_{pft}()$ and $PFT()$, there may be situations when *the rank of a task becomes smaller than the maximum among the ranks of all its successors*. The ready list of tasks when constructed in the order of $rank_{pft}$ values, will violate the stipulated precedence order among tasks. This case can be rectified with the following adaptation (refer Algorithm 4): *whenever the rank of a task τ_j is less than the maximum rank of its successors*

($MSR = \max_{\tau_k \in succ(\tau_j)} rank_{pft}[\tau_k]$), *$rank_{pft}[\tau_j]$ is minimally upscaled to: $MSR + \delta$, where δ is a small constant.* In our experiments, the value of δ has been considered as 0.1. The PFT values of τ_j on different processors are also proportionately upscaled.

Algorithm 4: $pft\_rank(G, P)$

Input: Task graph $G(V, E)$ and processor set P
Output: *Predicted Finish Time*, *rank* of all tasks

```
for τ_j = τ_exit to τ_entry do
    for each processor p_n in P do
        Calculate PFT[τ_j, p_n] using equation 4.1;
    Calculate rank_pft[τ_j] using equation 4.2;
    MSR = max_{τ_k ∈ succ(τ_j)} rank_pft[τ_k];
    if rank_pft[τ_j] ⩽ MSR then
        for each processor p_n in P do
            PFT[τ_j, p_n] = PFT[τ_j, p_n] × (MSR+δ) / rank_pft[τ_j];
        Recalculate rank_pft[τ_j] using equation 4.2;
return [PFT, rank_pft];
```

The objective of the *processor selection phase* is to generate a static schedule obtained by sequentially determining: (i) a *processor allocation*, and (ii) an *actual start time* for each task τ_j in the order prescribed by *taskList* such that the *schedule length* is minimized. The task (say, τ_j) in the *taskList* to be scheduled next, can be potentially allocated to any processor p_n on the heterogeneous platform. Each such possible task-processor allocation (say, $\langle \tau_j, p_n \rangle$) is associated with three attributes: (i) *Effective Start Time* (EST), (ii) *Effective Finish Time* (EFT), and (iii) *Optimistic Effective Finish Time* (O_{EFT}).

EST$[\tau_j, p_n]$: The effective execution start time of the entry task τ_{entry} on any processor p_n is *zero*. For other tasks, effective execution start time $EST[\tau_j, p_n]$ of task τ_j on processor p_n is defined as:

$$EST[\tau_j, p_n] = \begin{cases} 0, & if\ \tau_j = \tau_{entry} \\ \max\{avail[n], \max\limits_{\tau_i \in pred(\tau_j)} (AFT[\tau_i] + c_{i,j}^{m,n})\}, & otherwise \end{cases} \quad (4.3)$$

where $avail[n]$ is the earliest time at which processor p_n is available for task execution and $AFT[\tau_i]$ is the actual finish time of task τ_i. The inner max function in equation 4.3 returns the time at which all input data (required by τ_j) from τ_j's predecessors arrive at processor p_n.

EFT$[\tau_j, p_n]$: Given $EST[\tau_j, p_n]$, the effective execution finish time $EFT[\tau_j, p_n]$ of task τ_j on processor p_n is obtained as:

$$EFT[\tau_j, p_n] = EST[\tau_j, p_n] + \omega_{j,n} \tag{4.4}$$

O$_{\mathbf{EFT}}[\tau_j, p_n]$: This function provides an estimate of the sink task's completion time relative to the effective execution finish time of the current task τ_j on p_n. Formally, $O_{EFT}[\tau_j, p_n]$ is calculated as:

$$O_{EFT}[\tau_j, p_n] = EFT[\tau_j, p_n] + PFT[\tau_j, p_n] \tag{4.5}$$

Now, the task τ_j is actually allocated to a processor p_n for which $O_{EFT}[\tau_j, p_n]$ is minimal. After τ_j is scheduled on p_n, the EST and EFT of τ_j on p_n become the *Actual Start Time* $AST[\tau_j]$ and the *Actual Finish Time* $AFT[\tau_j]$ of task τ_j. The *makespan* of the schedule is equal to $AFT[\tau_{exit}]$ of the exit task τ_{exit}. The pseudocode of *HMDS-Bl* is presented in Algorithm 5.

Algorithm 5: HMDS-Bl(G, P)

Input: Task graph $G(V, E)$ and processor set P
Output: A schedule which minimizes *makespan*

```
1 [PFT, rank_pft] = pft_rank(G, P);
2 Sort tasks in non-increasing order of their rank_pft values and store in taskList;
3 for each task in taskList do
4     Extract the first task (say, τ_j) from taskList;
5     for each processor p_n in P do
6         Compute EFT[τ_j, p_n] and O_EFT[τ_j, p_n] of all task-processor pairs using equations
          4.4 and 4.5;
7     Determine: p_n | min_{p_r ∈ P} O_EFT[τ_j, p_r];
8     Assign τ_j on processor p_n;
```

Complexity Analysis: The computation of PFT considers each edge of the DAG exactly once and iterates over $|P|$ processors. Thus, PFT matrix creation overhead becomes $O(|P|(|V| + |E|))$ (refer equation 4.1; line 1 of *HMDS-Bl* (Algorithm 5)). Finding $rank_{pft}$ for one task requires $O(|P|)$ time. Hence, the complexity of computing $rank_{pft}$ for all tasks becomes $O(|V| \times |P|)$. In line 2, the sorting operation requires $O(|V| \log |V|)$ time. Therefore, the complexity of the *task prioritization phase* is $O(|P|(|V|+|E|)+|V| \times |P|+|V| \log |V|) = O(|P|(|V|+|E|))$. The overhead of the *processor selection phase* is primarily governed by the computational complexity of determining $EFT[\tau_j, p_n]$ and $O_{EFT}[\tau_j, p_n]$ (refer equations 4.4

and 4.5; in line 6) for all task-processor pairs within the nested *for* loops (outer loop: lines 3-8; inner loop: lines 5-6). Overhead of the remaining lines of *HMDS-Bl* take $O(1)$ time except line 7 which requires $O(|P|)$ time. The overhead of computing $EFT[\tau_j, p_n]$ is mainly dependent on the complexity of determining $EST[\tau_j, p_n]$ (refer equation 4.3). Determination of $EST[\tau_j, p_n]$ require $O(1)$ computations over all predecessors of task τ_j and hence has an overhead of $O(\#predecessors)$. The aggregate count of predecessors of all tasks is equal to the total number of edges in the DAG. Hence, the amortized overhead of finding $EST[\tau_j, p_n]$ becomes $O(|P|(|V|+|E|)/(|P|\times|V|)) = O((|V|+|E|)/|V|) = O(|E|/|V|)$. So, the overall complexity for computing $EST[\tau_j, p_n]$ on all task-processor pairs is $O(|E|/|V| \times |P| \times |V|) = O(|E| \times |P|)$. The overhead of determining $O_{EFT}[\tau_j, p_n]$ for all task-processor pairs is $O(|V| \times |P|)$. Hence, the complexity of *HMDS-Bl* including overheads for both *task prioritization* and *processor selection* can be expressed as: $O(|P|(|V|+|E|) + |P| + |E| \times |P| + |V| \times |P|) \approx O(|E| \times |P|)$.

Example: Fig. 4.1c lists the PFT and $rank_{pft}$ values corresponding to the example system shown in Figs. 4.1a and 4.1b. Here, we assume that the communication links between each pair of processors have heterogeneous bandwidths ($b_{1,2} = b_{2,1} = 1$, $b_{2,3} = b_{3,2} = 3$, $b_{3,1} = b_{1,3} = 2$, and $b_{1,1} = b_{2,2} = b_{3,3} = \infty$) and the communication startup costs are negligible for all processors (i.e., $L_m = 0$). The priority list of tasks is obtained as: $taskList = \langle \tau_1, \tau_4, \tau_5, \tau_2, \tau_6, \tau_3, \tau_7, \tau_9, \tau_8, \tau_{10} \rangle$. The Gantt chart in Fig. 4.2g shows the schedule generated by *HMDS-Bl*. It can be noted that from Fig. 4.2, *HMDS-Bl* (*makespan* 182) outperforms *HEFT* (*makespan* 198), *PEFT* (*makespan* 189), *PALG* (*makespan* 200), *PPTS* (*makespan* 187), *PSLS* (*makespan* 184) and *MMSH* (*makespan* 184).

Next, we discuss the Depth-First Branch and Bound (*DFBB*) search-based extension to *HMDS-Bl* called, *Heterogeneous Makespan-minimizing DAG Scheduler (HMDS).*

4.3.2 HMDS: DFBB Search Based Extension to HMDS-Bl

In *HMDS-Bl*, for a fixed allocation order dictated by $rank_{pft}$ values, each task has $|P|$ heterogeneous processor choices during allocation. Given this scenario, it may be observed that it is possible to derive up to $|P|^{|V|}$ distinct solutions for a task set of size $|V|$. It may be noted that *HMDS-Bl* solves the processor selection problem for each task τ_j by allocating that processor p_n for which $O_{EFT}[\tau_j, p_n]$ assumes the minimum value. However, this strategy may not always deliver the minimum *makespan* among the $|P|^{|V|}$ possible solutions, as

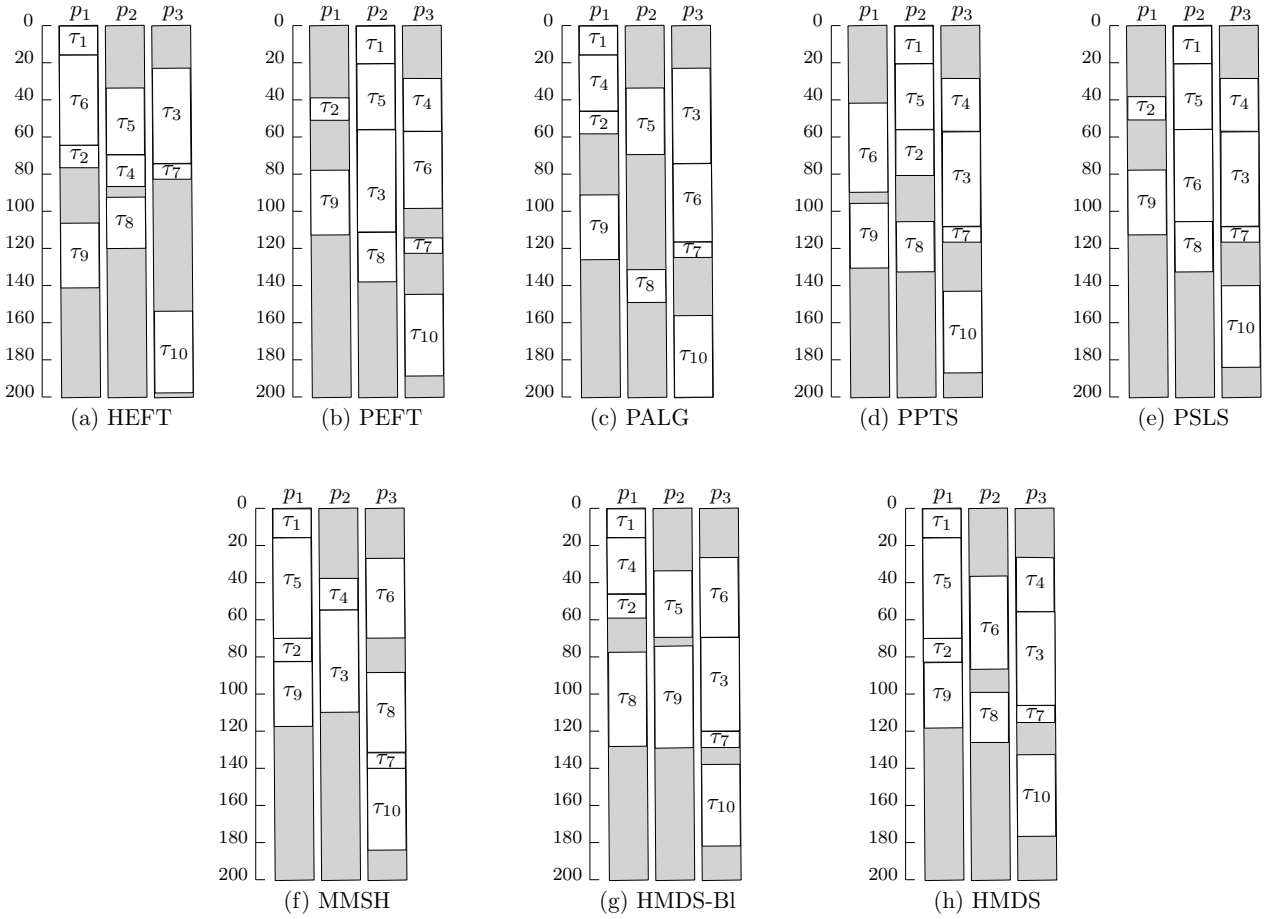


Figure 4.2: *Gantt charts depicting the schedules: (a) HEFT (makespan = 198); (b) PEFT (makespan = 189); (c) PALG (makespan = 200); (d) PPTS (makespan = 187); (e) PSLS (makespan = 184); (f) MMSH (makespan = 184); (g) HMDS-Bl (makespan = 182); (h) HMDS (makespan = 177), for the DAG in Fig. 4.1a.*

illustrated next by continuing with the same example discussed in Section 4.3.1. For this example, Fig. 4.2g shows the schedule Gantt chart obtained using *HMDS-Bl*. Using the same task allocation order suggested by $rank_{pft}$, Fig. 4.2h shows the outcome of a strategy where not all tasks have been assigned to the processors in which their O_{EFT} values are minimum. For the schedule in Fig. 4.2h, while tasks τ_1, τ_2, τ_3, τ_7, and τ_{10} are allocated to the processors where their O_{EFT} values are minimum, tasks τ_4, τ_5, τ_6, τ_8, and τ_9 have been assigned to processors where their O_{EFT} values are the second best (i.e., second minimum). The *HMDS-Bl* schedule in Fig. 4.2g delivers a *makespan* of 182, while the schedule depicted in Fig. 4.2h has a lower *makespan* of 177. We see that the modified task-processor assignment in Fig. 4.2h has allowed the possibility *to assign tasks* τ_8 *and* τ_9 *on processors where their execution times are significantly lower than on other processors*. This may be observed to be an important reason towards the achievement of a lower *makespan* in Fig. 4.2h. However, due to the inherent exponential solution space associated with the problem, we purview that it may be difficult to design a deterministic/greedy procedure which avoids actual solution enumeration for determining the choice of processor for each task, such that the delivered *makespan* may be significantly lower than *HMDS-Bl*. With this understanding, we have resorted to the design of an efficient DFBB search based scheduler called *HMDS*.

HMDS uses the same $PFT()$ function (equation 4.1) for task-processor pairs and the same task ranking scheme as *HMDS-Bl*. However, with *HMDS-Bl* as the underlying scheme, *HMDS* employs a low-overhead heuristic depth-first branch and bound search approach to provide significant performance gains, while incurring low and bounded additional solution generation times compared to *HMDS-Bl*. The pseudocode of *HMDS* is presented in Algorithm 6. It consists of two phases: (i) *initialization phase* and (ii) *allocation phase*. While lines 1 - 4 of function *HMDS()* comprises the *initialization phase*, line 5 and line 6 which calls function *assignTasks()* presented in Algorithm 7 and outputs the final schedule, comprises the *allocation phase*. Table 4.1 lists the important variables used in algorithm *HMDS* and describes their associated meanings.

Initialization Phase: Line 1 of Algorithm 6 records the current CPU clock time. Line 2 initializes the global variables $FSTime$ to 1, msl to infinity (∞), ops to 2, λ to 5 and $capT$ to 2^{10}. Line 3 computes two different parameters which are used in the subsequent *allocation phase* namely, (1) *Predicted Finish Time* $PFT[\tau_j, p_n]$ (refer equation 4.1). (2) *A rank value* $rank_{pft}[\tau_j]$. Given the PFT values of task τ_j on each processor, its average is considered as the rank ($rank_{pft}[\tau_j]$) of τ_j (refer equation 4.2). The $PFT[\ ]$ values for all task-processor

Table 4.1: *List of important variables used in HMDS and their meanings*

Variables	Definitions
$initT$	Stores the current CPU clock time.
msl	Records the best schedule length generated so far by *HMDS*.
ops	Bound on the number of processor choices (pruning mechanism-2).
λ	Allowable O_{EFT} degradation bound in *percentage* (pruning mechanism-3).
$capT$	Allowable run-time bound; set as a multiple of *HMDS-Bl*'s run-time (pruning mechanism-4).
$BAST$, $BAFT$, $BPRO$	Maintains enumerations of the best schedule at any given time using three arrays of size $\|V\|$. For all tasks in the schedule $BAST$ stores the start times, $BAFT$ holds finish times, and $BPRO$ stores the processor allocations.
p'	An array of size ops containing processor ids representing processor choice priorities for a task; sorted in non-decreasing order of O_{EFT}.
$taskListID$	Index of the task in $taskList$ being currently processed.
$FSTime$	Time required to generate the first solution in *HMDS*; this is approximately equal to the solution generation time of *HMDS-Bl*.

pairs and the $rank_{pft}[\,]$ values of all tasks are determined through the function $pft\_rank()$ (refer Algorithm 4). The tasks are stored in a vector $taskList$ in non-increasing order of their *ranks* (line 4).

Algorithm 6: HMDS(G, P)

Input: Task graph $G(V, E)$, processor set P
Global Variables: $initT$, $FSTime$, msl, $capT$, ops, λ, $BAST$, $BAFT$ and $BPRO$
Output: A schedule which minimizes *makespan*

1 $initT = clock()$;
2 $FSTime = 1$, $msl = \infty$, $ops = 2$, $\lambda = 5$, $capT = 2^{10}$;
3 $[PFT, rank_{pft}] = pft\_rank(G, P)$;
4 Sort tasks in non-increasing order of their $rank_{pft}$ values and store in $taskList$;
5 assignTasks(G, P, 1);
6 Print lists $BAST$, $BAFT$ and $BPRO$, representing the generated schedule;

Allocation Phase: The objective of the *allocation phase* is to generate a static schedule obtained by determining: (i) a *processor allocation* $BPRO[\tau_j]$, (ii) an *actual start time* $BAST[\tau_j]$, and (iii) an *actual finish time* $BAFT[\tau_j]$ for each task τ_j in the order prescribed by the priority vector $taskList$ such that the *makespan* is minimized. The task (say, τ_j) in $taskList$ to be scheduled next may be potentially allocated to any processor p_n on the heterogeneous platform. Each such possible task-processor allocation (say, $\langle \tau_j, p_n \rangle$) is associated with three attributes: (i) *Effective Start Time* (EST) (refer equation 4.3), (ii) *Effective*

Finish Time (EFT) (refer equation 4.4), and (iii) *Optimistic Effective Finish Time* (O_{EFT}) (refer equation 4.5). In order to allocate each task to a specific processor, we have designed an efficient DFBB search-based schedule generation scheme called $assignTasks()$.

Algorithm 7: assignTasks(G, P, $taskListID$)

Input: Task graph G, processor set P, $taskListID$
Global Variables: $initT$, $FSTime$, msl, $capT$, ops, λ, $BAST$, $BAFT$ and $BPRO$
Output: A schedule which minimizes *makespan*

```
1  if (clock() − initT)/FSTime > capT then
2      return;
3  if taskListID = |V| + 1 then
4      if msl = ∞ then
5          FSTime = clock() − initT; // Time to generate 1st sol.
6      if AFT[τ_exit] < msl then
7          msl = AFT[τ_exit];
8          Copy lists AST, AFT and PRO into BAST, BAFT and BPRO;
9      return;
10 τ_j = taskList[taskListID];
11 for each processor p_n in P do
12     Compute EST[τ_j, p_n], EFT[τ_j, p_n] and O_EFT[τ_j, p_n] for task τ_j using equations 4.3, 4.4
       and 4.5;
13 Let p' be an array of size ops such that O_EFT[τ_j, p'_1] ⩽ O_EFT[τ_j, p'_2] ⩽ ··· ⩽ O_EFT[τ_j, p'_ops];
14 limit = O_EFT[τ_j, p'_1] × (1 + λ/100);
15 for i = 1 to ops do
16     if O_EFT[τ_j, p'_i] ⩽ limit and O_EFT[τ_j, p'_i] < msl then
17         Assign τ_j to p'_i;
18         Copy EST[τ_j, p'_i], EFT[τ_j, p'_i] and p'_i to AST[τ_j], AFT[τ_j] and PRO[τ_j];
19         prevAvail = avail[p'_i];
20         avail[p'_i] = EFT[τ_j, p'_i];
21         assignTasks(G, P, taskListID + 1);
22         avail[p'_i] = prevAvail;
```

Pseudocode of this recursive $assignTasks()$ function is presented in Algorithm 7. Each call to $assignTasks()$ attempts to extend the currently generated partial schedule (stored within lists AST, AFT and PRO) by one more task (say, τ_j; line 10) in rank order, allocating it on a certain processor (say, p'_i; lines 10-22). In lines 11-12, the $EST[\tau_j, p_n]$, $EFT[\tau_j, p_n]$ and $O_{EFT}[\tau_j, p_n]$ values of task τ_j for each processor p_n are calculated using equations 4.3, 4.4 and 4.5, respectively. Next, a sorted list p' of processor ids is constructed such that

$O_{EFT}[\tau_j, p'_1] \leqslant O_{EFT}[\tau_j, p'_2] \leqslant \cdots \leqslant O_{EFT}[\tau_j, p'_{ops}]$ (line 13), where p'_i denotes the i^{th} element in the list p'; $i = 1, 2, \ldots, ops$. We first try to extend the schedule with τ_j allocated to p'_1, the processor on which τ_j's O_{EFT} is minimum. Subsequently, when the schedule backtracks, we try to extend it with τ_j on p'_2, the processor for which τ_j's O_{EFT} is second best, and this process continues likewise. After assigning τ_j to a certain processor p'_i (line 17), a recursive call to $assignTasks()$ is made in the attempt to further extend the schedule by one more task (line 21). $assignTasks()$ returns to its caller by generating a complete schedule when all tasks in DAG G have been explored (lines 3-9). After a complete schedule is generated, the currently known best schedule (stored in the lists $BAST$, $BAFT$ and $BPRO$) and its associated *makespan* msl are updated, if *makespan* of the current schedule is lower than msl (lines 6-8).

In order to keep solution generation times comparable to *HMDS-Bl* (to at most a small multiple of *HMDS-Bl*) while ensuring significantly better solutions, the search procedure has been equipped with the following novel pruning mechanisms.

4.3.2.1 Pruning Mechanism 1: Using a lower bound heuristic function

Pruning a partial schedule if its estimated lower bound on makespan is higher than the makespan of the best actual schedule obtained thus far: The condition, $O_{EFT}[\tau_j, p'_i] < msl$ in line 16 of the $assignTasks()$ function (refer Algorithm 7) implements this mechanism. Theorem 4.3.2 proves that $O_{EFT}[\tau_j, p_n]$ is a lower bound on the *makespan* achievable by any complete schedule which includes the current partial schedule along with the allocation of τ_j on p_n. With O_{EFT} being a lower bound, the search strategy gets empowered with the following salient property: *when the condition, $O_{EFT}[\tau_j, p'_i] < msl$ holds, there is no possibility of finding any complete schedule whose makespan is lower than the current best (msl), by extending further with the current partial schedule and allocating of τ_j on p_n.*

Lemma 4.3.1. *Given a fixed rank list of tasks, $PFT[\tau_j, p_n]$ is a lower bound on the time required to complete execution of the remaining unscheduled task nodes in the rank list, subsequent to the completion of task τ_j on processor p_n.*

Proof. (*by Induction*) We prove this property by induction, traversing through the rank list of tasks in reverse order from the exit task τ_{exit}, until the current task τ_j is reached.
Base case: When $\tau_j = \tau_{exit}$, $PFT[\tau_j, p_n]$ is set to 0 (equation 4.1), which is obviously a lower bound.

Induction hypothesis: When τ_j is any other task node ($\tau_j \neq \tau_{exit}$), we assume that for all immediate successors τ_k of τ_j, $PFT[\tau_k, p_r]$ is a lower bound, $\forall r \in [1, |P|]$.
Induction step: It may be appreciated that $c_{j,k}^{n,r}$ is the actual time required to transmit the output of τ_j (on p_n) to the input of τ_k (on p_r), and $\omega_{k,r}$ is the actual execution time of τ_k (on p_r). Hence when τ_k executes on p_r, the expression $PFT[\tau_k, p_r] + \omega_{k,r} + c_{j,k}^{n,r}$ represents a lower bound on the time required to complete the schedule subsequent to the generation of τ_j's output (on p_n). Consequently, the minimum value that is attained by this expression over all processor choices for τ_k, is also a lower bound. Finally, as the output of τ_j must be received by all its successors τ_k, $PFT[\tau_j, p_n] = \max_{\tau_k \in succ(\tau_j)} \left[\min_{p_r \in P} \{ PFT[\tau_k, p_r] + \omega_{k,r} + c_{j,k}^{n,r} \} \right]$ is a lower bound. □

Theorem 4.3.2. *Given a DAG G, $O_{EFT}[\tau_j, p_n]$ is a lower bound on the makespan achievable by any complete schedule which includes the current partial schedule along with the allocation of τ_j on p_n.*

Proof. From equation 4.5, it can be seen that $O_{EFT}[\tau_j, p_n]$ is obtained as a summation of two components, $EFT[\tau_j, p_n]$ and $PFT[\tau_j, p_n]$. Given the current partial schedule, the first component $EFT[\tau_j, p_n]$ denotes the actual finish time of τ_j on p_n. Lemma 4.3.1 proves that $PFT[\tau_j, p_n]$ is a lower bound on the time required to complete execution of the remaining unscheduled task nodes in the rank list, subsequent to the completion of τ_j on p_n. Hence, $O_{EFT}[\tau_j, p_n]$ is a lower bound. □

4.3.2.2 Pruning Mechanism 2: Cap on maximum #processor choices

Bounding the number of processor choices for each task to a small constant: A task τ_j is attempted to be allocated on different alternative processor choices with the choices being in non-decreasing order of τ_j's O_{EFT} values on those processors. As discussed above, schedules with lower *makespans* than that produced by *HMDS-Bl* may possibly be obtained for any of these processor choices. However, the likelihood of better solutions with later choices (beyond the first two processor choices, say) is often very low. This is because in most cases, the O_{EFT} values (which provide a lower bound on schedule length) for the later processor choices are significantly higher than the O_{EFT} value for the first choice. This observation is also validated by our experimental results. As an illustration let us analyse Fig. 4.4e (Experiment 4, Section 4.4.3), which shows the schedule length ratio values for the Gaussian Elimination task graph as the number of tasks vary from 20 to 55. It may

be seen that for task graphs containing 54 task nodes, average schedule lengths reduce by about 7.4% when the value of the number of processor choices (*ops*) is increased from 1 to 2. However, when *ops* is increased from 2 to 3, the average decrease in schedule lengths reduces to only 0.19%. Further, there is no decrease in schedule lengths when *ops* increases from 3 to 4. On the other hand, solution generation times increase exponentially as the value of *ops* becomes larger. For example, Fig. 4.4f shows that the average run-time values for *ops* equal to 1, 2, 3 and 4 are 0.08 ms, 13 *secs*, 110 *secs* and 250 *secs*, for the case when the DAGs contain 54 task nodes. Although the exact values may be different, this general trend has been found in all our experiments. Based on these observations, the number of alternative processor allocation choices *ops*, for any task τ_j has been limited to 2 (line 2 of Algorithm 6 and line 15 of Algorithm 7) in the *HMDS* algorithm.

4.3.2.3 Pruning Mechanism 3: O_{EFT} based cap on #procssor choices

Bounding the number of processor choices for each task based on the allowable increase in O_{EFT} values compared to the best choice: For any given task τ_j, the likelihood of finding lower *makespans* compared to *HMDS-Bl* reduces to a low value with a certain processor choice p'_i $(i > 1)$, if the value of $O_{EFT}[\tau_j, p'_i]$ is significantly higher than $O_{EFT}[\tau_j, p'_1]$. Our experimental results also validate this observation. For example, with the value of *ops* fixed to 2, Fig. 4.5b depicts the schedule length ratios and run-times when search corresponding to the second processor choice (p'_2) is only conducted if the values of $O_{EFT}[\tau_j, p'_2]$ is at most λ percent larger than $O_{EFT}[\tau_j, p'_1]$, for all tasks τ_j. It may be derived from the table that for 54 task node DAGs, compared to *HMDS-Bl*, average schedule lengths for *HMDS* improve by 3.9%, 6.1%, 6.9%, 7.2%, 7.3%, 7.39%, and 7.4% for λ equal to 1%, 3%, 5%, 10%, 15%, 20%, and 100%, respectively. At the same time, average run-times increase by about $2^{3.3}$, $2^{8.9}$, $2^{11.6}$, $2^{15.3}$, $2^{16.7}$, $2^{17.1}$, and $2^{17.3}$ times for λ equal to 1%, 3%, 5%, 10%, 15%, 20%, and 100%, with respect to *HMDS-Bl*. Based on this observation, in our experiments, we have assumed the value of λ to be 5% (line 2 of Algorithm 6 and line 14 of Algorithm 7) as a judicious trade-off between solution quality and corresponding solution generation times. It may be noted that for $\lambda = 5\%$, *HMDS* is able to achieve 93% of the maximum possible reduction (for the case with $\lambda = 100\%$) in schedule lengths with *ops* being fixed to 2. In addition, the average run-time for this case is also acceptable being around 0.25 *sec*.

4.3.2.4 Pruning Mechanism 4: Cap on solution generation times

Terminating the search based on the amount of time consumed with respect to that taken by HMDS-Bl: *HMDS* follows a depth-first branch and bound strategy where the first valid solution is generated when all task nodes have been explored once. It may be noted that for this case, all tasks are allocated to the processors where the O_{EFT} values are minimum, and so this solution is same as that produced by *HMDS-Bl*. The time taken to generate the first solution (say, $FSTime$) is slightly higher than the average run-time of *HMDS-Bl*, primarily because of the additional bound conditions in the *HMDS* algorithm. Further, *HMDS* is assured to return a solution at least as good as *HMDS-Bl*, if it is terminated after any length of time beyond $FSTime$. In the algorithm, the parameter *capT* denotes the upper bound on the allowable run-time of *HMDS* as a multiple of $FSTime$. Thus, $capT = 2^{10}$ indicates that *HMDS* will be allowed to run for at most $2^{10} \times FSTime\ ms$. Table 4.3 (Experiment 6, Section 4.4.3) shows the Average Improvement in Schedule lengths (AIS) with respect to *HMDS-Bl*, for various values of λ (= 3%, 5%, and 10%) and *capT* with *ops* fixed to 2. Here, AIS denotes the fractional decrease in the schedule length of *HMDS* with respect to the schedule length of *HMDS-Bl* (in *percentage*). As an acceptable balance between schedule length improvement and additional time overhead, we have fixed the value of *capT* to 2^{10} in our experiments. It may be observed from the table that for task graphs containing 54 nodes, $ops = 2$, $\lambda = 5\%$ and $capT = 2^{10}$, *HMDS* is able to achieve 6.4% improvement in average schedule lengths (about 1% less than the maximum possible *makespan* reduction), while still incurring run-times of less than 100 ms ($\approx 81\ ms$).

4.3.2.5 Complexity Analysis

To generate a single solution *HMDS* requires $O(|E| \times |P|)$ time, similar to *HMDS-Bl*. Given this base overhead, the total complexity of the algorithm is governed by the computational complexity associated with the $assignTasks()$ function. In the absence of the four pruning mechanisms, the overhead of $assignTasks()$ becomes $O(|P|^{|V|})$. By introducing the 1^{st} and 2^{nd} ($ops = 2$) pruning mechanisms of *HMDS*, the overhead reduces to $O(2^{|V|})$. In practice, through further addition of the 3^{rd} and 4^{th} pruning mechanisms, the *HMDS* algorithm is seen to scale appreciably well, as validated through our experiments. The 4^{th} pruning mechanism puts a cap on the run-time of *HMDS*, relative to *HMDS-Bl*.

Example Continued: We continue with the example in Section 4.3.1. Fig. 4.1c lists the

PFT and $rank_{pft}$ values corresponding to the example system shown in Figs. 4.1a and 4.1b. The priority list of tasks is obtained as: $taskList = \langle \tau_1, \tau_4, \tau_5, \tau_2, \tau_6, \tau_3, \tau_7, \tau_9, \tau_8, \tau_{10} \rangle$. The Gantt chart in Fig. 4.2h shows the schedule generated by *HMDS*. Comparing Fig. 4.2, we observe that *HMDS* (*makespan* 177; $ops = 2$, $\lambda = 10\%$, $capT = 2^2$) comprehensively outperforms *HMDS-Bl* (*makespan* 182), *HEFT* (*makespan* 198), *PEFT* (*makespan* 189), *PALG* (*makespan* 200), *PPTS* (*makespan* 187), and *PSLS* (*makespan* 184).

4.4 Experiments and Results

In this section, we experimentally evaluate the performance of *HMDS-Bl*, *HMDS*, and compare them against the state-of-art algorithms *HEFT* [85], *PEFT* [5], *PALG* [3], *PPTS* [24] and *PSLS* [112]. In our experiments, we consider the values of $ops = 2$, $\lambda = 5\%$, $capT = 2^{10}$, for the *HMDS* algorithm. In addition to these simulation-based experiments, a simple proof-of-concept implementation of the proposed work has been conducted on a real platform consisting of two heterogeneous processors *ATmega328p* (*Arduino Uno*) and *ATmega2560* (*Arduino Mega*) inter-connected through their serial ports. We are not including detailed description of the proof-of-concept real platform implementation here, as Chapter 3 contains an elaborate discussion on a similar implementation (refer Section 3.6). Before presenting the detailed results, we now describe the experimental setup and the performance metrics used by us.

4.4.1 Experimental Setup

The performance evaluation has been conducted through extensive simulation-based experiments using two real-world benchmark task graphs, namely, Gaussian Elimination [85] (refer Fig. 3.4a) and Epigenomics [39] (refer Fig. 3.4b). We now discuss data generation framework.

Data Generation Framework: An exhaustive set of experiments have been carried out using randomly generated data sets obtained by carefully varying a set of parameters.
1. **Number of tasks** $|V|$: Experiments have been conducted with different values of *matrix size* $\nu = \{6, 7, 8, 9, 10\}$. With these values of ν, five types of hypothetical DAGs having different number of tasks $|V|$ and edges $|E|$ are generated ($|V| = \{20, 27, 35, 44, 54\}$; $|E| = \{29, 41, 55, 71, 89\}$). Similarly, *parallel branch* $\vartheta = \{6, 7, 8, 9, 10\}$; DAGs having $|V| = \{28, 32, 36, 40, 44\}$ tasks and $|E| = \{32, 37, 42, 47, 52\}$ edges are generated.

2. **Number of processors**: $|P| = \{4, 8, 16, 32\}$

3. **Task execution times**: We have followed a three-step procedure to generate task execution time on each heterogeneous processor. First, the average execution time ($\overline{\omega_{DAG}}$), considering all tasks is determined in the task graph. Values of $\overline{\omega_{DAG}}$ used in this work: $\overline{\omega_{DAG}} = \{40, 80, 120, 160, 200\}$. Next, the average execution time ($\overline{\omega_j}$) for each task (τ_j), over all processors is found out. Values of $\overline{\omega_j}$ for the tasks are obtained from normal distributions having mean $\mu = \overline{\omega_{DAG}}$ and different values of standard deviation $\sigma = \{10, 20, 30\}$. Finally, the execution time ($\omega_{j,n}$) of τ_j on each processor p_n is determined. The $\omega_{j,n}$ values of a task τ_j are obtained for a normal distribution having mean $\mu = \overline{\omega_j}$ and different standard deviation values $\sigma = (\overline{\omega_j} \times \beta)$. The parameter β acts as the heterogeneity factor and ascertains the amount of skewness of a task's execution times on different processors. Values of β considered in this work are: $\beta = \{0.1, 0.25, 0.5, 0.75, 1\}$. After obtaining the initial $\omega_{j,n}$ values of a task τ_j, they are appropriately updated such that $\sum_{j=1}^{|V|} \sum_{n=1}^{|P|} \omega_{j,n}$ is equal to $|V| \times |P| \times \overline{\omega_{DAG}}$.

4. **Data communication workload**: Ratio of the time overhead associated with message transmission and task execution is referred to as *Communication-to-Computation Ratio* (CCR). Performance of the proposed schedulers *HMDS-Bl* and *HMDS* have been evaluated for different CCR values ($CCR = \{0.1, 0.5, 1, 2, 5\}$). The average communication workload $\overline{c_{DAG}}$ is then obtained as $\overline{c_{DAG}} = CCR \times \overline{\omega_{DAG}}$. Given $\overline{c_{DAG}}$, the average inter-task message size ($\overline{data_{DAG}}$; in Bytes) for a DAG is calculated as: $\overline{data_{DAG}} = \overline{c_{DAG}} \times \overline{B}$, where $\overline{B}$ ($= \frac{1}{|P| \times (|P|-1)/2} \sum b_{m,n}$ $(1 \leqslant m \leqslant |P|; 1 \leqslant n < m)$) denotes the average data communication bandwidth. We have carried out experiments for two different $\overline{B}$ values ($\overline{B} = \{5\ Gbps, 10\ Gbps\}$). The actual bandwidth $b_{m,n}$ of the communication channel between two processors p_m and p_n are obtained from a normal distribution having $\mu = \overline{B}$ and $\sigma = 0.2 \times \overline{B}$. These $b_{m,n}$ values are further updated appropriately such that $\sum_{m=1}^{|P|} \sum_{n=1}^{m-1} b_{m,n}$ becomes $|P| \times (|P| - 1)/2 \times \overline{B}$. The length of the output message ($data_{i,j}$) between a dependant task pair (τ_i, τ_j) is obtained from a normal distribution having $\mu = \overline{data_{DAG}}$ and $\sigma = 0.2 \times \overline{data_{DAG}}$. The $data_{i,j}$ values are then scaled in order to make $\sum data_{i,j}$ equal to $|E| \times \overline{data_{DAG}}$.

Simulation Framework: The simulation framework is written in C and is executed on a system having the following configuration: (i) Intel® Core™ i7-8550U CPU @ 1.80GHz ×8, (ii) 8 GiB Memory, and (iii) Ubuntu 18.04.2 LTS OS (64-bit).

4.4.2 Performance Metrics

Performance of the proposed methodology has been evaluated using three different parameters:

1. **Schedule Length Ratio**: The most commonly used performance measure of a scheduling algorithm on a task graph is the *schedule length/makespan*. Since a large set of task graphs with different input properties are used, we have employed a normalised *schedule length* measure called *Schedule Length Ratio* (*SLR*). This metric is used to compare performance of the proposed *makespan* minimization algorithms *HMDS-Bl* and *HMDS* against the existing state-of-art algorithms *HEFT* [85], *PEFT* [5], *PALG* [3], *PPTS* [24] and *PSLS* [112]. Given a task graph, *Schedule Length Ratio* (SLR) is defined as:

$$SLR = \frac{X_{ms}}{\sum_{\tau_j \in CP_{min}} \min_{p_n \in P} \omega_{j,n}} \tag{4.6}$$

   where X_{ms} represents the *makespan* achieved by an algorithm like, *HEFT*, *PEFT*, *PALG*, *PPTS*, *PSLS*, *HMDS-Bl* or *HMDS*. Assuming execution time of each task to be its minimum value over all heterogeneous processors, the denominator represents the sum of the execution times of all tasks in the critical path (CP_{min}) of the task graph. It may be noted that lower the value of achieved SLR better is the performance of the scheduling algorithm.

2. **Number of Improved Solutions**: We used this metric to pair-wise compare the performance of two scheduling strategies in a tabular format. In the table, we show the percentages of cases for which one strategy has performed better, similar or worse compared to the other.

3. **Run-time**: Through this metric, we have determined the average solution generation time taken by a scheduling strategy for data sets obtained by using a fixed set of parameter values.

4.4.3 Performance Results

In this subsection, we present detailed experimental results using two benchmark task graph models. In these experiments, each data point is the average over solutions generated with 100 different task graph data corresponding to a fixed set of parameters.

4.4.3.1 Experiment-1: Pair-wise *makespan* comparison of algorithms

Table 4.2 shows pair-wise performance comparisons among the following algorithms *HEFT*, *PEFT*, *PALG*, *PPTS*, *PSLS*, *HMDS-Bl* and *HMDS*. Specifically, the result corresponding to the $(row\ i, column\ j)^{th}$- entry in the table depicts the percentages of test cases for which the algorithm corresponding to the i^{th} row performs better, equal or worse than the algorithm in column j. A total of 50000 test cases using Gaussian Elimination task graphs have been considered for each pair of algorithms. For example, the $(1,1)^{th}$- entry in Table 4.2 shows that *HMDS* performs better, equal and worse in 85.9%, 3.6% and 10.5% test cases respectively, compared to *HEFT*.

Table 4.2: *Pair-wise makespan comparison of the scheduling algorithms*

		PALG	*PSLS*	*PPTS*	*PEFT*	*HEFT*	*HMDS-Bl*
HMDS	better	97.4%	81.7%	85.7%	81.9%	85.9%	78.9%
	equal	0.6%	9.4%	7.4%	10.4%	3.6%	21.1%
	worse	2.0%	8.9%	6.9%	7.7%	10.5%	0.0%
HMDS-Bl	better	95.7%	52.7%	62.1%	59.2%	76.8%	
	equal	0.4%	3.5%	3.6%	9.2%	2.8%	
	worse	3.9%	43.9%	34.3%	31.6%	20.4%	
HEFT	better	86.1%	26.5%	33.9%	27.6%		
	equal	4.3%	2.6%	1.1%	2.4%		
	worse	9.7%	70.9%	65.0%	70.0%		
PEFT	better	93.7%	35.0%	52.6%			
	equal	0.7%	13.8%	3.7%			
	worse	5.6%	51.2%	43.7%			
PPTS	better	90.8%	42.3%				
	equal	2.5%	2.2%				
	worse	6.7%	55.5%				
PSLS	better	94.8%					
	equal	0.6%					
	worse	4.6%					

4.4.3.2 Experiment-2: Comparison of schedule length ratios

This experiment measures the *schedule length ratio*s (*SLR*) of *HEFT*, *PEFT*, *PALG*, *PPTS*, *PSLS*, *HMDS-Bl* and *HMDS* for varying values of #tasks ($|V|$), #processors ($|P|$), heterogeneity (β) and Communication-to-Computation Ratios (CCR). Obtained results for

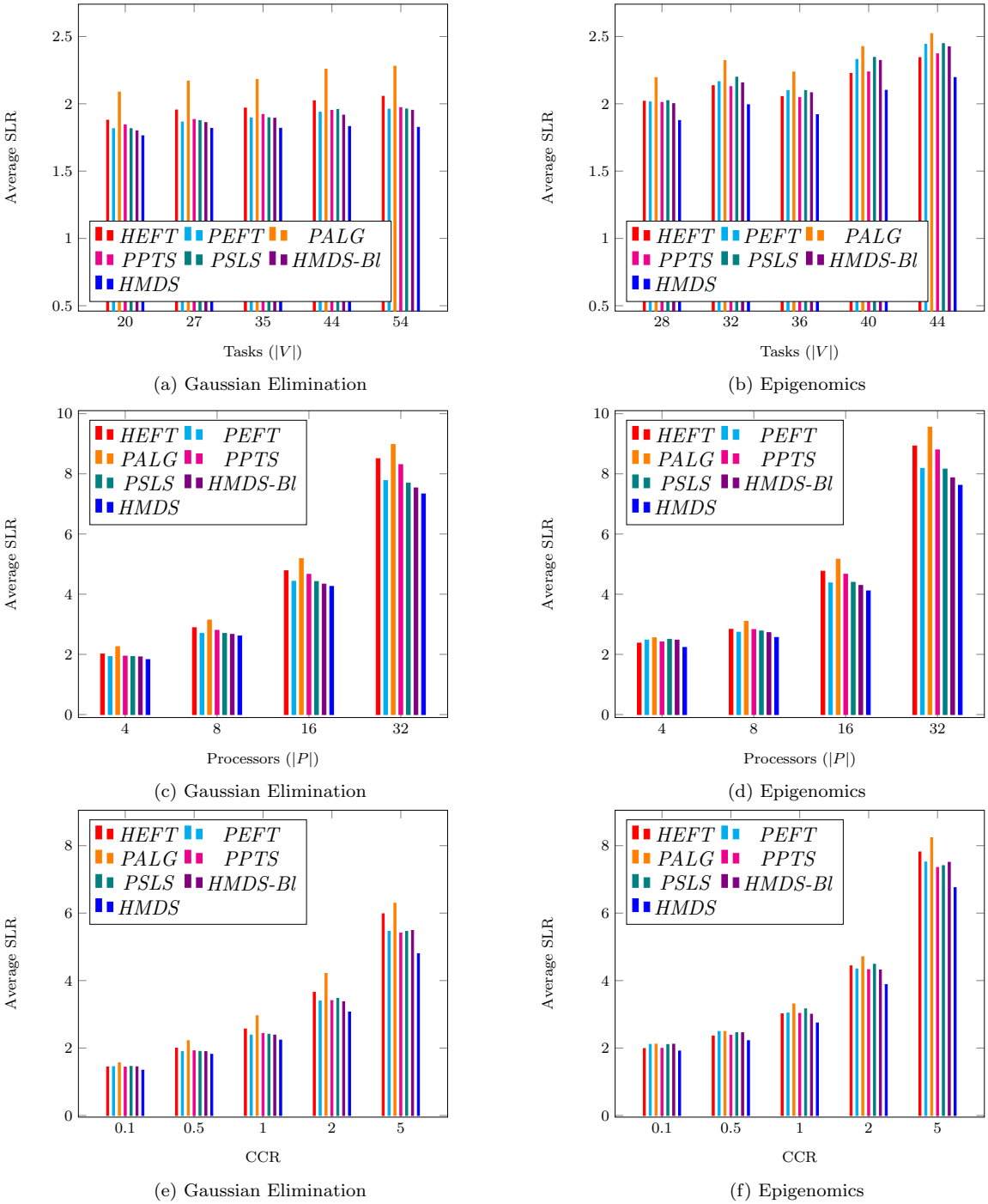
both Gaussian Elimination and Epigenomics are presented in Figs. 4.3, 4.4a and 4.4b. It may be observed that for all figures, *HMDS* comprehensively outperforms all the algorithms throughout in terms of achieved *makespans*.

Figs. 4.3a and 4.3b show average SLR values for the Gaussian Elimination and Epigenomics DAGs, as the number of tasks vary between 20 and 55 (ν for Gaussian Elimination and ϑ for Epigenomics are varied from 6 to 10). Here, the parameters $|P|$, *CCR* and β have been fixed at 4, 0.5 and 0.75, respectively. For Gaussian Elimination, *HMDS-Bl* is seen to deliver better results than *HEFT*, *PEFT*, *PALG*, *PPTS* and *PSLS*, in all cases, For Epigenomics, *HEFT* performs a bit better compared to *PEFT*, *PALG*, *PPTS*, *PSLS* and *HMDS-Bl*, when the number of tasks increases. *HMDS-Bl* delivers slightly better results than *PEFT*, *PALG* and *PSLS*, in all the cases.

Figs. 4.3c and 4.3d show average SLR as a function of #processors, while fixing $|V|$ to 44, *CCR* to 0.5 and β to 0.75. As an example of *HMSD's* performance, it may be observed in Fig. 4.3c (Gaussian Elimination) that for $|P| = 32$, the average schedule lengths of *HMDS* is lower than *HEFT*, *PEFT*, *PALG*, *PPTS*, *PSLS* and *HMDS-Bl* by approximately 16%, 6%, 22%, 13%, 5% and 3%, respectively. For Epigenomics (Fig. 4.3d), *HMDS* outperforms *HEFT*, *PEFT*, *PALG*, *PPTS*, *PSLS* and *HMDS-Bl* by approximately 17%, 7.4%, 25%, 15%, 7.1% and 3%, respectively. It may be noted that for both applications, the performance of *HMDS-Bl*, *PEFT* and *PSLS* improves with respect to *HEFT*, *PALG* and *PPTS* as the number of processors goes on doubling.

In Figs. 4.3e and 4.3f, we depict the average SLR as *CCR* is varied between 0.1 and 5. The parameters $|V|$, $|P|$ and β are set to 44, 4 and 0.75, respectively. For this case, the performance of *HMDS-Bl* is nearly same with *PEFT*, *PPTS* and *PSLS*. It may be seen that the average SLR increases with higher values of CCR due to an increase in overall workload. However, because of higher available parallelism, Gaussian Elimination generates shorter *makespans* than Epigenomics. It can also be observed from both the figures that the performance of *HEFT* and *PALG* gradually decreases with an increase in overall workload.

In Figs. 4.4a and 4.4b, we show the variation in SLR values as the degree of heterogeneity is increased from 0.1 to 1. Parameters $|P|$, CCR and $|V|$ are fixed at 4, 0.5 and 44. For Gaussian Elimination, it can be observed that *HMDS-Bl*, *PEFT*, *PPTS* and *PSLS* perform better than *HEFT*, while the trend is reverse for Epigenomics. For both applications, *PALG* may be seen to deliver worse results than the existing algorithms in all cases.



(a) Gaussian Elimination

(b) Epigenomics

(c) Gaussian Elimination

(d) Epigenomics

(e) Gaussian Elimination

(f) Epigenomics

Figure 4.3: *SLRs for varying #tasks, #processors and Communication-to-Computation Ratios.*

4.4.3.3 Experiment-3: Comparison of run-times

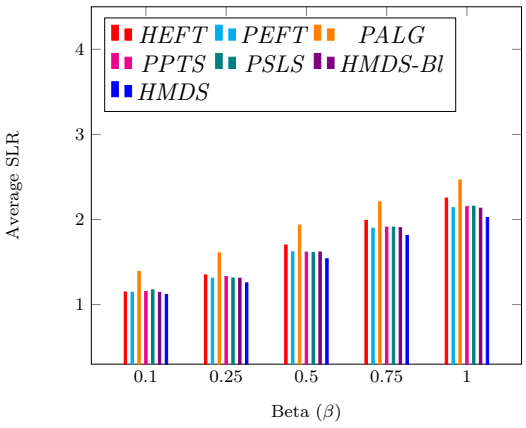
This experiment is a comparative analysis of the *run-times* of *HMDS-Bl*, *HEFT*, *PEFT*, *PALG*, *PPTS* and *PSLS*, for varying values of #tasks (Fig. 4.4c) and #processors (Fig. 4.4d) using Gaussian Elimination.

Fig. 4.4c shows average run-times, as the number of tasks is varied between 20 and 55. Values of the parameters $|P|$, *CCR* and β have been fixed at 4, 0.5 and 0.75. It can be observed that *PEFT*, *PPTS* and *HMDS-Bl* take similar run-times to generate the schedules, which is slightly higher than *HEFT* and *PALG's* run-times. Further in the figure, run-times of all the compared algorithms may be seen to be upper bounded by ≈ 80 microseconds, in all cases except *PSLS*.

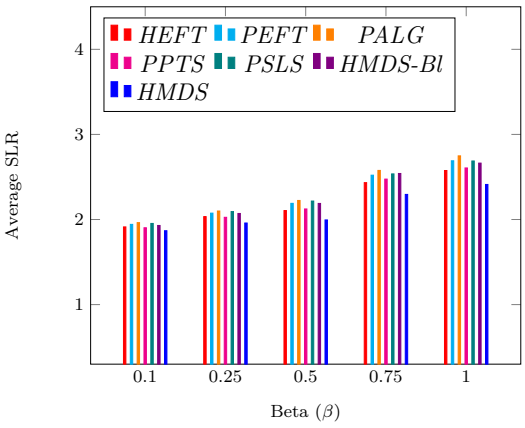
Fig. 4.4d shows average run-times as a function of processors, while fixing $|V|$ to 44, *CCR* to 0.5, and β to 0.75. It can be observed that the run-times of all the algorithms strictly increase with the number of processors. The algorithms' ranking can be viewed in terms of their average run-times from fastest to slowest as: $PALG \leqslant HEFT < HMDS\text{-}Bl < PEFT < PPTS < PSLS$. For Experiments-3, 4, 5, 6 and 7, we have not shown results for the Epigenomics application, since the trends of the results as obtained for Epigenomics are similar to that of Gaussian Elimination.

4.4.3.4 Experiment-4: Bound on #processor choices

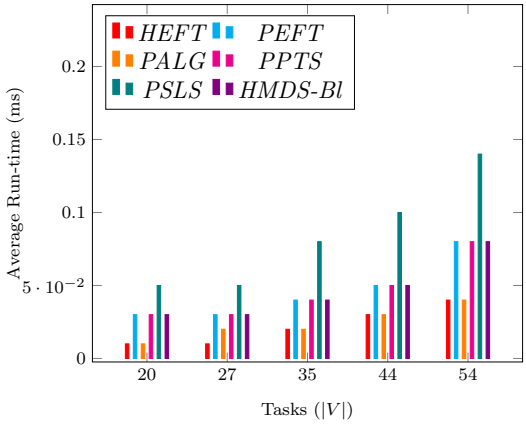
This experiment measures the *schedule length ratios (SLRs)* (Fig. 4.4e) and *run-times* (Fig. 4.4f) of *HMDS*, as the number of tasks are varied from 20 to 54 and the number of processor choices (*ops*) vary from 1 to 4. In this, the parameters $|P|$, *CCR*, and β are set to 4, 0.5 and 0.75. From Fig. 4.4e, it may be observed that for any given number of tasks, the average SLR value decreases sharply with an increase in the number of processor choices from $ops = 1$ to $ops = 2$. However, further increment of *ops* (from 2 to 3 or 3 to 4) delivers negligible additional decrease in SLRs. On the other hand, the solution generation times increase exponentially as the value of *ops* is increased from 2 to 4 (Fig. 4.4f). A detailed analysis of these result trends has been provided as part of the discussion on *mechanism*-2 of the pruning techniques applied in *HMDS* (refer Section 4.3.2).



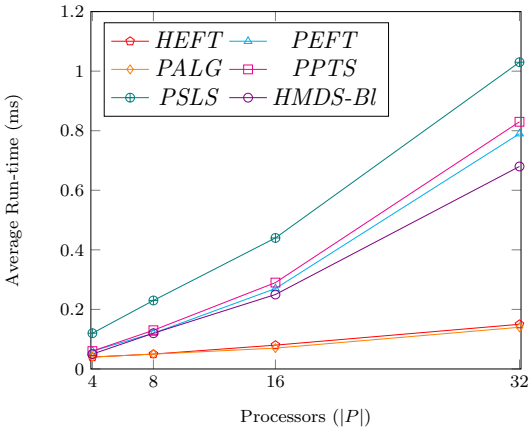
(a) Gaussian Elimination



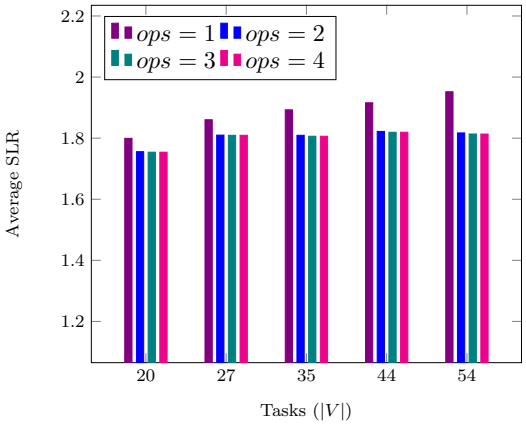
(b) Epigenomics



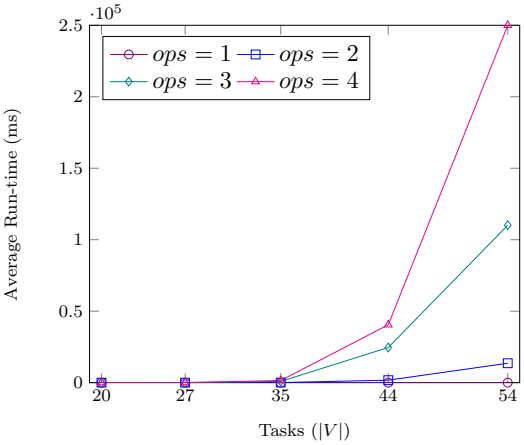
(c) Gaussian Elimination



(d) Gaussian Elimination



(e) Gaussian Elimination



(f) Gaussian Elimination

Figure 4.4: *(a) and (b) SLRs for varying heterogeneity; (c) and (d) Run-times for varying #tasks and #processors using Gaussian Elimination; (e) and (f) SLRs and run-times of HMDS for Gaussian Elimination.*

4.4.3.5 Experiment-5: Effect of O_{EFT} based bound

Fig. 4.5a depicts the average SLRs and run-times of *HMDS-Bl* as the number of tasks $|V|$ vary from 20 to 55 for Gaussian Elimination. Using Fig. 4.5a as the basis for comparison, Fig. 4.5b presents the Average Improvement in performance achieved by *HMDS*, while simultaneously showing the average slowdown suffered in the process, with respect to *HMDS-Bl*. Here, performance is measured in terms of Average Improvement in Schedule length ratios (AIS), which is defined as:

$$AIS = \frac{SLR_{HMDS\text{-}Bl} - SLR_{HMDS}}{SLR_{HMDS}} \times 100 \tag{4.7}$$

On the other hand, average slowdown ASD is defined as:

$$ASD = \frac{run\text{-}time_{HMDS}}{run\text{-}time_{HMDS\text{-}Bl}} \tag{4.8}$$

	SLR	*Run-time*
20	1.799453	0.02 *ms*
27	1.860302	0.03 *ms*
35	1.892773	0.04 *ms*
44	1.915857	0.05 *ms*
54	1.951962	0.08 *ms*

(a)

	$\|V\|$	$\lambda = 1$	$\lambda = 3$	$\lambda = 5$	$\lambda = 10$	$\lambda = 15$	$\lambda = 20$	$\lambda = 100$
AIS	20	0.69	1.3	2.1	2.38	2.48	2.48	2.5
ASD		$2^{0.5}$	$2^{0.6}$	$2^{1.8}$	2^{3}	$2^{3.5}$	$2^{3.8}$	$2^{4.3}$
AIS	27	1	1.9	2.3	2.65	2.7	2.74	2.8
ASD		$2^{0.7}$	$2^{2.4}$	$2^{3.8}$	$2^{6.1}$	$2^{7.2}$	$2^{7.6}$	2^{8}
AIS	35	1.7	3.6	4.1	4.51	4.58	4.58	4.6
ASD		$2^{1.3}$	$2^{3.8}$	$2^{6.1}$	$2^{8.8}$	$2^{9.8}$	$2^{10.3}$	$2^{10.9}$
AIS	44	2	3.9	4.6	4.9	5.12	5.13	5.14
ASD		2^{2}	$2^{6.2}$	$2^{8.8}$	$2^{12.2}$	$2^{13.7}$	$2^{14.3}$	2^{15}
AIS	54	3.9	6.1	6.9	7.2	7.3	7.39	7.4
ASD		$2^{3.3}$	$2^{8.9}$	$2^{11.6}$	$2^{15.3}$	$2^{16.7}$	$2^{17.1}$	$2^{17.3}$

(b)

Figure 4.5: *(a) SLRs and run-times of HMDS-Bl for Gaussian Elimination; (b) Average Improvement in Schedule length ratios (AIS in percentage) and Average slowdown (ASD) by HMDS; for varying λ using Gaussian Elimination. λ is the allowable O_{EFT} degradation bound in percentage.*

.

For any given number of tasks, results are presented for varying values of λ, which represents the allowable O_{EFT} (refer equation 4.5) degradation bound in *percentage*. Values of $|P|$, CCR and β are set to 4, 0.5 and 0.75. Discussion for the trends of the results in Fig. 4.5b has been done within the description of pruning *mechanism*-3 (refer Section 4.3.2).

4.4.3.6 Experiment-6: Effect of hard run time caps

Using the same experimental setup as employed for Experiment 5, we further examine the comparative performance of *HMDS* with respect to *HMDS-Bl* (by determining AIS;

refer equation 4.7) for various upper bounds on the allowable run-times for *HMDS*. Here, the upper bounds are obtained through the parameter *capT*, which is set as a multiple of *HMDS-Bl*'s run-time. *capT* essentially provides a mechanism to stop *HMDS's* search for better solutions after a pre-specified duration subsequent to search commencement. *HMDS* then returns the best solution found within this duration. The results obtained in Table 4.3 show that *HMDS* achieves a steady improvement in performance as more time is allowed in its search for better solutions. Further analysis on these results are provided in the discussion of the 4^{th} *mechanism* corresponding to the pruning techniques applied to *HMDS* (refer Section 4.3.2).

Table 4.3: *Average Improvement in SLR (AIS in percentage) by HMDS; for varying capT and λ using Gaussian Elimination. λ is the O_{EFT} degradation bound in percentage and capT is the allowable run-time bound*

λ	3%							5%							10%						
capT	2^1	2^2	2^4	2^6	2^8	2^{10}	2^{12}	2^1	2^2	2^4	2^6	2^8	2^{10}	2^{12}	2^1	2^2	2^4	2^6	2^8	2^{10}	2^{12}
AIS	1.6	2.4	3.8	5	5.8	6	6.1	1.5	2.2	3.2	4.5	5.6	6.4	6.7	1.4	2	2.9	3.9	5.2	6	6.5

4.4.3.7 Experiment-7: HMDS vs. HMDS-Bl

Using Gaussian Elimination as the task graph structure, Fig. 4.6a lists the average SLRs and run-times of *HMDS-Bl* as the number of tasks $|V|$ is varied from 54 to 209, on systems containing 4 and 8 processors, respectively. Fig. 4.6b depicts the Average Improvement in SLR achieved by *HMDS* over *HMDS-Bl* (Fig. 4.6a) along with corresponding run-times of *HMDS*. The parameters *CCR* and β are set to 0.5 and 0.75, while value of *capT* is varied from 6% to 10%. In Fig. 4.6b, we observe that for any given number of tasks, *HMDS* achieves a steady improvement in performance as more time is allowed in its search for better solutions. On the other hand, as is obvious, for any given bound on run-time, the performance degrades as the number of tasks and/or the number of processors increase. In spite of this, *HMDS* may be considered appreciably scalable and we see that even for 209 tasks, *HMDS* is able to deliver more than 3% (1.5%) performance gains on 4 (8) processors in less than 0.5 *Sec* (0.9 *Sec*). It may be noted that, these performance gains may be critical especially in many real-time cyber-physical control systems which often run in a repetitive loop, continuously controlling its associated plant. For example, a performance gain of say 2% means that control actuations can now be effected every 98 *ms*, if the original *makespan*

was 100 ms. This increased control actuation rate may potentially be critical towards enhancing stability of the designed system.

$\|P\|$	4							8						
$\|V\|$	54	77	104	135	152	189	209	54	77	104	135	152	189	209
SLR	1.951962	2.072274	2.166477	2.236507	2.392265	2.497077	2.582901	2.639374	2.689752	2.755731	2.771818	2.789517	2.791327	2.830357
Run-time	0.08 ms	0.14 ms	0.20 ms	0.28 ms	0.34 ms	0.49 ms	0.56 ms	0.16 ms	0.23 ms	0.36 ms	0.50 ms	0.63 ms	0.88 ms	0.93 ms

(a)

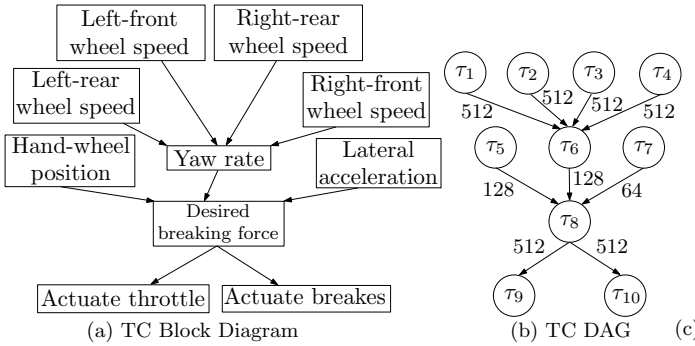
capT	$\|P\|$	4							8						
	$\|V\|$	54	77	104	135	152	189	209	54	77	104	135	152	189	209
2^6	*AIS* (%)	4.50	3.95	2.96	2.53	2.03	1.90	1.87	1.43	1.36	1.21	1.20	1.18	1.16	1.1
	Run-time (Sec)	0.005	0.009	0.01	0.02	0.02	0.03	0.04	0.01	0.01	0.02	0.03	0.04	0.06	0.06
2^8	*AIS* (%)	5.60	4.89	4.05	3.49	2.98	2.40	2.29	1.60	1.57	1.47	1.38	1.34	1.32	1.30
	Run-time (Sec)	0.02	0.04	0.05	0.07	0.09	0.13	0.14	0.04	0.06	0.09	0.13	0.16	0.23	0.24
2^{10}	*AIS* (%)	6.40	5.74	4.97	3.79	3.68	3.28	3.02	1.80	1.76	1.67	1.59	1.56	1.55	1.53
	Run-time (Sec)	0.08	0.14	0.20	0.29	0.35	0.50	0.57	0.16	0.24	0.37	0.51	0.65	0.90	0.95

(b)

Figure 4.6: *(a) SLRs and run-times of HMDS-Bl for varying $|P|$ and $|V|$ using Gaussian Elimination; (b) AIS and run-times of HMDS for varying $|P|$, $|V|$ and capT using Gaussian Elimination.*

4.5 Case Study: Traction Control System

To exhibit the practical applicability of proposed algorithms to real-world designs, we present a case study using the *Traction Control* (*TC*) application in automotive systems. The TC application helps to improve stability of the car when road conditions are slippery [41].



(a) TC Block Diagram (b) TC DAG

	p_1	p_2	p_3
τ_1	205	200	195
τ_2	207	200	193
τ_3	190	210	200
τ_4	200	198	202
τ_5	150	155	145
τ_6	297	300	303
τ_7	175	180	170
τ_8	405	400	395
τ_9	146	154	150
τ_{10}	199	201	200

(c) TC Execution time table

	p_1	p_2	p_3
$pow_{n,ind}$	0.07	0.04	0.03
$C_{n,ef}$	1.0	0.8	0.8
m_n	2.5	2.5	2.9
$f_{n,ee}$	0.29	0.21	0.22
$f_{n,max}$	1.0	1.0	1.0

(d) Power parameters

Figure 4.7: *Traction Control (TC): (a) TC Block Diagram; (b) TC DAG; (c) Execution times (in ms) of tasks in TC DAG on three processors; (d) Power parameters of three heterogeneous processors.*

Fig. 4.7a displays the block structure of TC as adapted from [41] and Fig. 4.7b shows its corresponding task graph representation. This task graph consists of 10 task nodes that

need to be scheduled on a distributed platform having three heterogeneous processors p_1, p_2 and p_3. The bandwidths of the communication links between each pair of processing devices are considered to be: $b_{1,2} = b_{2,1} = 250\ KB/s$, $b_{2,3} = b_{3,2} = 500\ KB/s$, $b_{3,1} = b_{1,3} = 1\ MB/s$, and $b_{1,1} = b_{2,2} = b_{3,3} = \infty$. Fig 4.7c depicts the execution times of each task associated with traction control, on the three heterogeneous processors. Each edge $e_{i,j}$ is assigned with a positive weight $data_{i,j}$ (in *bytes*; for example, $data_{1,6} = 512$ *bytes*) that represents the size of the message to be transferred from task τ_i to task τ_j. We have employed *HMDS*, *HMDS-Bl* and *PEFT* to generate three separate schedules for the TC task graph. Figs. 4.8a, 4.8b and 4.8c show the Gantt charts of the schedules obtained using *HMDS* ($ops = 2$, $\lambda = 5\%$, $capT = 2^1$), *HMDS-Bl* and *PEFT*. It may be observed that, *HMDS* is able to deliver a schedule with a lower *makespan* (1284 ms) compared to *HMDS-Bl* ($makespan = 1288\ ms$) and *PEFT* ($makespan = 1298\ ms$).

Discussion: In Real-Time Cyber-Physical Systems (*RT-CPSs*) involving control tasks similar to the TC application discussed above, having execution schedules with lower *makespans* may be beneficial in many ways. As an instance, we now present an illustration to show how the additional slack time acquired through a lower *makespan* schedule may be used in a Real-Time Traction Control (*RT-TC*) system to improve its energy efficiency.

Let us assume an end-to-end application deadline of 1500 ms for our RT-TC task graph (Fig. 4.7b). Given the *makespans* of the *HMDS* (1284 ms; Fig. 4.8a), *HMDS-Bl* (1288 ms; Fig. 4.8b) and *PEFT* (1298 ms; Fig. 4.8c) schedules, the available slack times for *HMDS*, *HMDS-Bl*, *PEFT* become 216 ms (1500 $ms - 1284\ ms$), 212 ms (1500 $ms - 1288\ ms$), 202 ms (1500 $ms - 1298\ ms$), respectively. Now, we use a prominent existing energy-aware scheduling algorithm called *DECM* [95] to minimize dynamic energy dissipations associated with the obtained *HMDS*, *HMDS-Bl* and *PEFT* schedules, by using the slack times available with them. For the three processors considered in the TC case study, we assume the same power parameters as used in [95]. Fig. 4.7d depicts an adapted version of the processor power parameters presented in Table 3 of [95]. The *DECM* algorithm takes as input a *schedule* along with its associated *slack*. In order to reduce energy dissipation, *DECM* modifies the input schedule by changing *task start times* and *processor operation frequencies during task execution*, while keeping *task-to-processor assignments* and *task execution order* same as the input schedule. The algorithm also ensures that task executions do not overlap and the end-to-end deadline is not violated in the process. It may be noted that all processors are assumed to operate at their maximum frequencies for the entire *makespan* in the input

schedule.

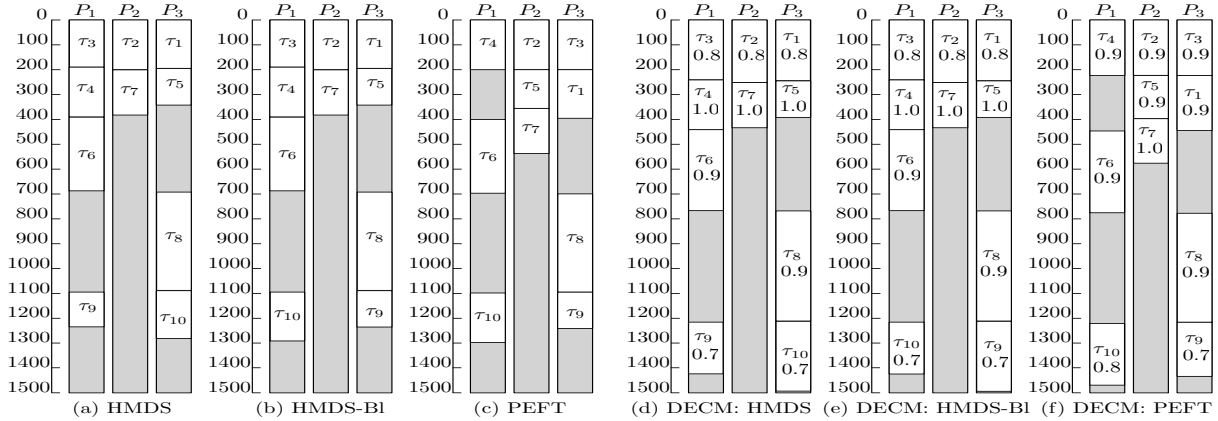


Figure 4.8: *TC Schedule Gantt charts: (a) HMDS (makespan = 1284 ms); (b) HMDS-Bl (makespan = 1288 ms); (c) PEFT (makespan = 1298 ms); (d) DECM: HMDS (makespan = 1495 ms, E(G) = 1610.47 W); (e) DECM: HMDS-Bl (makespan = 1499 ms, E(G) = 1624.35 W); (f) DECM: PEFT (makespan = 1470 ms, E(G) = 1643.31 W).*

Figs. 4.8a and 4.8d depict the Gantt charts of the input *HMDS* schedule and corresponding *DECM* generated modified schedule for the considered RT-TC application. It may be observed from the figures that task start times and normalized processor frequencies (where the maximum normalized processor frequency is represented as 1.0) may get changed in the output schedule. For instance, in Figs. 4.8a and 4.8d, we see that the ⟨start time, processor frequency⟩ for tasks τ_6, τ_{10} get changed from $\langle 390\ ms, 1.0\rangle$ and $\langle 1084\ ms, 1.0\rangle$ in the input schedule to $\langle 438\ ms, 0.9\rangle$, $\langle 1209\ ms, 0.7\rangle$ in the output schedule. Using the same mechanism for calculating the energy consumed by the schedule as presented in [95], the energy consumed by the original and modified *HMDS* schedules are obtained as 1986.56 W and 1610.47 W, respectively. Similarly, energy consumed by the modified *DECM* generated *HMDS-Bl* and *PEFT* schedules are obtained as 1624.35 W and 1643.31 W, respectively. Thus, we observe that even for this small RT-TC task graph, due to the lower *makespan* and higher associated slack time with *HMDS* (compared to *HMDS-Bl* and *PEFT*), the *DECM* algorithm is able to fetch higher energy savings when *HMDS* is used as the input schedule. This outcome therefore highlights an important advantage towards deriving lower *makespan* schedules for real-time embedded systems.

4.6 Summary

In this chapter, we have first proposed a list-based static scheduler named *HMDS-Bl*, whose objective is to generate schedules with minimum *makespans* for task graphs on heterogeneous platforms. Unlike the existing works such as *HEFT*, *PEFT*, etc., *HMDS-Bl* employs the actual communication cost between dependent task pairs (instead of average communication cost) to accurately incorporate the effects of heterogeneous data transmission times during task priority evaluation. Experimental evaluation shows that *HMDS-Bl* is able to deliver slightly better performance than state-of-the-art algorithms such as *HEFT*, *PEFT*, *PPTS*, etc. Subsequently, *HMDS-Bl* is empowered with a low-overhead depth-first branch and bound search technique called *HMDS*, to provide further improvements in performance. Experiments using two benchmark task graphs reveal the scheme's practical efficacy. From the experimental results, we can conclude that among the static scheduling algorithms studied in this chapter, *HMDS* comprehensively outperforms the others. Finally, the practical adaptability of the proposed work is shown using a real-world case study on traction control.

As discussed in the motivation section (refer Section 1.2 of Chapter 1), large systems which constitute multiple control subsystems typically follow a federated resource allocation policy as it allows simpler design, albeit, at the cost of significantly lower resource utilizations, in many cases. In order to improve the usage efficiency of available processing and network resources, in the next chapter, we extend our heterogeneous DAG scheduling framework to enable the co-scheduling of multiple independent real-time periodic DAG applications.

Chapter 5

DPMRS: An Energy-aware Real-time Scheduling of Multiple Periodic DAGs on Heterogeneous Systems

5.1 Introduction

The works are done in Chapters 3 and 4 dealt with single DAG-structured applications on heterogeneous distributed platforms. Large and complex Cyber-Physical Systems such as those in the automotive domain consists of multiple sub-systems, such as cruise control systems, anti-lock braking systems, fuel injection system, etc. These sub-systems are operated with their own dedicated control applications, each of which is represented as a precedence-constrained task graph. Traditionally, these individual task graphs are implemented on their own dedicated processing platforms. These systems are said to follow a federated architecture as each sub-component is associated with its own separate platforms. In contrast, an integrated execution architecture allows the combined execution of multiple task graph applications on a single consolidated platform. In general, federated architectures allow specification requirements related to timeliness, safety, energy, etc., to be satisfied while keeping the design methodology simpler compared to an integrated execution architecture. However, federated execution results in systems where processing, as well as network resources, are often severely under-utilized due to poor sharing of resources between sub-components. As a consequence, a federated architecture may result in higher design costs compared to an integrated execution.

Very few research works have targeted the co-scheduling of multiple real-time DAG applications on a single consolidated processing platform (refer Section 2.5 of Chapter 2), in spite of its practical importance. This is primarily due to the inherent computational and design complexity involved in this problem. Therefore in this work, we have endeavored to develop a scheduling strategy for multiple real-time task graphs on a distributed platform consisting of heterogeneous processors.

The key contributions of this work are summarized as follows:

1. We propose a static list-based real-time scheduling algorithm for *multiple independent periodic DAG applications* on a distributed heterogeneous platform while satisfying timing, resource and precedence constraints.

2. The objective of the scheduling algorithm is to minimize energy dissipated by the system during execution of a given set of DAGs using the DVFS approach.

3. The efficacy of the designed algorithm has been exhibited through four benchmark task graphs: CyberShake [39], Stencil [68], Gaussian Elimination [85] and Epigenomics [67]. The benchmarks have been chosen from diverse application domains namely, earthquake science, partial differential equations, linear algebra and bioinformatics.

The remainder of the chapter is organized in the following manner. Section 5.2 presents the system models and an example system that is used as a running example in the rest of the chapter. Section 5.3 describes the details of the proposed scheduling policies. Section 5.4 provides the experimental results. Section 5.5 exhibits a real-world case study on an automotive control system. Finally, Section 5.6 concludes the chapter.

5.2 System Models

This section presents the application and platform model, power and energy model, problem statement along with an example system, related to the proposed work.

5.2.1 Application and Platform Model

A real-time application say, r having deadline D^r is represented as a $DAG\ G^r(V^r, E^r)$, where the set of vertices $V^r = \{\tau_1, \tau_2, .., \tau_{|V^r|}\}$ represents tasks and E^r, the set of edges, represents precedence-constraints between task pairs. While individual tasks within an application are

non-preemptive, the application as a whole can be considered to allow restricted preemption. For example, an edge say, $e_{i,j} = (\tau_i, \tau_j) \in E^r$ denotes the dependency/precedence constraint between two tasks τ_i and τ_j. That is, τ_j can only start after τ_i completes execution and its output reaches τ_j. The edge $e_{i,j}$ is labeled with a positive weight $data_{i,j}$ indicating size of the message to be transmitted. We assume that each independent DAG has a single source node and a single sink node. A dummy source (sink) node with dummy edges connecting actual source (sink) nodes, is used in the situation where a DAG has more than one source (sink) node. The set of all direct successors (predecessors) of a task τ_j is represented as $succ(\tau_j)$ ($pred(\tau_j)$). The considered system consists of a set of independent persistently executing *periodic* real-time DAG applications $G = \{G^1, G^2, \ldots, G^r, \ldots, G^{|G|}\}$ having respective deadlines $D = \{D^1, D^2, \ldots, D^r, \ldots, D^{|G|}\}$ (refer Fig. 5.1). Each of these applications are assumed to commence execution at system start (i.e., time *zero*) and continue to execute periodically until the system is turned off. Each application G^r has an implicit deadline in which the deadline of the application is exactly equal to its period. The period is known to be the time interval between the release times of two consecutive application instances.

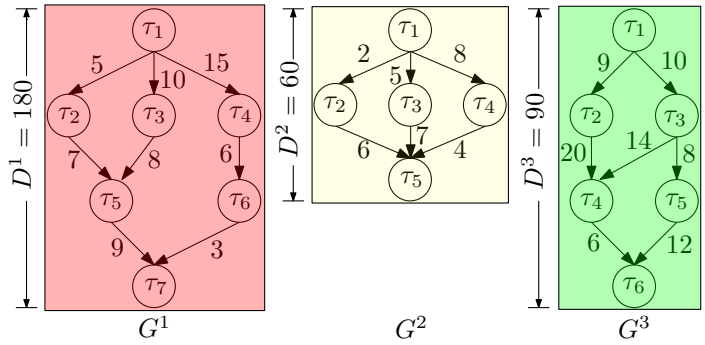


Figure 5.1: *An example of three independent application DAGs.*

We consider a fully-connected heterogeneous computing platform $P = \{p_1, p_2, ..., p_{|P|}\}$, where each processor pair $\langle p_x, p_y \rangle$ is connected using logically dedicated (possibly distinct bandwidths) communication links. As these bidirectional logical links can possibly have distinct bandwidths (heterogeneous), a matrix B of size $|P| \times |P|$ is used to represents the bandwidths between all processor pairs. An element say, $b_{x,y}$ of B indicates the data transmission rate between processor pairs p_x and p_y. Any processor say, p_y in the platform can operate at a set of α_y discreet frequencies $F_y = \{f_{y,1}, f_{y,2}, \ldots, f_{y,\alpha}, \ldots, f_{y,\alpha_y}\}$, such

that $f_{y,1}$ and f_{y,α_y} represent the minimum and maximum available frequency of p_y. The worst case execution time ($WCET$) of a task τ_j on p_y at frequency f_{y,α_y} is denoted as $\omega_{j,y}$. Then, the $WCET$ of τ_j on p_y at its frequency $f_{y,\alpha}$ can be determined as $\omega_{j,y} \times \frac{f_{y,\alpha_y}}{f_{y,\alpha}}$. Table 5.1 shows the $WCETs$ of each task on three heterogeneous processors when they are at maximum frequencies. Given $b_{x,y}$ and $data_{i,j}$, the message communication time $c_{i,j}^{x,y}$ from τ_i (executes on p_x) to τ_j (executes on p_y) is determined as: $c_{i,j}^{x,y} = data_{i,j}/b_{x,y}$. The message communication overhead is assumed to be negligible (i.e., $c_{i,j}^{x,y} = 0$) when both the tasks τ_i and τ_j are allocated to the same processor.

Table 5.1: *Execution times of tasks on three heterogeneous processors*

	G^1							G^2					G^3					
	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_1	τ_2	τ_3	τ_4	τ_5	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
p_1	11	9	29	10	15	8	10	28	12	9	8	8	4	12	13	15	27	17
p_2	16	10	14	12	13	13	22	7	11	7	17	6	12	15	9	10	16	19
p_3	12	23	18	8	9	15	21	13	15	14	12	12	23	32	11	21	25	6

5.2.2 Power and Energy Model

During the active operation of an embedded processor p_y, power is usually dissipated in three different ways: (i) *static power* pow_y^s, (ii) *dynamic power* pow_y^d and (iii) *independent power* pow_y^{ind}. The static power component pow_y^s is present for the entire duration when the processor is *on* and can be majorly attributed to the leakage from the millions of gates that constitute the processor. The frequency-dependent dynamic power loss pow_y^d depends on processor activity and is heavily sensitive to the processor's operating frequency at a given time. pow_y^{ind} represents the frequency-independent dynamic power loss which can only be removed when the processor is in *sleep* state. Thus, the total power dissipation combining these three factors when p_y is operating at frequency $f_{y,\alpha}$ is given by the expression:

$$pow(f_{y,\alpha}) = pow_y^s + h \times (pow_y^{ind} + pow_y^d) = pow_y^s + h \times (pow_y^{ind} + C_y^{ef} \times (f_{y,\alpha})^{m_y}) \quad (5.1)$$

where h represents the processor state ($h = 1$ indicates the *active* state and 0, the *sleep* state) and $f_{y,1} \leqslant f_{y,\alpha} \leqslant f_{y,\alpha_y}$; f_{y,α_y} denotes the maximum frequency of p_y. The effective switching capacitance C_y^{ef} and the dynamic power exponent m_y are processor-dependent constants.

Two types of energy management schemes are typically used at the operating system level. The first one is Dynamic Power Management (DPM), where certain parts of a system are strategically turned *off* when the processors are not in the active state. The other is Dynamic Voltage and Frequency Scaling (DVFS), which reduces power dissipation by slowing down processor speeds (operating frequency). In this work, we employ DVFS as our chosen energy management mechanism.

Dynamic energy dissipation for a task τ_j on a processor p_y (having maximum frequency f_{y,α_y}) at current frequency $f_{y,\alpha}$ is determined as:

$$\mathcal{E}_d(\tau_j, p_y, f_{y,\alpha}) = (pow_y^{ind} + C_y^{ef} \times (f_{y,\alpha})^{m_y}) \times \omega_{j,y} \times \frac{f_{y,\alpha_y}}{f_{y,\alpha}} \tag{5.2}$$

The total dynamic power dissipation associated with the execution of all tasks of an application G^r is calculated as

$$\mathcal{E}_d(G^r) = \sum_{j=1}^{|V^r|} \mathcal{E}_d(\tau_j, p_{\rho[j]}, f_{\rho[j],\varrho[j]}) \tag{5.3}$$

where $\rho[j]$ and $\varrho[j]$ denote the assigned processor and selected frequency level (of $\rho[j]$) for the execution of τ_j. Thus, $f_{\rho[j],\varrho[j]}$ denotes the operating frequency of processor $\rho[j]$ when τ_j executes on it. Static energy dissipation of the r^{th} application is given by

$$\mathcal{E}_s(G^r) = \sum_{y=1}^{|P|} pow_y^s \times SL(G^r) \tag{5.4}$$

where $SL(G^r)$ represents the schedule length of an application G^r, obtained by a specific scheduling algorithm. The total energy consumption ($\mathcal{E}(G^r)$) associated with the execution of G^r is given by the sum of both its static and dynamic energy dissipation [99] components, that is:

$$\mathcal{E}(G^r) = \mathcal{E}_s(G^r) + \mathcal{E}_d(G^r) \tag{5.5}$$

Reducing the frequency of a processor usually causes reduction in power and energy but longer task execution times. Although, dynamic power pow_y^d reduces as processor frequency is lowered, it may not always lead to lower energy, as task's execution times become higher. In general, a processor p_y has a critical operating frequency f_y^{cr} at which energy consumed during the execution of a task is minimum. This frequency f_y^{cr} is represented as [56, 88]: $f_y^{cr} = \sqrt[m_y]{pow_y^{ind}/(m_y - 1)C_y^{ef}}$.

Problem Statement *Given a periodic real-time application set G and a fully connected heterogeneous distributed processing platform P, where each processor p_y can operate at α_y discreet frequency levels, determine a feasible static schedule which minimizes overall energy, while satisfying constraints related to precedence, resource and timing.*

An Example System: Let us consider three independent real-time applications given in Fig. 5.1 to be executed on three heterogeneous processors. $D^1 = 180$, $D^2 = 60$ and $D^3 = 90$, are the deadlines of applications G^1, G^2 and G^3, respectively. Edge weights in any of these DAGs denote message sizes. The WCETs of each task on three processors are shown in Table 5.1. Here, we assume that the communication links between each pair of processors have heterogeneous bandwidths ($b_{1,2} = b_{2,1} = 1$, $b_{2,3} = b_{3,2} = 3$, $b_{3,1} = b_{1,3} = 2$, and $b_{1,1} = b_{2,2} = b_{3,3} = \infty$). The power parameters of the three processors have been adopted from [95, 99]. Table 5.2 depicts the power parameters.

Table 5.2: *Power parameters of three heterogeneous processors*

p_y	pow_y^s	pow_y^{ind}	C_y^{ef}	m_y	f_y^{cr}	f_{y,α_y}
p_1	0.01	0.02	1.3	2.9	0.19	1.0
p_2	0.01	0.05	0.5	2.1	0.32	1.0
p_3	0.01	0.04	0.2	3.0	0.46	1.0

5.3 The Proposed Schedulers

This section presents a list-based heuristic algorithm for DVFS-enabled energy-aware scheduling of a set of periodic independent DAGs, executing on a heterogeneous computing system. The proposed strategy has been named *DVFS-enabled Periodic Multi-DAG Real-time Scheduler for heterogeneous systems* (*DPMRS*).

5.3.1 The DPMRS Algorithm

The proposed scheduler *DPMRS* takes a set of periodic applications G and a heterogeneous computing platform P as inputs. All applications are assumed to commence execution at system start (i.e., zero) and continue to execute periodically until the system is turned off. The objective of *DPMRS* is to generate an energy-efficient static schedule for the set of applications for one hyperperiod. During system operation, this schedule repeats every hyperperiods. The goal of *DPMRS* is to non-preemptively assign each task of each instance

of the set of co-executing applications to a specific processor at an appropriate frequency with a chosen start time (which respect periodicity of the concerned application instance), such that the aggregate energy dissipated over the schedule length is minimized.

The pseudocode of *DPMRS* is presented in Algorithm 10. It consists of *three* functions: (i) *DPMRS()* (Algorithm 10), (ii) *periodicMerge()* (Algorithm 8), and (iii) *ERRRank()* (Algorithm 9). The main function *DPMRS()*, which conducts the overall scheduling, calls function *periodicMerge()* to merge all independent application DAGs into a single DAG at the first step. Given the merged DAG, *DPMRS()* next calls *ERRRank()* to compute two different parameters namely, (1) *Expected Relative Residual-workload* for each task-processor pair and (2) A rank $\mathcal{R}$ value for each task which is used to generate a task priority order. This priority order of tasks is used to fix the actual sequence in which tasks are considered for processor mapping and scheduling in the *DPMRS()* function. Next, we discuss these functions in detail and present their pseudocodes.

Algorithm 8: $periodicMerge(G, P)$

Input: Application set G and processor set P
Output: Determines merged periodic DAG G^0

$D^0 = LCM(D)$;
for $(r = 1; r \leqslant |G|; r++)$ **do**
  for $(n = 1; n \leqslant I^r; n++)$ **do**
    $\iota = \sum_{k=1}^{r-1} I^k + n$; /* *Id of the* n^{th} *instance DAG* (G_n^r) *of application* $G^r(V^r, E^r)$ *within merged DAG* G^0 */
    $AT[\iota] = (n-1) \times D^r$;
    $RD[\iota] = n \times D^r$;
    $ET[\iota] = |V^0| + 1$; /**id of source node of* G_n^r *within* G^0*/
    $V^0 = V^0 \cup V^r$; $E^0 = E^0 \cup E^r$;
    if $1 < n$ *and* $n \leqslant I^r$ **then**
      $E^0 = E^0 \cup \{e\}$; /* *Add a dummy edge e from sink node of* G_{n-1}^r *to source node of* G_n^r */
    $ST[\iota] = |V^0|$; /* *id of sink node of* G_n^r *within* G^0 */
Add dummy source node τ_0 and sink node τ_{exit} to G^0;
Update V^0, E^0;
return $G^0(V^0, E^0)$;

5.3.1.1 Function periodicMerge()

Each persistently executing application $G^r \in G$ starts at time *zero* (i.e., arrival time of the first instance $AT[G^r_1]$) and repeats at its own periodicity D^r (i.e., the n^{th} instance starts at $AT[G^r_n] = (n-1) \times D^r$). The applications being periodic, the time interval say D^0 (referred to as hyperperiod), after which arrival times of all application instances synchronize, is given by $D^0 = LCM(D)$ where, $D = \{D^1, D^2, \ldots, D^{|G|}\}$. In each hyperperiod, any application G^r sequentially invokes I^r $(= D^0/D^r)$ instances. It may be appreciated that, sequential execution of the I^r instances of any application DAG G^r can be modeled as the execution of a single composite DAG, where (i) end-to-end deadline of the composite DAG is D^0, (ii) the I^r DAG instances are arranged such that the sink node of any instance say G^r_{n-1} is connected to the source node of G^r_n, through a dummy edge to enforce precedence relationship between them, (iii) the execution start time of the source node of any instance say G^r_n happens on or after $AT[G^r_n] = (n-1) \times D^r$, (iv) the completion time of any instance say G^r_n happens on or before $n \times D^r$. All these composite DAGs can further be modeled as a single merged DAG G^0 by connecting their individual source and sink nodes to a single dummy source (τ_0) and sink node (τ_{exit}). It may be noted that G^0 has an end-to-end deadline of D^0. The nodes of G^0 are indexed such that the number associated with the j^{th} node of G^r_n (in G^0) becomes: $\sum_{a=1}^{r-1} |V^a| \times I^r + \sum_{b=1}^{n-1} |V^r| + j$ (refer Fig. 5.2). The pseudocode of *periodicMearge()* is presented in Algorithm 8. In the algorithm, the vectors AT, RD, ET and ST store respectively the arrival times, relative deadlines, entry tasks and sink tasks of all instances in G^0.

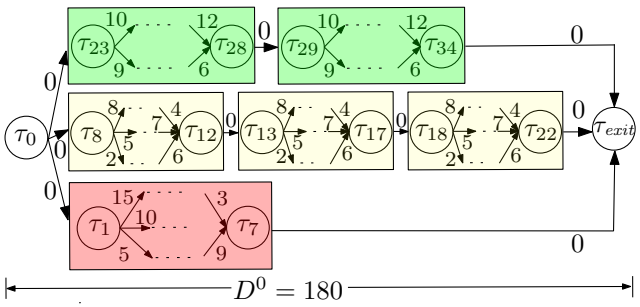


Figure 5.2: *Merged periodic application DAGs.*

Using the same example detailed in Section 5.2, Fig. 5.2 shows the merged DAG G^0 obtained by combining the three DAGs in Fig. 5.1. The task nodes in G^0 have been uniformly renumbered. Deadline of G^0 is: $D^0 = LCM(D) = 180$.

5.3.1.2 Function ERRRank()

This function determines *two* parameters: (i) *Expected Relative Residual-workload* and (ii) *rank*.

Algorithm 9: $ERRRank(G^0, P)$

Input: Task graph G^0 and processor set P
Output: Determines *Expected Relative Residual-workloads* for all task-processor pairs and rank of all tasks

1 $\iota = ST.length$; /* *total number of application instances in* G^0 */
2 **for** $\tau_j = \tau_{exit}$ *down to* τ_0 **do**
3 **for** *each processor* p_y *in* P **do**
4 **if** $\tau_j = \tau_{exit}$ **then**
5 $ERR[\tau_j, p_y] = 0$;
6 **else**
7 **if** $\tau_j = ST[\iota]$ ***and*** $\iota \geqslant 1$ **then**
8 $ERR[\tau_j, p_y] = D^0 - RD[\iota]$;
9 $\iota = \iota - 1$;
10 **else**
11 $ERR[\tau_j, p_y] = \max_{\tau_k \in succ(\tau_j)} \left[\min_{p_z \in P} \{ERR[\tau_k, p_z] + \omega_{k,z} + c_{j,k}^{y,z}\} \right]$;
12 Compute $\mathcal{R}[\tau_j]$ using equation 5.6;
13 $HRS = \max_{\tau_k \in succ(\tau_j)} \mathcal{R}[\tau_k]$;
14 **if** $\mathcal{R}[\tau_j] \leqslant HRS$ **then**
15 **for** *each processor* p_y *in* P **do**
16 $ERR[\tau_j, p_y] = ERR[\tau_j, p_y] \times \frac{HRS+\delta}{\mathcal{R}[\tau_j]}$;
17 Recompute $\mathcal{R}[\tau_j]$ using equation 5.6;
18 **return** [ERR, $\mathcal{R}$];

Expected Relative Residual-workload ($ERR[\tau_j, p_y]$): This parameter gives an optimistic estimate of the relative residual workload over all paths from node τ_j (after τ_j completes execution on processor p_y) to the sink node τ_{exit}. The function *ERRRank()* (refer Algorithm 9) calculates the ERR values of all task-processor pairs by traversing the merged task graph G^0 backward from τ_{exit} to τ_0. It may be observed from lines 4 and 5 that the residual-workload for the sink task τ_{exit} of G^0 on each processor p_y is *zero* (i.e., $ERR[\tau_{exit}, p_y] = 0$). For the other tasks in general, $ERR[\tau_j, p_y]$ is recursively determined relative to the minimum ERR value of τ_{exit} over all processors available in the system as depicted through the equation

in line 11. The ERR value of τ_j on p_y is obtained as the maximum ($\max_{\tau_k \in succ(\tau_j)}[\dots]$) optimistic residual-workload over all paths starting from τ_j leading to the sink node. The optimistic residual workload over paths via a particular successor node, say τ_k (of τ_j) is calculated in the following manner. We find the minimum ($\min_{p_z \in P}\{\dots\}$) residual workload for the path through τ_k, considering τ_k to be mapped to each of the available alternative processor choices. The residual workload for any particular processor choice, say p_z (for τ_k) is calculated as the summation over three terms: (i) $ERR[\tau_k, p_z]$: the *expected relative residual-workload* of task τ_k on p_z, (ii) $\omega_{k,z}$: the execution time of τ_k on p_z, and (iii) $c_{j,k}^{y,z}$: the actual communication workload required to transmit the output of τ_j (executing on p_y) to τ_k (executing on p_z).

As discussed earlier, the merged task graph G^0 consists of several application instances with each of them having a specific relative deadline. Due to these additional timeliness constraints, determination of ERR values for the sink nodes of application instances within G^0 requires special treatment. For a task node τ_j ($\in V^0$) which is the sink node of the ι^{th} application instance in G^0 ($\tau_j = ST[\iota]$), the ERR value for any processor p_y is determined as: $D^0 - RD[\iota]$. Let τ_j be the sink node of the n^{th} instance of the r^{th} application. Hence, the term $D^0 - RD[\iota]$ equals to $(I^r - n) \times D^r$. That is $D^0 - RD[\iota]$ represents the total duration corresponding to the summation of the end-to-end period lengths of the $I^r - n$ instances of application G^r which appear after instance G_n^r in G^0. For example, the task τ_{12} (refer Fig. 5.2) is the sink task of 1^{st} instance of 2^{nd} application in G^0. $ERR[\tau_{12}, p_y] = 180 - 60 = 120$, where $p_y \in P$ (refer Table 5.3). This ERR value denotes the aggregate estimated *makespan* for the subsequent instances (2^{nd} and 3^{rd} instances) of the 2^{nd} application in G^0.

$\mathcal{R}[\tau_j]$: Given the ERR values, rank $\mathcal{R}$ of any task τ_j is calculated as the task's average ERR value over all processors:

$$\mathcal{R}[\tau_j] = \sum_{y=1}^{|P|} \frac{ERR[\tau_j, p_y]}{|P|} \tag{5.6}$$

The rank of a task associates an appropriate priority value to the task. This priority governs the sequence (captured in list *taskList*) in which the tasks are considered for processor allocation. A task's priority value is intended to serve two objectives. *Primary Objective*: Preserving the precedence constraints associated with the merged DAG by ensuring that every parent of a given task is considered for processor mapping before the task itself. *Secondary Objective*: Tasks having higher residual workloads (workload imposed by still to

Table 5.3: *ERR and $\mathcal{R}$ values corresponding to the merged periodic DAG depicted in Fig. 5.2 and Table 5.1*

Tasks	ERR			$\mathcal{R}$
	p_1	p_2	p_3	
τ_0	149.0	149.0	149.0	149.00
τ_1	46.5	40.2	41.5	42.72
τ_2	25.0	25.8	23.5	24.78
τ_3	25.0	26.2	23.5	24.89
τ_4	18.0	24.0	21.0	21.00
τ_5	10.0	19.0	14.5	14.50
τ_6	10.0	13.0	11.5	11.50
τ_7	0.0	0.0	0.0	0.00
τ_8	139.0	142.0	139.3	140.11
τ_9	128.0	126.0	128.0	127.33
τ_{10}	128.0	126.0	128.3	127.44
τ_{11}	128.0	126.0	127.3	127.11
τ_{12}	120.0	120.0	120.0	120.00
τ_{13}	79.0	82.0	79.3	80.11
τ_{14}	68.0	66.0	68.0	67.33
τ_{15}	68.0	66.0	68.3	67.44
τ_{16}	68.0	66.0	67.3	67.11
τ_{17}	60.0	60.0	60.0	60.00
τ_{18}	19.0	22.0	19.3	20.11
τ_{19}	8.0	6.0	8.0	7.33
τ_{20}	8.0	6.0	8.3	7.44
τ_{21}	8.0	6.0	7.3	7.11
τ_{22}	0.0	0.0	0.0	0.00
τ_{23}	134.7	125.0	128.3	129.33
τ_{24}	114.0	108.0	114.7	112.22
τ_{25}	124.0	116.0	118.7	119.56
τ_{26}	99.0	98.0	96.0	97.67
τ_{27}	102.0	100.0	96.0	99.33
τ_{28}	90.0	90.0	90.0	90.00
τ_{29}	44.7	35.0	38.3	39.33
τ_{30}	24.0	18.0	24.7	22.22
τ_{31}	34.0	26.0	28.7	29.56
τ_{32}	9.0	8.0	6.0	7.67
τ_{33}	12.0	10.0	6.0	9.33
τ_{34}	0.0	0.0	0.0	0.00
τ_{35}	0.0	0.0	0.0	0.00

be allocated tasks) in the merged DAG should be considered earlier for processor mapping.

However, due to the inherent structural relationship between ERR and rank $\mathcal{R}$, there may be situations when a *task's* rank *is not greater than the maximum among the ranks of all its successor tasks*. In this case, one or more successors of a task (say τ_j) may be considered for processor allocation before τ_j, thus violating the *primary objective*. This case is corrected by minimally increasing $\mathcal{R}[\tau_j]$ to: $\max_{\tau_k \in succ(\tau_j)} \mathcal{R}[\tau_k] + \delta$, *where* δ *is a small constant* (refer lines 13-17 of Algorithm 9). We have used $\delta = 0.1$ in our experiments. The ERR values of task τ_j on different processors are also increased in a proportionate fashion.

5.3.1.3 Function DPMRS()

Line 1 of *DPMRS* (refer Algorithm 10) merges the given set of independent application DAGs into a single task graph G^0, using function *periodicMerge()*. Line 2 calculates ERR and rank $\mathcal{R}$ values using function *ERRRank()*. In line 3, the tasks are stored in a vector *taskList* in non-increasing order of their rank values. Line 4 reorders the elements of ST, ET and RD in non-increasing order of their rank ($\mathcal{R}$) values.

Line 5 determines *Minimum Estimated Energy* value ($MEE[\tau_j, p_y]$, $0 \leqslant j \leqslant |V^0|$; $1 \leqslant y \leqslant |P|$), for each task-processor pair. Considering a task τ_j to be executed at the α^{th} minimum frequency $f_{y,\alpha}$ on a processor p_y, this parameter estimates the minimum total dynamic energy that will be dissipated for executing τ_j along with the remaining still-to-be-allocated tasks up to the sink node τ_{exit}. The MEE value of τ_j on processor p_y at current frequency $f_{y,\alpha}$ is calculated recursively by traversing the merged DAG G^0 backwards from τ_{exit} to τ_j, as shown below:

$$MEE[\tau_j, p_y, f_{y,\alpha}] = \begin{cases} \mathcal{E}_d(\tau_{exit}, p_y, f_{y,\alpha}), & \text{if } \tau_j = \tau_{exit} \\ \min\limits_{\tau_k \in succ(\tau_j)} \Big[\min\limits_{p_z \in P, f_{z,\beta} \in F_z} \big\{ MEE[\tau_k, p_z, f_{z,\beta}] + \ \mathcal{E}_d(\tau_j, p_y, f_{y,\alpha}) \big\} \Big], & \text{otherwise} \end{cases} \tag{5.7}$$

The *while* loop (lines 7-30) sequentially generates a static schedule of the tasks in the order as specified by list *taskList*. The schedule determines for each task τ_j: (i) a processor $\rho[j]$ where τ_j is to be executed, (ii) an operating frequency $\varrho[j]$ at which τ_j should be executed on $\rho[j]$, and (iii) τ_j's execution start time $AST[\tau_j]$. In this work, a schedule for τ_j is referred to as *feasible*, if its completion (start) of execution happens on or before (after) the relative deadline (arrival-time) of the application instance in which τ_j belongs. The assignment of $AST[\tau_j]$ (discussed later) takes care that arrival-time constraint related to τ_j's application

instance is satisfied.

Each iteration of the *for* loop (lines 8-29) attempts to schedule, add a new task τ_j to an existing partial schedule (line 9). This task may potentially be mapped to any processor p_y and at any of p_y's available frequencies $f_{y,\alpha}$. Among these alternative choices, selection of the actual $\langle$*processor*, *frequency*$\rangle$-pair is made by judiciously considering *two* parameters: (i) $MEE[\tau_j, p_y, f_{y,\alpha}]$: *Minimum Estimated Energy* with τ_j on p_y at frequency $f_{y,\alpha}$ (refer equation 5.7), and (ii) $EDO[\tau_j, p_y, f_{y,\alpha}]$: *Estimated Deadline Overshoot* with τ_j on p_y at frequency $f_{y,\alpha}$ (refer equation 5.9). The parameter $EDO[]$ tries to estimate the duration by which deadline ($RD[\iota]$ of the application instance say ι to which τ_j belongs) may be overshoot if the remaining schedule is generated with τ_j being allocated on processor p_y at frequency $f_{y,\alpha}$.

Among possible mapping options, the allotment of a task τ_j is first considered within that subset X ($X = \{\langle p_1, f_{1,1}\rangle, \ldots, \langle p_1, f_{1,\beta}\rangle, \langle p_2, f_{2,1}\rangle, \ldots, \langle p_\lambda, f_{\lambda,\beta}\rangle\}$; $(1 \leqslant \lambda \leqslant |P|,\ f_{\lambda,1} \leqslant f_{\lambda,\beta} \leqslant f_{\lambda,\alpha_\lambda})$) of $\langle$*processor*, *frequency*$\rangle$-tuples for which the value $EDO[\tau_j, p_\lambda, f_{\lambda,\beta}]$ is atmost zero (line 13). Now, τ_j is actually allocated to the tuple $\langle p_\lambda, f_{\lambda,\beta}\rangle$ in X for which $MEE[\tau_j, p_\lambda, f_{\lambda,\beta}]$ is minimal (line 15). This step is an attempt to minimize dynamic energy dissipation while not violating stipulated relative deadlines. However, when the set X is empty (it indicates that there is no $\langle$*processor*, *frequency*$\rangle$-pair which the estimated scheduled *makespan* can meet deadline), τ_j is allocated to that $\langle p_\lambda, f_{\lambda,\beta}\rangle$-tuple for which $EDO[\tau_j, p_\lambda, f_{\lambda,\beta}]$ is minimal (line 17). Thus, the $\langle$*processor*, *frequency*$\rangle$-tuple finally mapped to task τ_j is:

$$\langle \rho[j], \varrho[j] \rangle = \begin{cases} \langle p_y, f_{y,\alpha} \rangle \mid MEE[\tau_j, p_y, f_{y,\alpha}] = \min\limits_{\langle p_\lambda, f_{\lambda,\beta}\rangle \in X} MEE[\tau_j, p_\lambda, f_{\lambda,\beta}], & if\ X \neq \phi \\ \langle p_y, f_{y,\alpha} \rangle \mid EDO[\tau_j, p_y, f_{\lambda,\alpha}] = \min\limits_{p_\lambda \in P, f_{\lambda,\beta} \in F_\lambda} EDO[\tau_j, p_\lambda, f_{\lambda,\beta}], & if\ X = \phi \end{cases} \tag{5.8}$$

In a certain iteration of the *for* loop, if all tasks can be scheduled while satisfying the relative deadline of each application instance, the *DPMRS* algorithm successfully terminates with a *feasible energy efficient schedule* (line 32).

Handling Deadline Overshoot: Line 21 within the *inner for* loop (lines 19-29) checks whether the current task τ_j is a sink task, while lines 22-23 determine if there is an actual *Deadline Overshoot* (DO). In case of an actual overshoot ($DO > 0$), *DPMRS* updates the *workload inflation factor* ($\Gamma[\tau_j]$; refer equation 5.10) values of a chosen subset of tasks (lines 26-27) and then attempts to regenerate a fresh schedule from the beginning, in a new iteration of the *while* loop (lines 7-30). This iterative process continues either until a *feasible* energy efficient schedule is generated (*safeFlag* is set) or when there is no possibility

Algorithm 10: DPMRS (G, P)

Input: Application set G and processor set P
Output: A *feasible* schedule which minimizes energy

1 $G^0 = periodicMerge(G, P)$;
2 $[ERR, \mathcal{R}] = ERRRank(G^0, P)$;
3 Sort tasks in non-increasing order of $\mathcal{R}$ and store in $taskList$;
4 Reorder ST, ET, RD in non-increasing order of $\mathcal{R}$;
5 Determine MEE values using equation 5.7;
6 $safeFlag = false$, $exitFlag = false$, $\forall_{\tau_j \in V^0} \Gamma[\tau_j] = 1$;
7 **while** $safeFlag = false$ ***and*** $exitFlag = false$ **do**
8 **for** $c = 1$ *to* $|V^0|$ **do**
9 Let $taskList[c]$ be the j^{th} task τ_j in G^0;
10 **for** *each processor* $p_y \in P$ **do**
11 **for** *each frequency* $f_{y,\alpha} \in F_y$ **do**
12 Compute $EDO[\tau_j, p_y, f_{y,\alpha}]$ using equation 5.9;
13 $X = A$ *set of* $\langle processor, frequency \rangle$ *tuples* $(\langle p_y, f_{y,\alpha} \rangle)$ *for which* $EDO[\tau_j, p_y, f_{y,\alpha}] \leqslant 0$;
14 **if** $X \neq \phi$ **then**
15 Assign τ_j to p_y at frequency $f_{y,\alpha}$, $(\langle p_y, f_{y,\alpha} \rangle \in X)$ for which $MEE[\tau_j, p_y, f_{y,\alpha}]$ is minimal using equation 5.8;
16 Goto line 29; /* *Skip remaining lines within for loop* */
17 Assign τ_j to that $\langle p_y, f_{y,\alpha} \rangle$ for which $EDO[\tau_j, p_y, f_{y,\alpha}]$ is minimal using equation 5.8;
18 /* *ST.length is the total number of instances in* G^0 */
19 **for** $\iota = 1$ *to ST.length* **do**
20 /* τ_j *is the sink task of the* ι^{th} *instance in* G^0 */
21 **if** $\tau_j = ST[\iota]$ **then**
22 $DO = AFT[\tau_j] - RD[\iota]$;
23 **if** $DO \leqslant 0$ **then** Goto line 29; /* *Throttling is not necessary if there is no deadline overshoot* */
24 **for** $q = \iota$ *down to* 1 **do**
25 **if** $\Gamma[ST[q]] \leqslant 2$ **then**
26 Construct a set Υ of tasks: $\Upsilon = \{\tau_i \mid \tau_i \in V^0, AT[q] \leqslant AST[\tau_i] \leqslant AST[ST[q]]\}$;
27 Elevate $\Gamma[\tau_j]$ values $\forall\ \tau_j \in \Upsilon$ (equation 5.10);
28 Goto line 30; /* *Deadline overshoot: Cancel current partial schedule, Retry from start* */
29 **if** $q = 0$ **then** $exitFlag = true$; Goto line 30; /* *No feasible schedule can be generated*/
30 **if** $\tau_j = \tau_{exit}$ **and** $AFT[\tau_{exit}] \leqslant D^0$ **then** $safeFlag = true$;
31 **if** $safeFlag = true$ ***and*** $exitFlag = false$ **then**
32 Output: *Feasible* schedule, compute $\mathcal{E}(G^0)$ (equation 5.5);
33 **else**
34 Output: No *feasible* schedule can be generated;

of generating a *feasible* schedule by proceeding further (*exitFlag* becomes *true*).

It may be observed that, in a scenario where *feasible* schedule for an application instance ι actually exists, ι can miss its deadline due to the following reason: $\langle$*processor*, *frequency*$\rangle$*-choices for tasks in application instance* ι *and other co-running instances have been unduly aggressive towards energy minimization* ($MEE[]$; *refer equation* 5.7) *or too optimistic towards remaining workload estimation.* This aggressiveness/optimism of *DPMRS* is controlled through the parameter EDO (*refer equation* 5.9). $EDO[\tau_j, p_y, f_{y,\alpha}]$ is a throttled estimate of the duration by which deadline ($RD[\iota]$ of the application instance say, ι to which τ_j belongs) may be overshoot if the remaining schedule is generated with τ_j being allocated on processor p_y at frequency $f_{y,\alpha}$. $EDO[\tau_j, p_y, f_{y,\alpha}]$ is defined as:

$$EDO[\tau_j, p_y, f_{y,\alpha}] = (EFT[\tau_j, p_y, f_{y,\alpha}] + ERR[\tau_j, p_y] - ERR[ST[\iota], p_y]) \times \Gamma[\tau_j] - RD[\iota] \tag{5.9}$$

where $EFT[\tau_j, p_y, f_{y,\alpha}]$ denotes *Effective Finish Time* (refer equation 5.12) and the term: $ERR[\tau_j, p_y] - ERR[ST[\iota], p_y]$, denotes the expected residual remaining workload after the completion of τ_j on p_y and before the relative deadline $RD[\iota]$ is reached. At any given iteration of the *while* loop, the term: $(EFT[\tau_j, p_y, f_{y,\alpha}] + ERR[\tau_j, p_y] - ERR[ST[\iota], p_y]) \times \Gamma[\tau_j]$, provides an inflated estimation of the total workload to be completed before $RD[\iota]$. Here, $\Gamma[\tau_j]$ ($1 \leqslant \Gamma[\tau_j] \leqslant 2$; refer equation 5.10) has been used as the *workload inflation factor*. This inflated estimation is considered to have *underestimated* the actual workload, if there is a deadline miss (line 23) in a given iteration. In such a case, the inflation factor for a certain subset of tasks is elevated proportionately with respect to the actual amount of deadline overshoot.

On a deadline miss, we try to generate a fresh schedule which meets $RD[\iota]$ in an iterative fashion (lines 24-28), elevating the Γ-values of a certain subset of tasks in each iteration. First, we try by elevating the Γ-values of the following subset Υ of tasks: $\Upsilon = \{\tau_i \mid \tau_i \in V^0, AT[\iota] \leqslant AST[\tau_i] \leqslant AST[ST[\iota]]\}$. If *DPMRS* fails to generate a partial schedule which meets $RD[\iota]$ even by maximally throttling EDO values of the tasks in Υ (refer line 25 and equation 5.9), a fresh attempt (a new iteration of the for loop; lines 24-28) is made after elevating the Γ-values of a new subset of tasks $\Upsilon = \{\tau_i \mid \tau_i \in V^0, AT[\iota - 1] \leqslant AST[\tau_i] \leqslant AST[ST[\iota - 1]]\}$. This iterative attempt is continued either until a partial schedule which meets $RD[\iota]$ is obtained or the loop (lines 24-28) exits with failure in which case the Algorithm terminates (line 29). Line 27 elevates *workload inflation factors* (Γ-

values) of each task $\tau_j \in \Upsilon$ in a step-wise fashion:

$$\Gamma[\tau_j] = \begin{cases} \Gamma[\tau_j] + \epsilon, & if \ \lceil \frac{DO}{RD[\iota]} \times 100 \rceil = 1 \\ \Gamma[\tau_j] + \epsilon \times \lceil \log_2 \lceil \frac{DO}{RD[\iota]} \times 100 \rceil \rceil, & otherwise \end{cases} \tag{5.10}$$

where ϵ is a small constant. We have used $\epsilon = 0.02$, in our experiments. It may be noted that, as the number of iterations of the *for* loop (lines 24-28) grows, within the overall schedule generation process, the Γ-values of tasks continue to increase and their EDO values get increasingly throttled as a consequence. Thus, the schedule generation process becomes increasingly aggressive towards deadline satisfaction and less aggressive towards energy minimization as time progresses.

$EST[\tau_j, p_y]$: The *Effective Start Time* ($EST[\tau_j, p_y]$) of a task τ_j on processor p_y is calculated using the following four pieces of information: (i) $\rho[\tau_i]$: the processor mapped to each predecessor task τ_i of τ_j, (ii) $AFT[\tau_i]$: the actual finish time of each predecessor τ_i on its mapped processor $\rho[\tau_i]$, (iii) $AT[\iota]$: arrival time of the application instance (say, ι) of which task τ_j is a node ($\tau_j \in \iota$), and (iv) $avl[y]$: the earliest time at which p_y becomes free for task execution. The values of $\rho[\tau_i]$ and $AFT[\tau_i]$ are known during the allocation of τ_j as processor assignment happen in rank order. Therefore, $EST[\tau_j, p_y]$ is determined as:

$$EST[\tau_j, p_y] = \begin{cases} \max\{avl[y], AT[\iota]\}, & if \ (\exists_\iota \mid \tau_j = ET[\iota]) \\ \max\{avl[y], \max\limits_{\tau_i \in pred(\tau_j)} (AFT[\tau_i] + c_{i,j}^{x,y})\}, & otherwise \end{cases} \tag{5.11}$$

where $c_{i,j}^{x,y}$ is the communication time required to transmit the output message from τ_i (executes on p_x) to τ_j (on p_y). Given $EST[\tau_j, p_y]$, *Effective Finish Time* of task τ_j on p_y at frequency $f_{y,\alpha}$ ($EFT[\tau_j, p_y, f_{y,\alpha}]$), is calculated as:

$$EFT[\tau_j, p_y, f_{y,\alpha}] = EST[\tau_j, p_y] + \omega_{j,y} \times \frac{f_{y,\alpha_y}}{f_{y,\alpha}} \tag{5.12}$$

After τ_j is actually mapped on p_y, EST and EFT are referred as AST and AFT. *Actual start time* of task τ_j on the mapped processor $\rho[j]$ is: $AST[\tau_j] = EST[\tau_j, \rho[j]]$. Similarly, *Actual finish time* of a task τ_j on processor $\rho[j]$ at frequency $\varrho[j]$ is: $AFT[\tau_j] = EFT[\tau_j, \rho[j], \varrho[j]]$.

Example Continued: For the example system detailed in Section 5.2, Fig. 5.1 and Fig. 5.2, the priority order of tasks in G^0 is obtained as: $taskList = \langle \tau_0, \tau_8, \tau_{23}, \tau_{10}, \tau_9, \tau_{11}, \tau_{12}, \tau_{25}, \tau_{24}, \tau_{27}, \tau_{26}, \tau_{28}, \tau_{13}, \tau_{15}, \tau_{14}, \tau_{16}, \tau_{17}, \tau_1, \tau_{29}, \tau_{31}, \tau_3, \tau_2, \tau_{30}, \tau_4, \tau_{18}, \tau_5, \tau_6,$

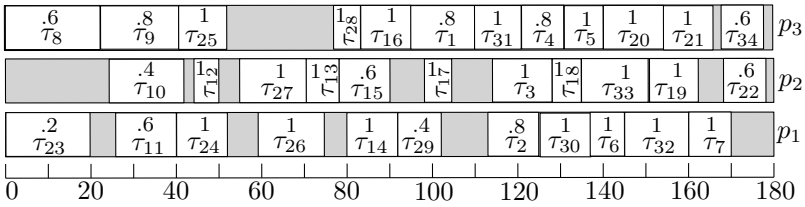


Figure 5.3: *Gantt chart of the schedule generated by DPMRS for the merged DAG in Fig. 5.2:* $SL = 178$, $\mathcal{E}(G^0) = 209.52\ W$.

τ_{33}, τ_{32}, τ_{20}, τ_{19}, τ_{21}, τ_7, τ_{22}, τ_{34}, $\tau_{35}\rangle$. The *DPMRS* algorithm continues for 5 iterations and finally generates a *feasible* schedule (refer Fig. 5.3) with *makespan* 178 and energy $\mathcal{E}(G^0) = 209.52\ W$.

5.3.2 Complexity Analysis

We determine the complexity of *DPMRS* by analysing the overhead of the steps in Algorithm 10. The function *periodicMerge()* in line 1, which merges independent application DAGs into a single DAG G^0, incurs a computational overhead of $O(|V^0| \times |V^0|)$. In lines 2 and 5, the computation of ERR and MEE considers each edge of the merged DAG G^0 exactly once. ERR iterates over $|P|$ processors and MEE iterates over F frequencies (where $F = \sum_{y=1}^{|P|} |F_y|$). Thus, the complexity associated with the construction of ERR is $O(|P| \times (|V^0| + |E^0|))$ and MEE is $O(F \times (|V^0| + |E^0|))$. As the overhead of obtaining the value of rank for a single task is $O(|P|)$ (refer equation 5.6), the total complexity of calculating the list $\mathcal{R}$ for all tasks is $O(|V^0| \times |P|)$. The sorting operation in line 3 requires $O(|V^0| \log |V^0|)$ time. Line 4 requires $O(I \log I)$ time to sort the lists ST, ET and RD of size I (where $I = \sum_{r=1}^{|G|} I^r$).

Task to processor-and-frequency mapping (lines 8-29) happens in a repetitive fashion within the *while* loop (lines 7-30). Observing equations 5.9, 5.10 and the mapping process, it may be inferred that the mapping process (lines 8-29) can return $false$ for both *safeFlag* and *exitFlag*, at most $50 \times I$ times. Thus, the number of iterations of the *while* loop (in lines 7-30) is upper bounded by $50 \times I$. The complexity of *task to processor-and-frequency mapping process* is dominated by the computational overhead of computing $EDO[\tau_j, p_y, f_{y,\alpha}]$ (refer equation 5.9; in line 12) for all tasks on all $\langle$*processor*, *frequency*$\rangle$-pairs within the nested *for* loops. The overhead of computing $EDO[\tau_j, p_y, f_{y,\alpha}]$ is further dependent on the overhead of $EFT[\tau_j, p_y, f_{y,\alpha}]/EST[\tau_j, p_y]$. It may be noted that computation of EST for any given task-processor pair requires constant time calculations over all predecessors of a task and hence,

incurs an overhead of $O(\#predecessors)$. However, the total number of predecessors in the task graph is equal to the total number of edges. Therefore, the amortized overhead for determining $EST[\tau_j, p_y]$ (and also $EFT[\tau_j, p_y, f_{y,\alpha}]$, $EDO[\tau_j, p_y, f_{y,\alpha}]$) becomes $O(|E^0|/|V^0|)$ Given this overhead, the total complexity incurred in computing $EDO[\tau_j, p_y, f_{y,\alpha}]$ for all tasks on all $\langle processor, frequency \rangle$-pairs become $O(F \times |V^0| \times |E^0|/|V^0|) = O(F \times |E^0|)$. Complexity of the remaining lines of *DPMRS* are smaller then the overhead of computing *EDO*. Therefore, the overall complexity of the *DPMRS* Algorithm is: $O(|V^0| \times |V^0| + |P| \times (|V^0| + |E^0|) + F \times (|V^0| + |E^0|) + |V^0| \times |P| + |V^0| \log |V^0| + I \log I + 50 \times I \times F \times |E^0|) = O(I \times F \times |E^0|)$. It may be noted that the value of I in the above expression is governed by individual period values and size of the hyperperiod H. For a given set of arbitrary period values, it is possible that H (and consequently I) becomes very large (say, when the period values are relatively prime). A large value of I in turn, will have the effect of making the complexity of *DPMRS* vary high. However in general, application periods are adjustable with in certain bounds [17, 21]. By choosing appropriate period values, reasonably small values of H as well as I can be obtained [14, 66], so that the complexity of *DPMRS* can be contained within acceptable limits.

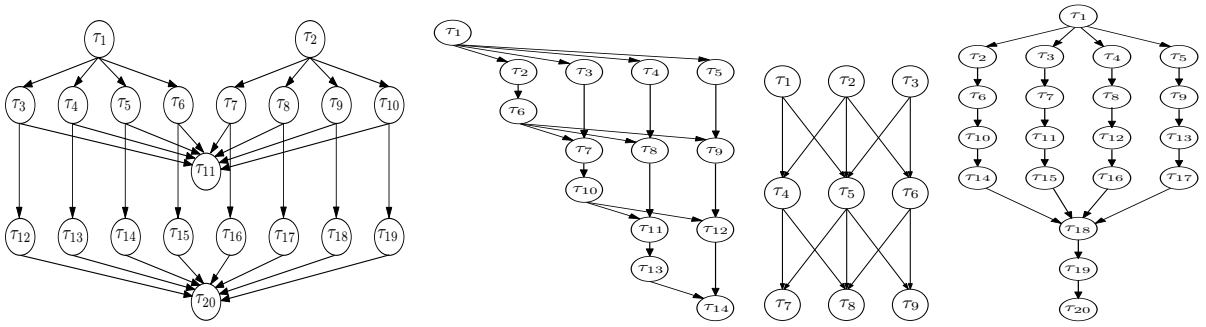
5.4 Experiments and Results

The performance of the proposed real-time schedulers *DPMRS* and *NDPMRS* have been experimentally assessed through an extensive set of simulation-based experiments. We first describe the experimental setup, the performance metrics and the comparison with related works in the following subsections, followed by detailed experimental results. Similar to Chapter 4, a simple real-platform implementation of the proposed work has been carried out, in addition to the simulation-based experiments discussed above.

5.4.1 Experimental Setup

The experiments have been conducted using four real-world benchmark task graphs: CyberShake [39], Gaussian Elimination [85], Stencil [68] and Epigenomics [67]. Fig. 5.4 depicts the task graph structures of these four benchmarks. The benchmarks used include of CPU-intensive (Epigenomics), I/O-intensive (Gaussian Elimination, Stencil) applications, as well as applications that have considerable memory requirements (CyberShake). Such diverse

applications help performance evaluation of the proposed strategy under various applications and system scenarios.



(a) *CyberShake* **(b)** *Gaussian Elimination* **(c)** *Stencil* **(d)** *Epigenomics*

Figure 5.4: *Benchmark DAGs.*

CyberShake (Fig. 5.4a) is an earthquake hazard characterization workflow used by the Southern California Earthquake Center. This workflow has five types of tasks: *ExtractSGT* (τ_1-τ_2), *SeismogramSynthesis* (τ_3-τ_{10}), *ZipSies* (τ_{11}), *PeakValCalcOkaya* (τ_{12}-τ_{19}) and *ZipPSA* (τ_{20}). The size of the CyberShake graph is influenced by the number of *ExtractSGT* and *SeismogramSynthesis* tasks. For simplicity, we fix the number of *ExtractSGT* tasks to 2 and divide *SeismogramSynthesis* tasks near-equally among the *ExtractSGT* tasks. If the number of *SeismogramSynthesis* tasks is υ, then the total number of task nodes and edges is equal to $(2 \times \upsilon + 4)$ and $4 \times \upsilon$, respectively.

Gaussian elimination (Fig. 5.4b) also known as row reduction, is an algorithm used to solve linear equations in linear algebra. The structure of Gaussian Elimination is governed by the size of its input parameter called *matrix-size* (ν). Given ν, the total number of tasks is equal to $((\nu^2 + \nu - 2)/2)$ and edges equal to $(\nu^2 - \nu - 1)$.

Stencil (Fig. 5.4c) task graph is based on the one-dimensional stencil computation method used to solve partial differential equations in one spatial dimension. Stencil equation solvers can be influenced by the number of levels (λ) and the number of tasks per level (η). The number of task nodes and edges in a Stencil task graph is $(\lambda \times \eta)$ and $[(\lambda - 1) \times \{3(\eta - 2) + 4\}]$, respectively. In our experiments, we have considered $\lambda = \eta$.

Epigenomics (Fig. 5.4d) is used to automate various genome sequence processing operations. The structure of Epigenomics is characterized by the input parameter *parallel branches* ϑ. Given ϑ of a task graph, the total number of tasks and edges is given by $(4\vartheta + 4)$ and $(5\vartheta + 2)$, respectively.

Multiple source (sink) nodes exist in the CyberShake and Stencil task graphs. We add a dummy source (sink) node, and all the actual source (sink) nodes are set as the immediate successor (predecessor) nodes of the dummy source (sink) node.

Data Generation Framework: Our experimental analysis has been carried out using randomly generated data sets obtained by carefully varying an exhaustive set of input parameters.

1. **Number of DAGs**: $|G| = \{2, 4, 6, 8, 10\}$.

2. **Number of tasks in each DAG**: For CyberShake, *SeismogramSynthesis* $\upsilon = \{8, 9, 10\}$. With these values of υ, three types of CyberShake DAGs ($|V|$, $|E|$): ($\langle 20, 32\rangle$, $\langle 22, 36\rangle$, $\langle 24, 40\rangle$) are generated. Similarly for Gaussian Elimination, *matrix-size* $\nu = \{5, 6, 7\}$. With these values ν, three types of Gaussian Elimination DAGs ($|V|$, $|E|$): ($\langle 14, 19\rangle$, $\langle 20, 29\rangle$, $\langle 27, 41\rangle$) are generated. For Stencil, we have generated three types of task graphs having $\langle |V|, |E|, \lambda\rangle$ as $\langle 16, 30, 4\rangle$, $\langle 25, 52, 5\rangle$ and $\langle 36, 80, 6\rangle$. Similarly for Epigenomics, task graphs having $\langle |V|, |E|, \vartheta\rangle$ as $\langle 20, 22, 4\rangle$, $\langle 24, 27, 5\rangle$ and $\langle 28, 32, 6\rangle$ are generated.

3. **Number of processors**: $|P| = \{4, 8, 16, 32\}$.

4. **Task execution times**: The execution times of all tasks (of any DAG) on each heterogeneous processor, are generated using three phases: (i) the average execution time overall tasks of the DAG is denoted as $\overline{\omega_{DAG}}$. In this work, we used the values of $\overline{\omega_{DAG}} = \{25, 50, 75, 100\}$. (ii) Given $\overline{\omega_{DAG}}$, the average execution time ($\overline{\omega_j}$) of a task (say, τ_j), over all processors is determined. Values of $\overline{\omega_j}$ of the tasks are generated from normal distributions having mean $\mu = \overline{\omega_{DAG}}$ and distinct values of standard deviation σ ($= \{5, 10\}$). (iii) Finally, the *WCET* of τ_j on each processor ($\omega_{j,y}$) is determined. Values of $\omega_{j,y}$ of a task τ_j are obtained from a normal distribution having mean $\mu = \overline{\omega_j}$ and standard deviation $\sigma = (\overline{\omega_j} \times \beta)$; where β denotes the heterogeneity factor that defines the degree of skewness among the task's execution times on various heterogeneous processors. Here, we used the values of $\beta = \{0.1, 0.25, 0.5, 0.75, 1\}$. The obtained $\omega_{j,y}$ values are then appropriately scaled so that $\sum_{j=1}^{|V|} \sum_{y=1}^{|P|} \omega_{j,y} = |V| \times |P| \times \overline{\omega_{DAG}}$.

5. **Data communication workload**: The ratio of the time overhead related to data transmission and task execution is referred to as *Communication-to-Computation Ratio* (*CCR*). Values of *CCR* used in this work: $CCR = \{0.1, 0.25, 0.5, 0.75, 1\}$. Given *CCR*, the average communication workload $\overline{c_{DAG}}$ is determined as $\overline{c_{DAG}} = CCR \times \overline{\omega_{DAG}}$. The average inter-task message size ($\overline{data_{DAG}}$; in *bytes*) for a DAG is computed as: $\overline{data_{DAG}} = \overline{c_{DAG}} \times \overline{B}$,

where $\overline{B}$ ($= \frac{1}{|P|\times(|P|-1)/2} \sum_{x=1}^{|P|} \sum_{y=1}^{x-1} b_{x,y}$) is the average communication bandwidth of the considered computing platform. Experiments have been conducted for two different values of $\overline{B}$ ($= \{3\ Mbps, 5\ Mbps\}$). The actual bandwidth of the communication link between processors p_x and p_y is referred as $b_{x,y}$. The values of $b_{x,y}$ are generated from a normal distribution having mean $\mu = \overline{B}$ and different standard deviation values σ ($= 0.2 \times \overline{B}$). The $b_{x,y}$ values are then scaled so that $\sum_{x=1}^{|P|} \sum_{y=1}^{x-1} b_{x,y} = |P| \times (|P|-1)/2 \times \overline{B}$. Similarly, the size of the output message ($data_{i,j}$) for each edge (τ_i, τ_j) in the DAG is generated from a normal distribution having mean $\mu = \overline{data_{DAG}}$ and various values of standard deviation σ ($= 0.2 \times \overline{data_{DAG}}$). These $data_{i,j}$ values are then appropriately scaled in order to make $\sum_{i=1}^{|V|} \sum_{j=1}^{|V|} data_{i,j} = |E| \times \overline{data_{DAG}}$.

6. **Power parameters**: Similar to [99], the power values of the processors are generated randomly with the ranges as follows: $pow_y^s = 0.01$, $0.03 \leqslant pow_y^{ind} \leqslant 0.07$, $0.08 \leqslant C_y^{ef} \leqslant 1.2$, $2.5 \leqslant m_y \leqslant 3.0$, and $\forall_{p_y \in P} f_{y,\alpha_y} = 1\ GHz$. The processors' frequencies are discrete, and the precision is $0.2\ GHz$. The values of these parameters reflect the characteristics of some high-performance embedded processors, such as *ARM Cortex-A9* and *Intel Mobile Pentium III*.

7. **Relative deadline of a DAG**: Determination of the relative deadline of a DAG consists of two phases. For each DAG say G^r, first *PEFT* [5] is used to determine the *makespan* SL_r considering all processors to be assigned to $G^r \in G^0$ (i.e., G^r runs stand-alone) and assuming all processors to be always executing at their maximum frequencies. After obtaining SL_r of all the DAGs, the relative deadline of G^r is computed as: $D^r = roundOff(T_{SL} \times \{t_1 + \frac{SL_r - minSL_r}{maxSL_r - minSL_r} \times t_2\})$, where $T_{SL} = \sum_{r=1}^{|G|} SL_r$, $minSL_r$ and $maxSL_r$ are the minimum and maximum among all SL_r values, respectively. In the above equation, t_1 and t_2 are two design constraints, whose values are set to 0.4 and 0.2. The *roundOff* is used to round-off its argument (say, Y) to the lowest multiple of 500 which is greater than or equal to Y.

Simulation Framework: The simulation framework is written in C and is executed on a system having the following configuration: (i) Intel® Core™ i7-8550U CPU @ 1.80GHz ×8, (ii) 8 GiB Memory, and (iii) Ubuntu 18.04.4 LTS OS (64-bit).

5.4.2 Performance Metrics

The performance of the proposed schedulers has been analysed using two different metrics:

1. **Normalized Energy-dissipation** (NE): Given a set of DAGs, *Normalized Energy-dissipation (NE; in percentage)* is defined as:

$$NE = \frac{\mathcal{E}_{ACT}}{\mathcal{E}_{MAX}} \times 100$$

where $\mathcal{E}_{ACT}$ and $\mathcal{E}_{MAX}$ represent the actual and maximum energy consumed in the execution of all instances of all DAGs over the length of the hyperperiod. In fact, $\mathcal{E}_{ACT}$ is the amount of energy consumed by the processors to execute tasks as per the schedule generated by a scheduling algorithm. $\mathcal{E}_{MAX}$ denotes the total amount of energy that may be dissipated by continuously operating on all processors at their maximum frequencies (i.e., highest level) over the entire duration of the hyperperiod.

2. **Normalized Running Time** (NRT): This metric measures the ratio of the total run-time to the total number of task nodes in G^0. NRT is defined as:

$$NRT = \frac{total\ run\text{-}time}{\sum_{r=1}^{|G|} I^r \times |V^r|}$$

3. **Deadline Miss Rate** (DMR): This parameter measures the percentages of input test cases for which at least one application instance within the hyperperiod misses its deadline.

5.4.3 Comparison with Related Works

To the best of our knowledge, there does not exist any work in literature that target *computation communication co-scheduling of a set of (more than one) real-time periodic DAGs in a distributed heterogeneous environment.* However, as discussed in the related work section 2.5.3 of Chapter 2, there are a few works which deal with communication-aware scheduling of *single* real-time DAG over a heterogeneous platform. Two such works namely *GDES* [99] and *NDES* [99], which attempt to minimize system-level energy dissipation in DVFS and non-DVFS enabled systems respectively, have been found to relate most closely to the system model considered in this chapter. Hence, we have chosen these two works as candidate state-of-the-art techniques against whom *DPMRS* can be compared. In order to match the system model used in the current work, *GDES* and *NDES* have been adapted as follows. As *GDES* and *NDES* work for single DAG applications, the available set U of processors has been partitioned into $|G|$ disjoint subsets of size $|U|/|G|$ each. Each such

processor subset has then been randomly allocated to a distinct DAG application. Then, *GDES* and *NDES* have been separately applied to derive a schedule for each partitioned subsystem obtained in the previous step. Finally, energy consumption for the *GDES*/*NDES* schedulers has been determined by calculating the total energy consumed by all application schedules over the interval D^0. Since the applications are allowed to run concurrently, our strategy gives all of them approximately equal opportunities to execute at the same time on the given platform. Obviously, with this fully partitioned approach, results for *GDES* and *NDES* cannot be obtained when $|G| > |U|$. Note that *GDES* is not an independent scheduler, it requires a schedule as an input. In our experiments, we have considered a version of *GDES* which takes *NDES* as its input. We have referred to this algorithm as "*NDES&GDES*". Additionally, we have analyzed the performance of *DPMRS* against a baseline DVFS-oblivious version of *DPMRS*, which we call *NDPMRS*.

The NDPMRS Algorithm: On systems which are not DVFS-enabled, each processor can run only at a single frequency. *DPMRS* can be easily adapted and applied on such systems. We refer to this Non-DVFS version of *DPMRS* as *NDPMRS*. We use the same example system (refer Section 5.2 and Fig. 5.2; with the restriction that each processor can run only at its maximum frequency) and apply it as input to *NDPMRS*. The *NDPMRS* algorithm continues for 9 iterations and finally generates a *feasible* schedule (refer Fig. 5.5) with *makespan* 179 and energy $E(G^0) = 224.62\ W$. It can be noted that, *DPMRS* is able to fetch higher dynamic energy savings of 15.1 W as it is able to opportunistically use slack processor capacities in order to reduce processor operating frequencies.

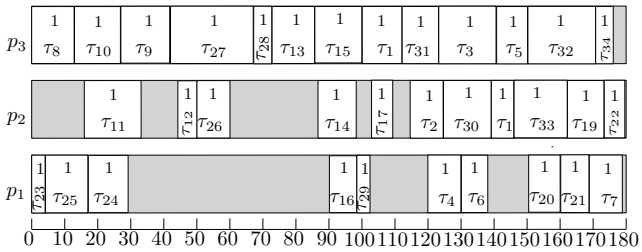


Figure 5.5: *Gantt chart of the schedule generated by NDPMRS for the merged DAG in Fig. 5.2:* $SL = 179$, $E(G^0) = 224.62\ W$.

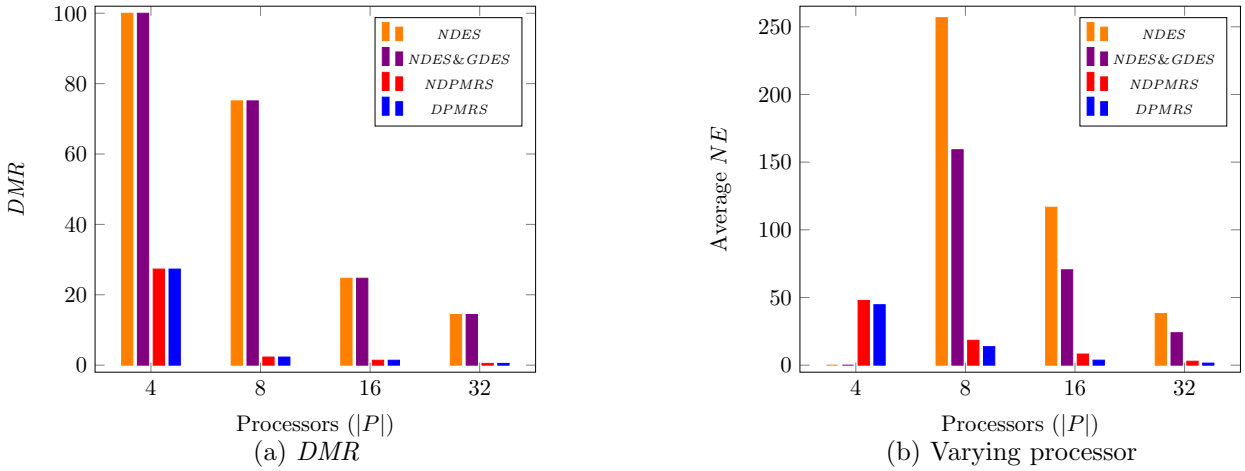
5.4.4 Performance Results

Three classes of experiments have been conducted to evaluate the performance of *DPMRS*. *Experiment*-1 compares the energy efficiencies and deadline miss rates of *DPMRS*, with the state-of-art works *GDES* and *NDES* as well as the baseline strategy *NDPMRS*. In *experiment*-2, the proposed algorithms are evaluated against variation in two separate parameters *heterogeneity* and *CCR*. *Experiment*-3 measures the energy efficiency and run-time related performance, as the number of DAGs is varied from 2 to 10. Each data point in experiments 2 and 3 is obtained as the average over 250 runs of a specific scheduler on different DAG instances produced by assigning specific values to a carefully chosen set of parameters. Given a set of values for task execution times and average data communication times, Figs. 5.7 and 5.8 generate results for two different values of workloads. Here, *workload* has been measured as: $WL = \sum_{j=1}^{|V^0|} \sum_{y=1}^{|P|} \omega_{j,y} + \sum_{i=1}^{|V^0|} \sum_{j=1}^{|V^0|} data_{i,j}/\bar{B}$. For one set of plots (*DPMRS-WL1* and *NDPMRS-WL1*) the parameter values are used as it is, to generate the input workloads. For the other set of plots (*DPMRS-WL2* and *NDPMRS-WL2*) each parameter value has been reduced by 25%, to generate the input workload. These two distinct values of workloads have been used to show the efficiency of the algorithms in terms of their ability to achieve additional energy savings when the input workload reduces by 25% for a given set of parameter values.

5.4.4.1 Experiment-1: Effect of variation in #processors

This experiment measures the *Normalized Energy-dissipation* (NE) and *Deadline Miss Rates* (DMR) of *DPMRS*, *NDPMRS*, *NDES&GDES* and *NDES*, as the number of processors $|P|$ varies from 4 to 32. The parameters β and CCR are set to 0.75 and 0.5, respectively. All plots have been generated using four application instances (a *CyberShake* with 20 tasks, a *Gaussian Elimination* with 14 tasks, a *Stencil* with 25 tasks, an *Epigenomics* with 25 tasks). Values of the hyperperiod are obtained from the range [500, 3000]. Obtained results are presented in Fig. 5.6 considering 1000 input test cases for each scenario (which consists of a distinct number of processors).

Given the 4 benchmark applications in a scenario consisting of $|U|$ processors, a separate *NDES&GDES* (*NDES*) scheduler is used for each disjoint partition consisting of a DAG application to be executed over $|U|/4$ processors. In comparison, *DPMRS* (*NDPMRS*) is a global algorithm which take as input a merged task graph consisting of all DAG appli-



(a) *DMR* (b) Varying processor

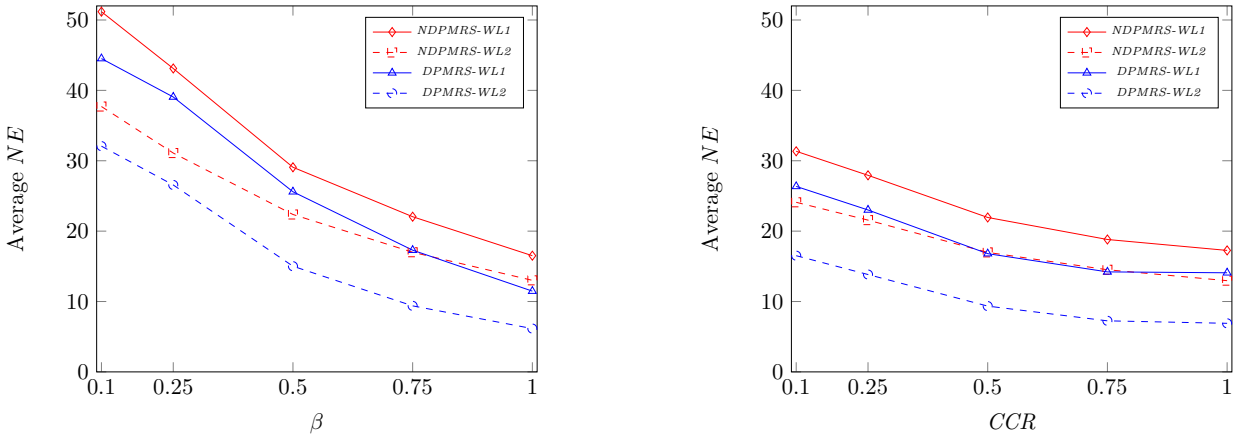
Figure 5.6: *(a) Deadline Miss Rates (DMR; in %) and (b) Normalized Energy-dissipation (NE; in %) w.r.t. varying processors.*

cations to be executed in a hyperperiod, and produces as output an execution schedule of the applications on the consolidated platform of $|U|$ heterogeneous processors. Although, it works with a single merged task graph, a feasible *DPMRS* (*NDPMRS*) schedule is able to guarantee that no application instance within the hyperperiod ever misses its deadline. Compared to *NDES&GDES* (*NDES*), the global nature of *DPMRS* (*NDPMRS*) allows it to harness significantly improved processor/communication resource sharing among different DAGs, in addition to better exploitation of task-processor affinities in the heterogeneous environment. Due to such efficient sharing and affinity awareness, *DPMRS* (*NDPMRS*) is able to achieve considerably lower deadline miss rates and energy dissipation, compared to *NDES&GDES* (*NDES*). This is reflected in the results presented in Fig. 5.6a and Fig. 5.6b. For example, in the scenario consisting of 4 (8) processors in Fig. 5.6a, while *deadline miss rate* suffered by *DPMRS* and *NDPMRS* is only 27.3% (2.3%), *NDES&GDES* and *NDES* suffers 100% (75.1%) miss rates. Similarly, for the scenario with $|U| = 16$ processors in Fig. 5.6b, it may be observed that the *Normalized Energy-dissipation* suffered by *DPMRS* and *NDPMRS* are 3.69 W (8.2 W). In comparison, *NDES&GDES* and *NDES* suffer significantly higher dissipation − 70.45 W and 116.68 W, respectively. It may be noted that for $|U| = 4$, as *NDES&GDES* and *NDES* suffer 100% *deadline misses*, their average NE values have been shown as *blank*, in Fig. 5.6b. In the next two experiments, we discuss results for the average NE values of *DPMRS* and *NDPMRS* on 8 processor systems, with respect to variations in *heterogeneity*, *CCR* and *#DAGs*. Results of *NDES&GDES* and

NDES have not been included further because they are seen to exhibit very heavy deadline misses (deadline misses in 75.1% of the 1000 test cases considered on 8 processor systems) as a result of their partitioned nature.

5.4.4.2 Experiment-2: Effect of variation in CCR and heterogeneity

Using the same experimental setup as employed for *experiment*-1, here we measure the *Normalized Energy-dissipation* (NE) of *DPMRS* and *NDPMRS* for varying heterogeneity (β) and Communication-to-Computation Ratios (CCR). Obtained results are presented in Fig. 5.7a and Fig. 5.7b. We have not presented the results of *NDES&GDES* and *NDES* because, similar to the results trends in experiment-1, their performance is seen to be significantly poorer in all cases.



(a) Varying heterogeneity

(b) Varying CCR

Figure 5.7: *Normalized Energy-dissipation (NE) w.r.t. varying heterogeneity and CCR.*

Fig. 5.7a shows average NE values as the degree of heterogeneity is increased from 0.1 to 1. Values of the parameters $|P|$ and CCR have been fixed at 8 and 0.5. As the average task-to-processor affinity decreases with increase in the degree of heterogeneity, the performance of both the algorithms *DPMRS-WL1* and *NDPMRS-WL1* in terms of achieved NE becomes lower when heterogeneity is higher. It may be observed that for any degree of heterogeneity, *DPMRS-WL2* (*NDPMRS-WL2*) performs better compared to *DPMRS-WL1* (*NDPMRS-WL1*).

Fig. 5.7b shows the variation in average NE values as CCR varied between 0.1 and 1. Parameters $|P|$ and β are fixed at 8 and 0.75. A higher CCR value implies lower execu-

tion demand for each task on any processor at maximum frequency. Such lower execution demands, in turn, naturally enhance the possibility of processor frequency reduction. Consequently, this leads to lowering in the obtained NE values. For example, the average normalized energy values of *DPMRS-WL1* are 26.36 W and 16.80 W for $CCR = 0.1$ and $CCR = 0.5$.

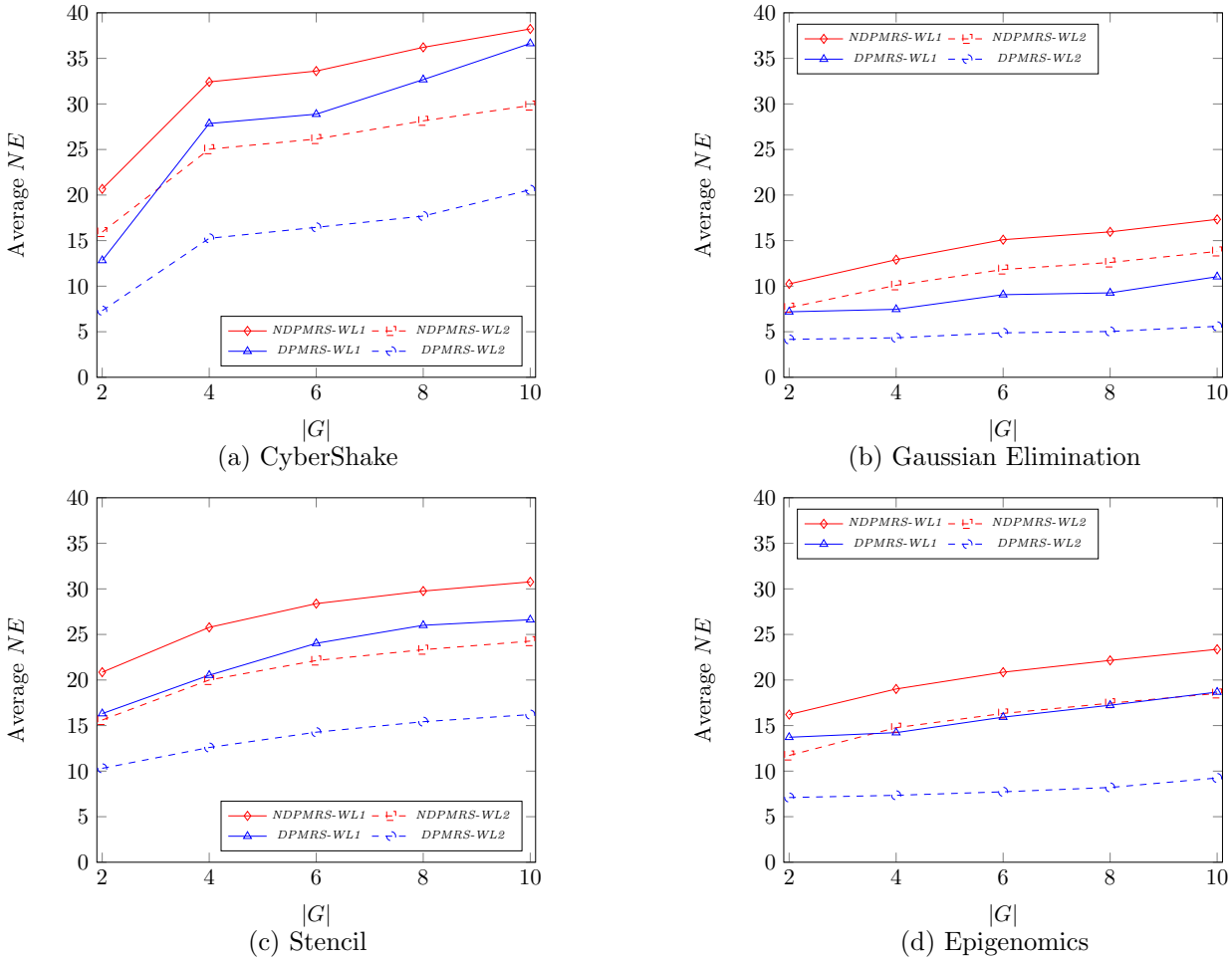


Figure 5.8: *Normalized Energy-dissipation (NE; in percentage) w.r.t. varying #DAGs for CyberShake, Gaussian Elimination, Stencil and Epigenomics.*

5.4.4.3 Experiment-3: Effect of variation in #tasks

This experiment measures the *Normalized Energy-dissipation* (NE) and *Normalized Running Time* (NRT) of *DPMRS* and *NDPMRS* as the number of DAGs $|G|$ varies from 2

to 10, for each benchmark task graph. In this, the parameters $|P|$, β and CCR have been fixed at 8, 0.75 and 0.5 respectively, and the hyperperiods are limited within the range [500, 3000]. Obtained NE and NRT results for the benchmarks CyberShake, Gaussian Elimination, Stencil and Epigenomics, are presented in Fig. 5.8 and Table 5.4, respectively.

Fig. 5.8 depicts the obtained average NE values for different number of DAGs. It can be observed that for any given number of DAGs, the average normalized energy-dissipation value of *DPMRS-WL1* is lower than *NDPMRS-WL1*. As an example for $|G| = 6$, *DPMRS-WL1* reduces energy consumption by 5%, 6%, 4% and 5%, compared with *NDPMRS-WL1* for CyberShake (Fig. 5.8a), Gaussian Elimination (Fig. 5.8b), Stencil (Fig. 5.8c) and Epigenomics (Fig. 5.8d), respectively. As is obvious, NE values increase as workload becomes higher with increase in the number of DAGs. For all the benchmarks, due to lower relatives workloads *DPMRS-WL2* (*NDPMRS-WL2*) performs better than to *DPMRS-WL1* (*NDPMRS-WL1*) in terms of achieved energy consumption rates.

Table 5.4 lists the average NRT values for different number of DAGs. It is observed that the normalized running times of both the algorithms *DPMRS* and *NDPMRS* strictly increase with the number of DAGs. As is obvious, the NRT values of *DPMRS* is higher than *NDPMRS*. For all the benchmarks, the NRT values of *DPMRS* is upper bounded by $\approx 0.2\ ms$.

Table 5.4: *Normalized Running Time (NRT; in ms) for varying #DAGs*

	CyberShake					Gaussian Elimination				
#DAGs	2	4	6	8	10	2	4	6	8	10
DPMRS	0.10	0.12	0.13	0.15	0.17	0.09	0.10	0.11	0.12	0.13
NDPMRS	0.006	0.008	0.009	0.011	0.014	0.007	0.008	0.009	0.010	0.011
	Stencil					Epigenomics				
#DAGs	2	4	6	8	10	2	4	6	8	10
DPMRS	0.15	0.16	0.18	0.19	0.20	0.08	0.09	0.10	0.11	0.12
NDPMRS	0.009	0.010	0.012	0.014	0.016	0.006	0.007	0.008	0.010	0.012

5.5 Case Study: Automotive Control System

To examine the practical applicability of both the proposed schedulers *NDPMRS* and *DPMRS* to real-world designs, we now conduct a case study using three automotive control applications namely, *Adaptive Cruise Control* (ACC), *Traction Control* (TC) and *Electric*

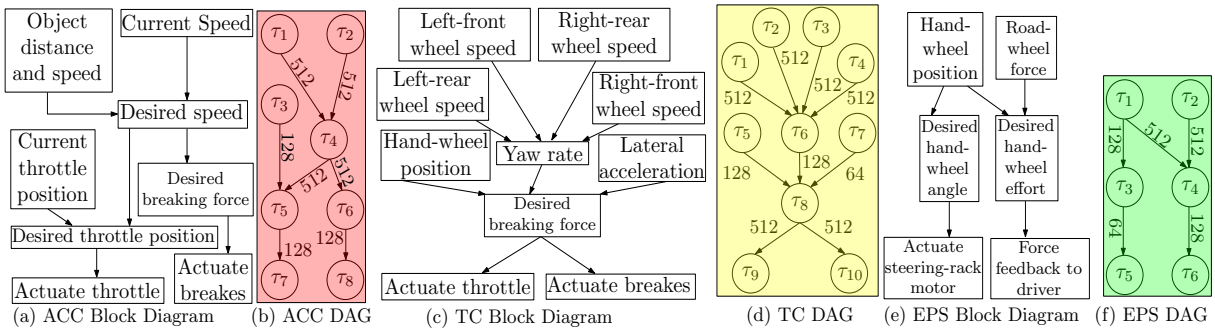


Figure 5.9: *Automotive Control System (ACS): (a) Adaptive Cruise Control (ACC) Block Diagram; (b) ACC DAG; (c) Traction Control (TC) Block Diagram; (d) TC DAG; (e) Electric Power Steering (EPS) Block Diagram; (f) EPS DAG.*

Power Steering (*EPS*). The block structures of ACC, TC and EPS are detailed in Figs. 5.9a, 5.9c and 5.9e, respectively (adapted from [41]). Figs. 5.9b, 5.9d and 5.9f display their task graph representations. The ACC application helps in maintaining a safe distance within two cars automatically. On the other hand, EPS provides steering assistance to a driver with the help of an electric motor. The TC application helps to improve a car's stability when road conditions are slippery. These automotive applications have stipulated end-to-end deadlines to allow correct and timely interaction between components such as sensors, processors and actuators. Let us assume the deadlines to be 2000 ms, 2000 ms and 1000 ms for ACC, TC and EPS.

Table 5.5: *ACS: Task's execution times on three heterogeneous processors*

	ACC								TC										EPS					
	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
p_1	300	150	175	300	250	200	150	200	200	180	200	155	150	120	180	332	150	199	273	300	67	48	65	91
p_2	120	160	90	157	124	347	160	180	215	200	150	98	88	300	95	400	146	225	149	120	240	150	289	78
p_3	100	274	244	261	300	167	279	173	175	274	100	48	74	243	206	276	154	200	150	175	300	97	150	100

The automotive applications are merged using Algorithm 8. The merged task graph G^0 which has an end-to-end deadline of 2000 consists of 4 instances: 1 from ACC $\langle \tau_2 - \tau_9 \rangle$, 1 from TC $\langle \tau_{12} - \tau_{21} \rangle$ and 2 from EPS $\langle \tau_{24} - \tau_{29}, \tau_{32} - \tau_{37} \rangle$ applications. Thus, G^0 consists of 40 task nodes (including dummy nodes $\langle 0, 1, 10, 11, 22, 23, 30, 31, 38, 39 \rangle$). We assumed a distributed platform having three heterogeneous processors u_1, u_2 and u_3. The bandwidths of the communication links between each pair of processing devices are: $b_{1,2} = b_{2,1} = 250\ KB/s$, $b_{2,3} = b_{3,2} = 500\ KB/s$, $b_{3,1} = b_{1,3} = 1\ MB/s$, and $b_{1,1} = b_{2,2} = b_{3,3} = \infty$. Tables 5.2 and 5.5 depict the power parameters and execution times associated with

each task on the three heterogeneous processors. Each edge $e_{i,j}$ is labelled with a positive $data_{i,j}$ (in *bytes* value; for example, in ACC $data_{1,4} = 512$ *bytes*). We have employed *DPMRS*, *NDPMRS*, *NDES&GDES* and *NDES* to generate four separate schedules. Gantt chart of the schedules *DPMRS*, *NDPMRS*, *NDES&GDES* and *NDES*, are presented in Fig. 5.10. It may be observed that, *DPMRS* is able to deliver a lower energy dissipation rate ($E(G^0) = 2391.42\ W$) compared to *NDPMRS* ($E(G^0) = 2536.66\ W$), *NDES&GDES* ($E(G^0) = 3568.67\ W$) and *NDES* ($E(G^0) = 3853.77\ W$).

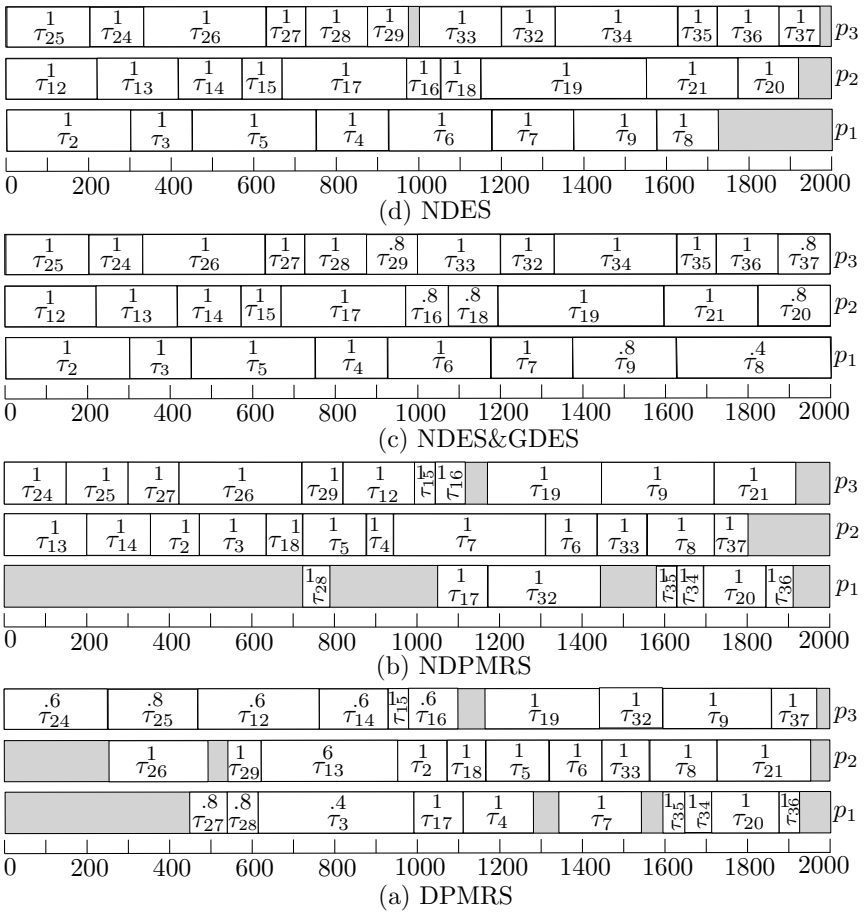


Figure 5.10: *Gantt charts depicting the schedules: (a) DPMRS (SL = 1967, $E(G^0) = 2391.42\ W$), (b) NDPMRS (SL = 1921, $E(G^0) = 2536.66\ W$), (c) NDES&GDES (SL = 2000, $E(G^0) = 3568.67\ W$), and (d) NDES (SL = 1944, $E(G^0) = 3853.77\ W$), for the ACS task graphs in Fig. 5.9.*

5.6 Summary

This chapter proposed a static list-based real-time scheduling algorithm called *DPMRS* for multiple independent periodic applications on a distributed heterogeneous platform. The overall objective of *DPMRS* is to minimize total dynamic energy dissipation associated with the execution of multiple independent periodic DAGs using a DVFS approach. For the same problem, this work also presented another scheduler named *NDPMRS* (an adaption to *DPMRS*), targeting the platforms which are not DVFS-enabled. Experimental analysis employing four benchmark task graphs acknowledges that *DPMRS* performs appreciably over extensive sets of test scenarios, pointing to the practical effectiveness of the scheme. As is obvious, *DPMRS* is able to fetch higher dynamic energy savings compared to *NDPMRS*. From the experimental results, we can also conclude that compared to the state-of-the-art energy-aware single DAG scheduler *NDES&GDES* (*NDES*), the global nature of *DPMRS* (*NDPMRS*) allows it to harness significantly improved processor/communication resource sharing among different DAGs, in addition to better exploitation of task-processor affinities in the heterogeneous environment. Due to such efficient sharing and affinity awareness, *DPMRS* (*NDPMRS*) is able to achieve considerably lower deadline miss rates and energy dissipation, compared to *NDES&GDES* (*NDES*). Finally, the practical adaptability of *DPMRS* and *NDPMRS* is exhibited through a case study with an automotive control system.

In the next chapter, we present a real-time policy called *SHIELD*, whose objective is to maximize total security utility for a given DAG-structured application having known minimum security strength specifications for its messages.

5. DPMRS: AN ENERGY-AWARE REAL-TIME SCHEDULING OF MULTIPLE PERIODIC DAGS ON HETEROGENEOUS SYSTEMS

Chapter 6

SHIELD: Security-aware Scheduling for Real-time DAGs on Heterogeneous Systems

6.1 Introduction

The scheduling strategies proposed in the first three contributory chapters (refer Chapters 3, 4 and 5) do not focus towards ensuring the security needs of networked RT-CPS applications. However, data communication between dependent task nodes running on different processing elements is often realized through message transmission over a public network and hence such transmissions are susceptible to multiple security threats such as *snooping*, *alteration* and *spoofing*. *Snooping* denotes the interception of information being transmitted by unauthorized entities. *Alteration* refers to the change of information bits in transmitted messages by unauthorized users so that a modified message is obtained at the receiving end. The third type of security threat, *spoofing*, refers to an entity impersonating another entity. This may lead to the receipt of malicious messages from the impersonated entity, who is misinterpreted as a valid sender. Several alternative security protocols having varying security strengths and associated implementation overheads are available in the market for incorporating *confidentiality* [97] (protection against unauthorized access and misuse), *integrity* [11] (recognizing message bit modification at receiver end) and *authentication* [23] (identifying messages from unauthorized senders) on the transmitted messages. While message size and conceptually its associated transmission overheads may be marginally increased due to the assignment of security protocols, significant computation overheads must be incurred for securing the message at the location of its source task node and for unlocking security/mes-

sage extraction at the destination. Obtained security strengths and associated computation overheads vary depending on the set of protocols chosen for a given message from an available pool of protocols. Given lower bounds on the security demands of an application's messages, selecting the appropriate protocols for each message such that a system's overall security is maximized while satisfying constraints related to the resource, task precedence and deadline is a challenging and computationally hard problem.

The key contributions of this chapter are summarized as follows:

1. We propose an efficient heuristic strategy called *SHIELD* for security-aware real-time scheduling of DAG-structured applications to be executed on distributed heterogeneous systems.

2. The efficacy of the proposed scheduler is exhibited through extensive simulation-based experiments using two DAG-structured application benchmarks. Our performance evaluation results demonstrate that *SHIELD* significantly outperforms two greedy baseline strategies *SHIELDb* in terms of solution generation times (i.e., run-times) and *SHIELDf* in terms of achieved security utility.

3. Additionally, a case study on the *Traction Control* application in automotive systems has been included to exhibit the applicability of *SHIELD* in real-world settings.

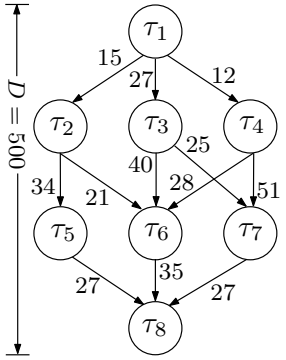
The rest of the chapter is organized as follows. Section 6.2 discusses the system models and in section 6.3, we present the proposed heuristic scheduling strategy. Section 6.4 experimentally evaluates the proposed scheduler. Section 6.5 exhibits a real-world case study on a *Traction Control* application in automotive systems. Finally, section 6.6 provides the concluding remarks.

6.2 System Models

In order to describe the proposed strategy *SHIELD*, a detailed discussion on the application, platform and security models considered in this work is required. We present these models below.

6.2.1 Application and Platform Model

Directed Acyclic Graph (DAG) is a typical way to characterize a real-time application $G(V, E)$ (refer Fig. 6.1a), where the set of vertices $V = \{\tau_1, \tau_2..., \tau_{|V|}\}$ symbolizes tasks and the set of edges E[1] conveys the precedence constraints among task pairs. An edge $e_{i,j} = (\tau_i, \tau_j) \in E$ denotes the dependency between tasks τ_i and τ_j. We assume that our application task graph has a single source τ_{entry} and a single sink τ_{exit}.



(a) A DAG $G(V, E)$

Tasks	p_1	p_2	p_3
τ_1	64	56	78
τ_2	46	36	58
τ_3	42	63	49
τ_4	36	54	42
τ_5	91	45	55
τ_6	58	72	84
τ_7	54	81	63
τ_8	75	42	61

(b) WCET Table

Figure 6.1: *(a) A DAG 'G'; (b) WCETs of* 8 *tasks on* 3 *processors.*

The platform $P = \{p_1, p_2, ..., p_{|P|}\}$ comprises of $|P|$ heterogeneous processors. These $|P|$ processors are fully interconnected through a set of $(|P|*(|P|-1))/2$ heterogeneous communication links having possibly distinct bandwidths. The bandwidths between different pairs of processors are captured in a matrix B of size $|P| \times |P|$. An element $b_{m,n} \in B$ represents the data transfer rate between processors p_m and p_n. As the links between the processors are bidirectional, $b_{m,n} = b_{n,m}$. The communication cost ($c_{i,j}^{m,n}$) for an edge $e_{i,j}$ with load $data_{i,j}$ on link $\langle p_m, p_n \rangle$ is calculated as: $c_{i,j}^{m,n} = data_{i,j}/b_{m,n}$. When τ_i and τ_j are mapped to the same processor, the value of $c_{i,j}^{m,n} = 0$. Further, the average communication time cost $\overline{c}_{i,j}$ between τ_i and τ_j is determined as: $\overline{c}_{i,j} = data_{i,j}/\overline{B}$, where $\overline{B} = \frac{1}{|P|\times(|P|-1)/2} \sum b_{m,n}$ $(1 \leqslant m \leqslant |P|; 1 \leqslant n < m)$. The worst case execution time ($WCET$) of a task τ_j on p_n is denoted as $\omega_{j,n}$. Fig. 6.1b shows the *WCETs* of each task on three heterogeneous processors.

[1]The symbol $e_{i,j}$ has been alternatively referred to as message transferred between τ_i and τ_j, in rest of the chapter.

6.2.2 Security Model

Messages transmitted over untrusted networks must be secured using services such as *confidentiality*, *integrity* and *authentication* in order to prevent tampering and leakage of sensitive data. For each security service, a set of alternative security protocols are available, any one of which may be employed in a given design. These protocols vary with respect to achievable security strengths and overheads. Here, *strength* refers to a relative numeric estimate (having a value in the range $(0, 1]$) across multiple protocols of the same or different security services. This numeric estimate is directly proportional to the computational overhead that is necessary to break security of the service protocol. When the y^{th} security protocol of service type x is applied on a message $e_{i,j}$, we denote its security strength by $s_{i,j}^{x,y}$. Without loss of generality, given any two available security protocols say, the y^{th} and $(y+k)^{th}$ protocol $(k \geqslant 1)$, we assume that $s_{i,j}^{x,y} \leqslant s_{i,j}^{x,y+k}$. Furthermore, for each service type x, a message $e_{i,j}$ is marked by a minimum threshold *security demand* $s_{i,j}^{x,\min}$; thus, $s_{i,j}^{x,y} \geqslant s_{i,j}^{x,\min}$.

Table 6.1: *Strengths and overheads of security service protocols;* $s^{x,y}$*: security strengths;* $\chi_1^{x,y}$ *(*$\chi_2^{x,y}$*): data independent (dependent) overheads*

confidentiality $x=1$	*SEAL*	*RC4*	*Blowfish*	*Knufu/ Khafre*	*RC5*	*Rijndael*	*DES*	*IDEA*
$s^{1,y}$	0.08	0.14	0.36	0.40	0.46	0.64	0.90	1.00
$\chi_2^{1,y}$ *(KB/ms)*	168.75	96.43	37.50	33.75	29.35	21.09	15.00	13.50
integrity $x=2$	*MD4*	*MD5*	*RIPEMD*	*RIPE MD-128*	*SHA-1*		*RIPE MD-160*	*Tiger*
$s^{2,y}$	0.18	0.26	0.36	0.45	0.63		0.77	1.00
$\chi_2^{2,y}$ *(KB/ms)*	23.90	17.09	12.00	9.73	6.88		5.69	4.36
authentication	*HMAC-MD5*			*HMAC-SHA-1*		*CBC-MAC-AES*		
$x=3$	$y=1$			$y=2$		$y=3$		
$s^{3,y}$	0.55			0.91		1		
$\chi_1^{3,y}$ *(ms)*	90			148		163		

In Table 6.1, we consider three security services, $x \in \{1, 2, 3\}$. Let $x = 1$ denotes the *confidentiality* service. There are *eight* different types of available protocols namely *SEAL*, *RC4*, *Blowfish*, *Knufu*, *RC5*, *Rijndael*, *DES*, and *IDEA*, for $x = 1$. Thus, when $x = 1$, the value of y ranges from *one* to *eight*. If a message say, $e_{1,2}$ of Fig. 6.1a is secured with the cryptographic algorithm *Blowfish* ($x = 1$, $y = 3$), hash function *Tiger* ($x = 2$, $y = 7$) and authentication protocol *HMAC-MD5* ($x = 3$, $y = 1$), its security strength for the different

services will be $s_{1,2}^{1,3} = 0.36$, $s_{1,2}^{2,7} = 1.00$ and $s_{1,2}^{1,3} = 0.55$. Let the symbol $w_{i,j}^{x}$ denote relative importance of different service types x (say, *confidentiality*, *integrity* and *authentication*) towards overall protection of a message $e_{i,j}$ against prevailing security threats (such that $\sum_{x=1}^{X} w_{i,j}^{x} = 1$). Now the utility associated with the use of y^{th} security protocol of service type x on a message $e_{i,j}$ is obtained as: $w_{i,j}^{x} * s_{i,j}^{x,y}$. The overall security utility of $e_{i,j}$ is then defined as:

$$SU_{i,j} = \sum_{x=1}^{X} \sum_{y=1}^{Y_x} w_{i,j}^{x} * s_{i,j}^{x,y} \tag{6.1}$$

where X denotes the number of security services being considered for a system and Y_x represents the number of security protocols available for the x^{th} service type. The security performance of a system may be measured as the sum of individual security utility values over all messages in the system:

$$T_{SU} = \sum_{e_{i,j} \in E} SU_{i,j} \tag{6.2}$$

The objective of this work is to maximize T_{SU}, the system-level total security utility.

On the other hand, *security overhead* of a system may be defined as an estimate of the overall computational effort associated with the securing of all messages in the system. There are two security-related temporal overhead components associated with each message: (i) the overhead of securing the message at its source node before transmission, and (ii) the overhead of unlocking its security in order to transform it back to a readable form at its destination node. Without loss of generality, we consider both these overhead components to have the same value. Let us assume that a message $e_{i,j}$ is transmitted from processor p_n to p_r. The generalised expression of the overhead for securing $e_{i,j}$ using the y^{th} protocol of service type x on p_n is represented as:

$$ov_{i,j}^{x,y} = \chi_{1,n}^{x,y} + data_{i,j}/\chi_{2,n}^{x,y} \tag{6.3}$$

Here, $\chi_{1,n}^{x,y}$ denotes the data-independent component of the temporal overhead associated with the execution of the y^{th} protocol on p_n. $\chi_{2,n}^{x,y}$ on the other hand, is the data-dependent component and is defined as the rate at which a message's data can be secured (unit: *KB/ms*) using the y^{th} protocol. Earlier, we have assumed $s_{i,j}^{x,y} \leqslant s_{i,j}^{x,y+k}$ $(k \geqslant 1)$ and $s_{i,j}^{x,y} \propto ov_{i,j}^{x,y}$. Hence, $ov_{i,j}^{x,y} \leqslant ov_{i,j}^{x,y+k}$, which implies, $\chi_{1}^{x,y} \leqslant \chi_{1}^{x,y+k}$ and $\chi_{2}^{x,y} \geqslant \chi_{2}^{x,y+k}$. Assuming that $e_{i,j}$ is secured using X different security services, the total overhead becomes: $\sum_{x=1}^{X} ov_{i,j}^{x,y}$. For a task τ_j executing on a processor p_n, the total security overhead of unlocking its input

messages (denoted by, $e_{i,j} \in E$) and locking its outgoing messages ($e_{j,k} \in E$) may be obtained as:

$$so_j^n = \sum_{e_{i,j} \in E} \sum_{x}^{X} ov_{i,j}^{x,y} + \sum_{e_{j,k} \in E} \sum_{x}^{X} ov_{j,k}^{x,y} \tag{6.4}$$

The right hand side (R.H.S.) of the above expression has two major components which respectively represent the overheads related to the incoming and outgoing messages of τ_j. For the three security services targeted in this work namely, *confidentiality* ($x = 1$), *integrity* ($x = 2$), and *authentication* ($x = 3$), the following have been assumed: the data-independent components are negligible for the *confidentiality* and *integrity* services [59,60] while the data-dependent component has no effect on the overhead for the *authentication* service [109]. Hence, we have considered, $\chi_{1,n}^{1,y} = \chi_{1,n}^{2,y} = 0$ and $\chi_{2,n}^{3,y} = \infty$. Using data in Table 1, the total security overhead associated with the task τ_5 in Fig. 6.1a, can be derived as discussed next. τ_5 has one input message of size 14 *KB* and one output message of size 27 *KB*. Let the security protocols *Blowfish* (*confidentiality*), *Tiger* (*integrity*) and *HMAC-HD5* (*authentication*) be used for both the input and output messages of τ_5. The total overhead obtained using equation (6.4) becomes [(0+14/37.5)+(0+14/4.36)+90]+[(0+27/37.5)+(0+27/4.36)+90] = 190.5 *ms*.

Next, we describe the proposed list-based heuristic algorithm called "*Security-aware Scheduling for Real-time DAGs on Heterogeneous Systems (SHIELD)*".

6.3 SHIELD: The Proposed Scheduler

SHIELD is a real-time scheduling strategy for heterogeneous platforms whose objective is to maximize total security utility for a given task graph application having known minimum security strength specifications for its messages. The scheduler works in three phases namely, (i) *task prioritization*, (ii) *processor allocation*, and (iii) *security enhancement*. The first two phases of *SHIELD* are encapsulated in a function called "*Heterogeneous Security-aware Makespan-minimizing Scheduler (HSMS)*" (refer line 1 of Algorithm 15). The first phase (*task prioritization*) which comprises lines 1 - 3 of Algorithm 11, computes the ranks of all the tasks and creates a priority order among the tasks. The second phase of *SHIELD* (which comprises lines 4 - 8 of function *HSMS* (refer Algorithm 11)) is used to generate a minimum *makespan* schedule for the application such that the minimum security requirements are met for all messages. *SHIELD* returns with failure if the *makespan* returns by *HSMS* violates

deadline (refer line 2 of Algorithm 15). Otherwise, *SHIELD* enters its third phase where it attempts to enhance the security strengths of all messages such that total security utility of the system is maximized.

6.3.1 Task Prioritization

The priority of any task is defined by its rank and the rank $R[\tau_j]$ of a task τ_j is determined as:

$$R[\tau_j] = \overline{\omega}_j + \max_{\tau_k \in succ(\tau_j)} \{\overline{c}_{j,k} + R[\tau_k]\} + \overline{so}_{j,min} \tag{6.5}$$

where $\overline{\omega}_j = \sum_{n=1}^{|P|} \omega_{j,n}/|P|$ and $\overline{so}_{j,min} = \sum_{n=1}^{|P|} so^n_{j,min}/|P|$, respectively denote the average execution time and minimum average security overhead associated with task τ_j over all processors. For a given processor p_n, $so^n_{j,min}$ is obtained via equation (6.4) by using those minimum values of y (y refers to security protocol ids) for each value of x (x refers to service type ids) for which the corresponding security strength $s^{x,y}_{i,j} \geqslant s^{x,min}_{i,j}$. For each successor τ_k of τ_j, the expression, $\overline{c}_{j,k} + R[\tau_k]$, delivers the aggregate value combining the rank of τ_k ($R[\tau_k]$), and the average communication time incurred for transmitting $data_{j,k}$ ($\overline{c}_{j,k}$). The max$\{\dots\}$ block thus returns the maximum aggregate value overall successors of τ_j. In line 2 (of Algorithm 11), the rank of each task is computed recursively from τ_{exit} to τ_j. Basically, the *rank* $R[\tau_j]$ of each task τ_j is an estimate of the overall relative remaining workload (combining task execution, message transmission as well as security overhead) associated with the sub-graph rooted at τ_j (up to the sink task of the DAG). The rank of each task is intended to serve two important objectives: (1) Ensuring that all ancestors of a given task are always considered for processor allocation before the task itself so that precedence constraints associated with the task graph are never violated. (ii) Tasks with relatively higher total estimated workloads corresponding to the remaining (still to be allocated) tasks in the task graph should be considered earlier for processor allocation. After obtaining the rank values, a task priority list *taskList* is created in line 3 where tasks are arranged according to the non-increasing order of $R[\tau_j]$.

6.3.2 Processor Allocation

The *processor allocation phase* (lines 4 - 8 of *HSMS*) aims to produce a static schedule by sequentially deciding for each task: (i) a *processor allocation*, (ii) an *actual start time*, and (iii) an *actual finish time*, in the order defined by the list *taskList*, such that the overall

schedule *makespan* is minimized while satisfying minimum security requirements of each message of the given task graph. The task τ_j (in *taskList*) to be scheduled next may be assigned to any heterogeneous processor in the platform. Each possible task-processor assignment is determined using two attributes: (i) *Effective Start Time* (EST) and (ii) *Effective Finish Time* (EFT).

Effective Start Time (EST) of the entry task τ_{entry} on each processor p_n is *zero*. For all other tasks, $EST[\tau_j, p_n]$ of τ_j on p_n is computed as:

$$EST[\tau_j, p_n] = \begin{cases} 0, & \text{if } \tau_j = \tau_{entry} \\ \max(avail[n], \max\limits_{\tau_i \in pred(\tau_j)} \{AFT[\tau_i] + \lceil c_{i,j}^{m,n} \rceil\}, & otherwise \end{cases} \tag{6.6}$$

where $AFT[\tau_i]$ is the actual finish time of task τ_i and $avail[n]$ is the earliest time at which processor p_n becomes available for execution. Given $EST[\tau_j, p_n]$, $\omega_{j,n}$, and so_j^n, the *effective finish time* $EFT[\tau_j, p_n]$ of task τ_j on p_n is determined as:

$$EFT[\tau_j, p_n] = EST[\tau_j, p_n] + \omega_{j,n} + \lceil so_j^n \rceil \tag{6.7}$$

Line 7 allocates that task τ_j on p_n for which the $EFT[\tau_j, p_n]$ value is minimal. Finally, the system level total security utility which is obtained by satisfying the minimum security requirements by all messages, is determined in line 8. After τ_j is mapped on p_n, the EST and EFT of τ_j on p_n become the *Actual Start Time* $AST[\tau_j]$ and the *Actual Finish Time* $AFT[\tau_j]$ of τ_j.

Algorithm 11: $HSMS(G, S^{req}, P)$

Input: $G(V, E)$, security requirement S^{req}, processor set P
Output: Security-aware *makespan*-minimizing schedule

1 Initialize $s_{i,j}^{x,y}$ ($\geqslant s_{i,j}^{x,\min}$) to least security strength that satisfies the requirements of each message $e_{i,j}$ for service type x;
2 Compute $R[\tau_j]$ using equation (6.5) for each task τ_j by traversing the DAG G starting from τ_{exit} to τ_j;
3 Sort tasks τ_j in non-increasing order of $R[\tau_j]$ and store in *taskList*;
4 **for** *each task* τ_j *in taskList* **do**
5 **for** *each processor* $p_n \in P$ **do**
6 Compute so_j^n, $EST[\tau_j, p_n]$, and $EFT[\tau_j, p_n]$ using equations (6.4), (6.6), and (6.7);
7 Allocate task τ_j on p_n for which $EFT[\tau_j, p_n]$ is minimal;
8 Determine T_{SU} using equation (6.1);

6.3.3 Security Enhancement

This phase attempts to upgrade the security strengths of the messages in the given application. The objective is to utilize both the global slack (between $AFT[\tau_{exit}]$ and D) as well as local slacks (between two consecutive tasks allocated on the same processor) to improve message security strengths such that security utility (T_{SU}) of the overall system is maximized. It progressively improves the system-level security utility in a step-by-step fashion while keeping the task-to-processor assignments and task execution order as prescribed by *HSMS* undisturbed. At each step, the strategy attempts to increase the assigned security strength of a certain message $e_{i,j}$ along one of its service dimensions (x), by one level say, from $s_{i,j}^{x,y}$ to $s_{i,j}^{x,y+1}$ (if $s_{i,j}^{x,y} < s_{i,j}^{x,Y_x}$). The selection of a ⟨message, service⟩-pair $\langle e_{i,j}, x\rangle$ is based on the highest value of a parameter called *benefit-to-cost ratio* $BCR_{i,j}^{x}$. Given the current protocol strength level y for a service of type x, $BCR_{i,j}^{x}$ is a ratio as depicted in equation (6.8). Here, $BCR_{i,j}^{x}$ provides a measure of the gain in security utility that may be obtained by enhancing the protocol strength level from y to $y+1$ to the additional resource cost that is incurred in the process. Formally, $BCR_{i,j}^{x}$ can be determined as:

$$BCR_{i,j}^{x} = \frac{B_{i,j}^{x}}{C_{i,j}^{x}} \tag{6.8}$$

Symbolically, $B_{i,j}^{x}$ is represented as:

$$B_{i,j}^{x} = w_{i,j}^{x} * (s_{i,j}^{x,y+1} - s_{i,j}^{x,y}) \tag{6.9}$$

On the other hand, $C_{i,j}^{x}$ denotes the corresponding additional computation time cost. When τ_i executes on processor p_m and τ_j on p_n, the cost $C_{i,j}^{x}$ reflects the total additional security overhead incurred for securing $e_{i,j}$ during transmission (at τ_i) and for unlocking its security during reception (at τ_j). Symbolically, $C_{i,j}^{x}$ can be obtained as:

$$C_{i,j}^{x} = \delta_i + \delta_j \tag{6.10}$$

where,

$$\delta_i = ov_{i,j}^{x,y+1} - ov_{i,j}^{x,y} \tag{6.11}$$

$$\delta_j = ov_{j,k}^{x,y+1} - ov_{j,k}^{x,y} \tag{6.12}$$

6.3.3.1 SHIELDb: The Baseline List Scheduler

In this subsection, we discuss Algorithm 12 which represents the pseudocode of *SHIELDb*. The *heapInsert()* (refer Algorithm 13) is used to initialize and maintain a max heap H, at any intermediate stage of the *SHIELD* algorithm. Each element of the heap is a ⟨message, service⟩-pair $\langle e_{i,j}, x\rangle$ at its current protocol strength level y ($< Y_x$), with the key value being $BCR^x_{i,j}$ (refer equation (6.8)). At line 4, *heapInsert()* initializes H with each ⟨message, service⟩-pair $\langle e_{i,j}, x\rangle$ being assigned to an available protocol of lowest acceptable strength (such that $s^{x,y}_{i,j} \geqslant s^{x,\min}_{i,j}$). It may be noted that in the schedule generated by *HSMS*, all messages have been assigned with these lowest acceptable protocol strengths values $s^{x,y}_{i,j}$. The *while* loop (lines 5-12) repeats until the available slacks in the partial schedule become

Algorithm 12: *SHIELDb*(G, S^{req}, D, P)

Input: $G(V, E)$, security requirement S^{req}, D, processor set P
Output: Maximizes the aggregate security strength of system

1 Call *HSMS*(G, S^{req}, P) to obtain the allocated processor *pro*, *AST*, *AFT* of each task, the assigned security service protocols on the messages of G, security overhead of each task-processor pair so^n_j, and priority list *taskList*;
2 **if** $AFT[\tau_{exit}] > D$ **then** Exit;
3 Initialize a Max-Heap H, and set $curCost = 0$, $\tau_\lambda = \tau_{exit}$;
4 Call *heapInsert*$(i, j, x, s^{x,y}_{i,j}, s^{x,y+1}_{i,j}, H)$ for each service type x on every edge $e_{i,j}$, if $s^{x,y}_{i,j}$ is smaller than $s^{x,Y_x}_{i,j}$;
5 **while** H *is non-empty* **do**
6 $\quad \{B^x_{i,j}, C^x_{i,j}, s^{x,y}_{i,j}, i, j\}$ = Extract top element from H;
7 $\quad so_i^{pro[\tau_i]} += \delta_i$, $so_j^{pro[\tau_j]} += \delta_j$;
8 $\quad$ *updateSched*$(G, S, \tau_i, taskList)$;
9 $\quad$ **if** $AFT[\tau_{exit}] > D$ **then**
10 $\quad\quad$ Revert lines 7 - 8;
11 $\quad$ **else if** $s^{x,y+1}_{i,j} < s^{x,Y_x}_{i,j}$ **then**
12 $\quad\quad$ *heapInsert*$(i, j, x, s^{x,y+1}_{i,j}, s^{x,y+2}_{i,j}, H)$;
13 Output: *Valid* schedule having security utility T_{SU};

insufficient to enhance the protocol strength of any ⟨message, service⟩-pair in H. At any given iteration of the *while* loop, line 6 extracts the ⟨message, service⟩-pair say, $\langle e_{i,j}, x\rangle$ which offers the highest benefit-to-cost ratio, from the max heap H. Line 7 updates $so_i^{pro[\tau_i]}$ and $so_j^{pro[\tau_j]}$, the security overheads at the source and destination nodes of message $e_{i,j}$, after

enhancing its protocol strength for service type x, from y to $y+1$.

Algorithm 13: *heapInsert*$(i, j, x, s_{i,j}^{x,y}, s_{i,j}^{x,y+1}, H)$

Input: $i, j, x, s_{i,j}^{x,y}, s_{i,j}^{x,y+1}, H$
Output: Update the heap H
1 Compute $B_{i,j}^{x}$, $C_{i,j}^{x}$ and $BCR_{i,j}^{x}$ using equations (6.9),(6.10) and (6.8);
2 *insert* $\{B_{i,j}^{x}, C_{i,j}^{x}, s_{i,j}^{x,y}, i, j\}$ in H with key $BCR_{i,j}^{x}$;

In line 8, *updateSched()* (refer Algorithm 14) updates the partial schedule after the security enhancement a message $e_{i,j}$, while keeping undisturbed the task-to-processor assignments and task execution order as prescribed by *HSMS*. The partial schedule is updated by minimally increasing (if needed) the start times as well as the finish times of the following tasks.

Algorithm 14: *updateSched*$(G, S, \tau_\lambda, taskList)$

Input: $G, S, \tau_\lambda, taskList$
Output: Update start and finish time of each task, from τ_λ to τ_{exit}
1 Update $avail[n]$ for each processor $p_n \in P$;
2 **for** $\tau_k = \tau_\lambda$ *to* τ_{exit} **do**
3 Compute $EST[\tau_k, pro[\tau_k]]$ and $EFT[\tau_k, pro[\tau_k]]$ using equations (6.6) and (6.7);
4 Set $AST[\tau_k] = EST[\tau_k, pro[\tau_k]]$;
5 Set $AFT[\tau_k] = EFT[\tau_k, pro[\tau_k]]$;
6 Update $avail[pro[\tau_k]]$;

1. The finish time of τ_i is increased by $\lceil \delta_i \rceil$ (refer equation 6.11) on processor $pro[\tau_i]$.

2. Due to increase in the finish time of τ_i, start time of τ_j may need to be enhanced by atmost $\lceil \delta_i \rceil$ on processor $pro[\tau_j]$. Finish time of τ_j must also be minimally increased by at most $\lceil \delta_i + \delta_j \rceil$ (refer equation 6.12).

3. Due to enhancement in the finish times of τ_i and τ_j, the start times of all descendent tasks of τ_i and τ_j in DAG G, may need to be delayed.

4. These adjustments in the execution intervals of τ_i, τ_j and their descendent tasks, may induce overlapped execution intervals in the partial schedule. Such overlaps must be removed by minimally shifting/delaying the execution of tasks in the partial schedule.

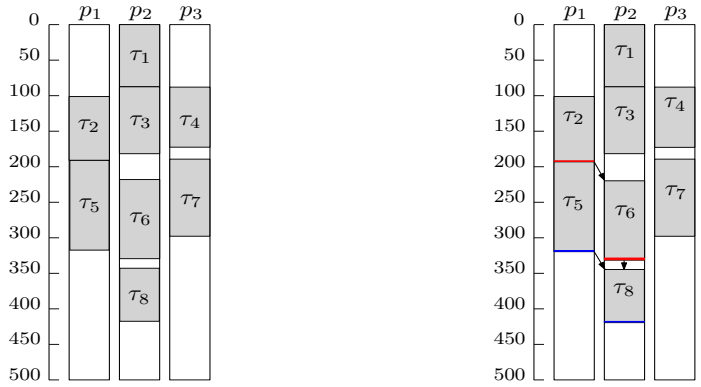
The adjustment procedure in step 4 above may potentially lead to a scenario where the completion time of the exit task τ_{exit} overshoots the stipulated deadline D of the application (line 9). In such a scenario, the modifications carried out in steps 1 to 4 above are reverted back (line 10). This case implies that the protocol strength enhancement of $\langle e_{i,j}, x\rangle$ was not successful due to insufficient resources. Further, as the protocol strength enhancement attempt failed, $\langle e_{i,j}, x\rangle$ is not re-inserted back into the heap H. On the other hand, if the protocol strength enhancement is successfully conducted and service type x has not already been assigned to the highest protocol level, *SHIELDb* calls *heapInsert()* to update $BCR^x_{i,j}$ and insert $\langle e_{i,j}, x\rangle$ into the heap using $BCR^x_{i,j}$ as key (lines 11-12). This procedure continues until heap H becomes empty.

Table 6.2: *Messages security requirements S^{req} for the DAG in Fig. 6.1a*

Messages	$s^{1,\min}_{i,j}$	$s^{2,\min}_{i,j}$	$s^{3,\min}_{i,j}$	$w^1_{i,j}$	$w^2_{i,j}$	$w^3_{i,j}$
$data_{1,2}$	0.3	0.2	0.4	0.2	0.3	0.5
$data_{1,3}$	0.1	0.2	0.2	0.5	0.3	0.2
$data_{1,4}$	0.1	0.4	0.3	0.3	0.6	0.1
$data_{2,5}$	0.2	0.2	0.4	0.3	0.5	0.2
$data_{2,6}$	0.4	0.2	0.3	0.7	0.1	0.2
$data_{3,6}$	0.3	0.1	0.1	0.2	0.4	0.4
$data_{3,7}$	0.3	0.2	0.1	0.1	0.3	0.6
$data_{4,6}$	0.2	0.5	0.3	0.2	0.6	0.2
$data_{4,7}$	0.3	0.4	0.2	0.2	0.2	0.6
$data_{5,8}$	0.3	0.1	0.4	0.2	0.4	0.4
$data_{6,8}$	0.3	0.2	0.3	0.3	0.6	0.1
$data_{7,8}$	0.4	0.3	0.4	0.2	0.3	0.5

Example - 1: Fig. 6.1a depicts an example task graph application having 8 task nodes and 12 edges. Here, the deadline for the application is 500. Fig. 6.1b lists the worst-case execution times of all tasks on three available processors. Table 6.2 depicts the minimum security requirements and relative priority for three security service types, associated with each message of the DAG application in Fig. 6.1a. Table 6.3 lists the performance values of alternative protocols available for each service type on the three processors. Bandwidths $b_{m,n}$ of the communication channel between different processor pairs are: $b_{1,2} = b_{2,1} = 1$, $b_{2,3} = b_{3,2} = 3$ and $b_{3,1} = b_{1,3} = 2$. The priority order of tasks is obtained as: $taskList =$

$\{\tau_1, \tau_3, \tau_4, \tau_2, \tau_6, \tau_7, \tau_5, \tau_8\}$. Fig. 6.2a shows the schedule Gantt chart obtained using *HSMS*. It can be observed from Fig. 6.2a that the *HSMS* schedule whose *makespan* is 417, has global slack 83 ($D - AFT[\tau_{exit}] = 500 - 417$), along with various local slack intervals on different processors: $\langle p_1$: 87-102, 317-344 $\rangle$, $\langle p_2$: 181-216, 330-344 $\rangle$, $\langle p_3$: 87-91, 174-190, 294-344 $\rangle$.



(a) *HSMS* **(b)** *PartialSched1*

Figure 6.2: *Gantt charts of schedules: (a) HSMS (makespan* $= 417$*,* $T_{SU} = 4.828$*); (b) PartialSched1 (makespan* $= 418$*), for the DAG in Fig. 6.1a.*

Fig. 6.2b shows the partial schedule of *SHIELDb* after the first security strength enhancement, which happens in line 8. This enhancement corresponds to the increment of the confidentiality service ($x = 1$) of message $e_{2,6}$ from protocol *Knufu/Khafre* ($y = 4$; $s_{2,6}^{1,4} = 0.4$) to *RC5* ($y = 5$; $s_{2,6}^{1,5} = 0.46$). Due to this change in protocol strength, the execution time of task τ_2 (on p_1) and τ_6 (on p_2) increases by 1 *ms* each. Hence, the completion time of τ_2 is changed from 195 to 196. Increase in τ_2's completion time induces a delay of 1 *ms* in the start time of τ_6. Thus, the finish time of τ_6 gets increased by 2 *ms* from 330 to 332. Further, finish times of τ_5 (successor of τ_2 in the DAG (Fig. 1a) and scheduled after τ_2 on processor p_1 (Fig. 2b)) and τ_8 (successor of τ_5 and τ_6 in the DAG and scheduled after τ_6 on processor p_2) get enhanced by 1 *ms*. It may be noted that, due to available local slack in processor p_2, the execution interval of τ_8 gets shifted by *one* time unit, although τ_6 is shifted by *two* time units in the schedule.

6.3.3.2 SHIELD: An Enhancement over SHIELDb

In each iteration, *SHIELDb* extracts a ⟨message, service⟩-pair from the heap H and minimally updates the partial schedule using steps 1 to 4 discussed above. This partial schedule

Table 6.3: *Performance of alternative protocols of different security services on three processors*

Confidentiality	$s^{1,y}$	0.08	0.14	0.36	0.40	0.46	0.64	0.90	1.00
	p_1	1012.5	578.58	225.0	202.50	176.10	126.54	90.0	81.0
$\chi_n^{1,y}$(KB/ms)	p_2	1518.7	867.87	337.5	303.75	264.15	189.81	135.0	121.5
	p_3	1181.2	675.01	262.5	236.25	205.45	147.63	105.0	94.5
Integrity	$s^{2,y}$	0.18	0.26	0.36	0.45	0.63	0.77	1.00	
	p_1	143.40	102.54	72.0	58.38	41.28	34.14	26.16	
$\chi_n^{2,y}$ (KB/ms)	p_2	215.10	153.81	108.0	87.57	61.92	51.21	39.24	
	p_3	167.30	119.63	84.0	68.11	48.16	39.83	30.52	
Authentication	$s^{3,y}$		0.55		0.91		1.00		
	p_1		15		24.67		27.17		
$\chi_n^{3,y}$ (ms)	p_2		10		16.44		18.11		
	p_3		12.86		21.14		23.29		

adjustment procedure repeats $\sum_{x=1}^{X} |E| \times Y_x$ times, in the worst case. Given a message $e_{i,j}$ whose security is being enhanced, the execution intervals of $O(|V|)$ tasks ranked at most $R[\tau_i]$ may need to be adjusted in the schedule. However, this expensive overhead can be reduced if we have a mechanism which can minimize the number of times such schedule adjustments are conducted. We now discuss *SHIELD* which delivers significantly lower overheads compared to *SHIELDb*, while producing an identical schedule as output. The pseudocode of *SHIELD* is presented in Algorithm 15. Lines 1-6, 17-23 and 25 are very similar to that of *SHIELDb* (refer Algorithm 12). The idea of *SHIELD* can be summarised through the following steps.

1. Determine a collection say, ξ of ⟨message, service⟩-pairs (which can be obtained as a sequence, through a set of consecutive extractions from heap H), all of whom can be used to effect message security enhancements, while ensuring that the resulting combined overhead will not lead to deadline violation. It may be noted from equation (6.10) that for a given ⟨message, service⟩-pair $\langle e_{i,j}, x\rangle$ whose security has to be enhanced, the schedule *makespan* may get extended by at most $C_{i,j}^x$ for a single protocol enhancement step.

2. This sequence of ⟨message, service⟩-pairs is obtained by repeatedly extracting the root (say, $\langle e_{i,j}, x\rangle$) of H (line 6), accumulating the corresponding maximum overhead $C_{i,j}^x$ in a variable *curCost* (line 9), determining the new key value $BCR_{i,j}^x$, and finally re-inserting $\langle e_{i,j}, x\rangle$ back into the heap with the new key value (line 12). Such repeated extractions from the heap are conducted until the total overhead in *curCost* is less

Algorithm 15: *SHIELD*(G, S^{req}, D, P)

Input: $G(V, E)$, security requirement S^{req}, D, processor set P
Output: Maximizes the aggregate security strength of system

1 Call *HSMS*(G, S^{req}, P) to obtain the allocated processor *pro*, *AST*, *AFT* of each task, the assigned security service protocols on the messages of G, security overhead of each task-processor pair so_j^n, and priority list *taskList*;
2 **if** $AFT[\tau_{exit}] > D$ **then** Exit;
3 Initialize a Max-Heap H, and set $curCost = 0$, $\tau_\lambda = \tau_{exit}$;
4 Call $heapInsert(i, j, x, s_{i,j}^{x,y}, s_{i,j}^{x,y+1}, H)$ for each service type x on every edge $e_{i,j}$, if $s_{i,j}^{x,y}$ is smaller than $s_{i,j}^{x,Y_x}$;
5 **while** H *is non-empty* **do**
6 $\quad \{B_{i,j}^x, C_{i,j}^x, s_{i,j}^{x,y}, i, j\}$ = Extract top element from H;
7 $\quad$ **if** $curCost + C_{i,j}^x \leq D - AFT[\tau_{exit}]$ **then**
8 $\quad\quad so_i^{pro[\tau_i]} += \delta_i$, $so_j^{pro[\tau_j]} += \delta_j$;
9 $\quad\quad$ Set $curCost = curCost + C_{i,j}^x$;
10 $\quad\quad$ **if** $R[\tau_i] > R[\tau_\lambda]$ **then** Set $\tau_\lambda = \tau_i$;
11 $\quad\quad$ **if** $s_{i,j}^{x,y+1} < s_{i,j}^{x,Y_x}$ **then**
12 $\quad\quad\quad$ $heapInsert(i, j, x, s_{i,j}^{x,y+1}, s_{i,j}^{x,y+2}, H)$;
13 $\quad\quad$ **else if** $curCost > 0$ **then**
14 $\quad\quad\quad$ $updateSched(G, S, \tau_\lambda, taskList)$;
15 $\quad\quad\quad$ Set $curCost \leftarrow 0$, $\tau_\lambda = \tau_{exit}$;
16 $\quad\quad\quad$ $heapInsert(i, j, x, s_{i,j}^{x,y}, s_{i,j}^{x,y+1}, H)$;
17 $\quad\quad$ **else**
18 $\quad\quad\quad$ $so_i^{pro[\tau_i]} += \delta_i$, $so_j^{pro[\tau_j]} += delta_j$;
19 $\quad\quad\quad$ $updateSched(G, S, \tau_i, taskList)$;
20 $\quad\quad\quad$ **if** $AFT[\tau_{exit}] > D$ **then**
21 $\quad\quad\quad\quad$ Revert lines 18 - 19;
22 $\quad\quad\quad$ **else if** $s_{i,j}^{x,y+1} < s_{i,j}^{x,Y_x}$ **then**
23 $\quad\quad\quad\quad$ $heapInsert(i, j, x, s_{i,j}^{x,y+1}, s_{i,j}^{x,y+2}, H)$;
24 **if** $curCost > 0$ **then** textitupdateSched$(G, S, \tau_\lambda, taskList)$;
25 Output: *Valid* schedule having security utility T_{SU};

than the available global slack ($D - AFT[\tau_{exit}]$) (line 7).

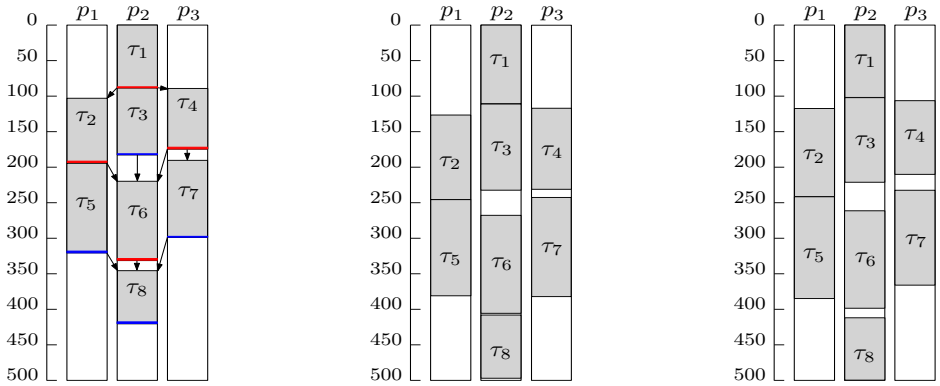
3. For a given ⟨message, service⟩-pair $\langle e_{i,j}, x\rangle$, the additional security overhead components δ_i and δ_j at the source and destination tasks τ_i and τ_j of $e_{i,j}$ (refer equation 9), are added to the execution times of τ_i and τ_j (line 8).

4. Next, we find out the highest ranked task (say, τ_λ) among all the source tasks (τ_i) of messages in collection ξ (line 10).

5. Finally, the partial schedule is updated (by minimally enhancing the start times of all tasks whose ranks are at most τ_λ) such that there are no overlaps between the execution durations of: (i) inter-dependent tasks in the task graph, and (ii) tasks allocated on the same processor. This operation is performed by function *updateSched*$(G, S, \tau_\lambda, taskList)$ in line 14 of *SHIELD*. It may be noted that *updateSched*() does not perform any schedule adjustments for tasks whose ranks are higher than τ_λ.

6. As mentioned in step 3, the $c^x_{i,j}$ values (which are accumulated in a variable *curCost*) provide upper bounds on the possible increase in *makespans* due to security protocol enhancements. Thus, subsequent to partial schedule adjustment (in line 15 or 20), the enhanced schedule *makespan* value may be less than the final value of *curCost*. Hence, it is possible that the *if*-condition in line 7 of Algorithm 15, gets satisfied more than once. However, when the *if*-condition in line 7 fails just after a schedule adjustment step (global slack is insufficient; $curCost = 0$), then *SHIELD* executes lines 19-27, similar to lines 7-14 of *SHIELDb* (Algorithm 12).

Example - 2: We employ the same scenario as used in *Example* - 1. Fig. 6.3a shows the partial schedule as obtained after the first $five$ message security strength enhancements, all of whom correspond to protocols of the confidentiality service ($x = 1$). The first *four* of these enhancements are associated with the message $e_{2,6}$, which is initially designated to use the confidentiality protocol *Knufu/Khafre* ($y = 4$; $s^{1,4}_{2,6} = 0.4$) in the *HSMS* schedule. The *four* confidentiality protocol enhancements to carried out by *SHIELD* to message $e_{2,6}$ are: *RC5* ($y = 5$; $s^{1,5}_{2,6} = 0.46$) → *Rijndael*($y = 6$; $s^{1,6}_{2,6} = 0.64$) → *DES* ($y = 7$; $s^{1,7}_{2,6} = 0.9$) → *IDEA* ($y = 8$; $s^{1,8}_{2,6} = 1.0$). The *fifth* confidentiality protocol enhancement is for the message $e_{1,4}$: *RC4* ($y = 2$; $s^{1,2}_{1,4} = 0.14$) → *Blowfish* ($y = 3$; $s^{1,3}_{1,4} = 1$). Due to these changes in protocol strengths, the execution times of task τ_2 (on p_1), τ_6 (on p_2), τ_1 (on p_2) and τ_4



(a) *PartialSched2* **(b)** *SHIELD* **(c)** *SHIELDf*

Figure 6.3: *Gantt charts depicting the schedules: (a) PartialSched2 (makespan = 419); (b) SHIELD (makespan = 499, T_{SU} = 11.856); (c) SHIELDf (makespan = 500, T_{SU} = 8.663), for the DAG in Fig. 6.1a.*

(on p_3) increase by 1 *ms* each. It may be noted that τ_1 has the highest rank among the above-mentioned tasks. Now, the execution intervals of all the tasks ranked lower than τ_1 in the partial schedule are adjusted minimally so that there are no overlaps in the updated schedule. Thus, the completion time of τ_1 (which is changed from 87 to 88) induces a delay of 1 *ms* in the start times of τ_2, τ_3 and τ_4. So, the finish times of τ_2, τ_3 and τ_4 get increased by 2, 1, and 2 time units: ⟨195 to 197⟩, ⟨181 to 182⟩ and ⟨174 to 176⟩. Further, finish times of all the descendent/lower-ranked tasks τ_5, τ_6, τ_7 and the exit node τ_8 (of τ_1 to τ_4) get increased by 2, 3, 1 and 2 time units. Thus, the new *makespan* of the updated schedule becomes 419 (finish time of τ_8).

Gantt charts in Figs. 6.2a and 6.3b depict the schedules generated by *HSMS* (*makespan*: 417, T_{SU}: 4.828) and *SHIELD* (*makespan*: 499, T_{SU}: 11.856), respectively. Table 6.4 lists the messages' security strengths archived by the algorithms for the available service types.

6.3.4 Complexity Analysis

The complexity of the *SHIELD* algorithm is comprised of the complexities of its three integral phases, namely (i) *task prioritization*, (ii) *processor allocation* and (iii) *security enhancement*. In the *task prioritization* phase, overhead of computing ranks (refer equation (6.5)) of all tasks is obtained as the summation of the complexities for determining, (a) $\overline{\omega}_j$: takes $O(|V| \times |P|)$ time; (b) $\overline{c}_{i,j}$: takes $O(|E| + |P|^2)$ time; and (c) $\overline{so}_{j,min}$: takes $O(|E| \times |P|)$ time (assuming the total number of available security services to be constant).

Table 6.4: *Schedulers assigned security strengths of each message*

Messages	*HSMS*			*SHIELD*			*SHIELDf*		
	$s^1_{i,j}$	$s^2_{i,j}$	$s^3_{i,j}$	$s^1_{i,j}$	$s^2_{i,j}$	$s^3_{i,j}$	$s^1_{i,j}$	$s^2_{i,j}$	$s^3_{i,j}$
$data_{1,2}$	0.36	0.26	0.55	1.00	1.00	1.00	0.40	0.26	0.91
$data_{1,3}$	0.14	0.26	0.55	1.00	1.00	1.00	0.40	0.26	1.00
$data_{1,4}$	0.14	0.46	0.55	1.00	1.00	0.91	0.40	1.00	0.55
$data_{2,5}$	0.36	0.26	0.55	1.00	1.00	0.55	0.36	1.00	0.91
$data_{2,6}$	0.40	0.26	0.55	1.00	1.00	1.00	1.00	0.26	0.91
$data_{3,6}$	0.36	0.18	0.55	1.00	1.00	1.00	0.40	0.26	1.00
$data_{3,7}$	0.36	0.26	0.55	1.00	1.00	1.00	0.40	0.36	1.00
$data_{4,6}$	0.36	0.63	0.55	1.00	1.00	1.00	0.40	1.00	1.00
$data_{4,7}$	0.36	0.46	0.55	1.00	1.00	1.00	0.40	0.63	1.00
$data_{5,8}$	0.36	0.18	0.55	1.00	1.00	1.00	0.36	0.26	1.00
$data_{6,8}$	0.36	0.26	0.55	1.00	1.00	0.55	0.40	1.00	0.55
$data_{7,8}$	0.40	0.36	0.55	1.00	1.00	1.00	0.46	0.46	1.00

Thus, the overall complexity of computing the ranks of all tasks is $O(|E| \times |P|)$ (refer line 2 of Algorithm 11). The tasks are then sorted in non-decreasing order based on their ranks which takes $O(|V| \times log|V|)$ time. Thus, the total time complexity of the *task prioritisation phase* becomes $O(|E| \times |P| + |V|\ log(|V|)) = O(|E| \times |P|)$.

Complexity of the *processor allocation phase* is dominated by the overhead of computing $EST[\tau_j, p_n]$ (refer equation 6.6; in line 6 of Algorithm 11) for each task-processor pair within the nested *for* loops (outer loop: lines 4-7; inner loop: lines 5-6). Determining $EST[\tau_j, p_n]$ requires $O(1)$ computations over all predecessors of task τ_j and has an overhead of $O(\#predecessors)$. The total number of predecessors of all tasks is equal to the aggregate count of edges in the task graph. Thus, the amortized overhead of determining $EST[\tau_j, p_n]$ is $O(|P|(|V|+|E|)/(|P|\times|V|)) = O((|V|+|E|)/|V|) = O(|E|/|V|)$. So, the total overhead of computing $EST[\tau_j, p_n]$ on all task-to-processor pairs is $O(|P|\times|V|\times|E|/|V|) = O(|E|\times|P|)$.

In the worst case, the *security enhancement phase* (refer Algorithm 15) may need to conduct level-by-level increments of the security strengths of all ⟨message, service⟩-pairs, starting from the least available security strength to the highest available strength. If X denotes the number of security service types and Y_x the number of available protocols (at distinct security strength levels) for service type x, the total number of security strength level increments of ⟨message, service⟩-pairs, in the worst case, is given by: $\sum_{x=1}^{X} Y_x \times |E|$.

Each such increment is mainly associated with (a) a heap extraction/re-insertion (which consumes $O(log(|E| \times X))$ time), and (b) partial schedule adjustment (takes $O(|V|)$ time). For *SHIELDb*, the complexity of this phase becomes: $O((\sum_{x=1}^{X} Y_x \times |E|)(log(|E| \times X) + |V|)) \approx O(|E|\ log(|E|) + |E| \times |V|)$, assuming the values of X and Y_x to be constant. For *SHIELD*, the expensive $O(|V|)$ partial schedule adjustment step is conducted only a small constant number (say, κ) of times on average. Hence, the complexity of this phase gets reduced and can be represented as: $O(|E|\ log(|E|) + \kappa\ |V|) \approx O(|E|\ log(|E|))$.

Hence, the overall complexity of *SHIELDb*, including overheads for all three phases, can be expressed as: $O(|E| \times |P| + |E| \times |P| + |E|\ log(|E|) + |E| \times |V|) \approx O(|E| \times |V|)$. Similarly, the overall complexity of *SHIELD* is: $O(2\ |E| \times |P| + |E|\ log(|E|)) \approx O(|E| \times |P|)$.

6.4 Experiments and Results

In this section, we experimentally evaluate the performance of *SHIELD* against a greedy baseline strategy called *SHIELDf*, developed by us. We now present a brief overview of *SHIELDf* before discussing the experimental setup, the performance metrics, and the experimental results.

SHIELDf: Given a *HSMS* schedule that satisfies the minimum security strength requirements of each inter-task message of an application, both the algorithms *SHIELD* and *SHIELDf* attempt to enhance the security strengths of each message, while keeping the *HSMS* generated task-to-processor assignments and task scheduling order undisturbed. However, the two algorithms differ in the definition of the *key* associated with the max heap H of ⟨message, service⟩-pairs. Whereas *SHIELD* performs security strength updates for ⟨message, service⟩-pairs $\langle e_{i,j}, x \rangle$ using *benefit-to-cost ratios* ($BCR_{i,j}^{x}$) as *keys*, *SHIELDf* uses only the *benefit* values ($B_{i,j}^{x}$) as *keys*. Hence unlike *SHIELD*, at each security enhancement step, *SHIELDf* picks the ⟨message, service⟩-pair that provides the highest gain in security strengths while ignoring the additional execution time resource that is incurred to achieve this gain.

Example - 3: Continuing with the same example scenario as used *Examples* 1 and 2, the schedule obtained using *SHIELDf* is shown in Fig. 6.3c. It can be observed that the aggregate security strength achieved using *SHIELD* ($T_{SU} = 11.856$; refer Fig. 6.3b) is higher than *SHIELDf* ($T_{SU} = 8.663$).

6.4.1 Experimental Setup

The *SHIELD* strategy has been evaluated and compared against *SHIELDb* and *SHIELDf* using extensive simulation-based experiments conducted by employing two real-world task graph application benchmarks: *Gaussian Elimination* [85] and *Cybershake* [39].

Gaussian Elimination: Gaussian Elimination is a function for solving systems of linear equations. The linear equations are denoted as a square matrix of size $\nu \times \nu$, where ν is provided as input. The number of tasks and edges in a Gaussian Elimination task graph is calculated as $((\nu^2 + \nu - 2)/2)$ and $(\nu^2 - \nu - 1)$, respectively. For example, a Gaussian Elimination graph with *matrix size*, $\nu = 5$ (refer Fig. 5.4b) has 14 task nodes and 19 edges.

Cybershake: Cybershake is an earthquake hazard characterization graph used by the Southern California Earthquake Center. This task graph has five types of nodes: ExtractSGT, SeismogramSynthesis, PeakValCalcOkaya, ZipSies, and ZipPSA, of which the SeismogramSynthesis nodes are the most computationally intensive. Cybershake task graph size is influenced by the number of ExtractSGT and SeismogramSynthesis nodes. For simplicity, we fix the number of ExtractSGT nodes to 2 and divide SeismogramSynthesis nodes near-equally among the ExtractSGT nodes. If the number of SeismogramSynthesis nodes is υ, then the total number of task nodes and edges in the task graph becomes $(2\upsilon + 4)$ and 4υ. An example of the Cybershake graph with $\upsilon = 8$ (refer Fig. 5.4a) has 20 task nodes and 32 edges.

Data Generation Framework: We have generated an exhaustive set of random input test cases by carefully varying the following set of parameters to conduct our experiments.

1. **Task graph size**: For Gaussian Elimination, *matrix-sizes* $\nu = \{10, 14, 17, 20, 22\}$ results in task graphs with number of tasks $|V| = \{54, 104, 152, 209, 252\}$ and edges $|E| = \{89, 181, 271, 379, 461\}$. Similarly for Cybershake, *SeismogramSynthesis* $\upsilon = \{23, 48, 73, 98, 123\}$ results in task graphs with $|V| = \{50, 100, 150, 200, 250\}$ and $|E| = \{92, 192, 292, 392, 492\}$.

2. **Number of processors**: $|P| = \{4, 8, 16, 32\}$.

3. **Task execution times**: The worst case execution time of each task is generated by the following three steps. (i) Selection of an average execution time $\overline{\omega}_{DAG} = \{100, 200, 300, 400, 500\}$ over all tasks in the DAG. (ii) Given $\overline{\omega}_{DAG}$, we generate the average execution time $(\overline{\omega}_j)$ of each task τ_j over all processors using normal distribution with mean $\mu = \overline{\omega}_{DAG}$ and different standard deviation values $\sigma = \{10, 20, 30\}$. (iii) Finally, we obtain the execution times as $\omega_{j,n}$ using normal distribution with $\mu = \overline{\omega_j}$ and $\sigma = \overline{\omega_j} \times \beta$,

for each task τ_j on each processor p_n. Here, $\beta = \{0.1, 0.25, 0.5, 0.75, 1\}$ is the heterogeneity factor that indicates the skewness among execution times of a task on different processors.

4. **Communication-to-Computation Ratio (CCR)**: *CCR* is the ratio of the overhead related to inter-task message transmission and task execution. A higher CCR indicates that the system spends relatively more time in data transmission than task computation. Values of CCR chosen in this work are: $CCR = \{0.1, 0.25.0.5, 0.75, 1\}$. The mean inter-task message size ($\overline{data}_{DAG}$; in bytes) for a task graph is given by:

$$\overline{data}_{DAG} = CCR \times \overline{\omega}_{DAG} \times \overline{B}$$

where $\overline{B}$ is the average communication bandwidth randomly sampled from $\overline{B} = \{5\ Gbps, 10\ Gbps\}$. An element $b_{m,n} \in B$ denotes the actual bandwidth on a link $\langle p_m, p_n \rangle$. The values of $b_{m,n}$ are obtain through sampling from a normal distribution with $\mu = \overline{B}$ and $\sigma = 0.2 \times \overline{B}$. These $b_{m,n}$ values are further scaled appropriately such that $\sum_{m=1}^{|P|} \sum_{n=1}^{m-1} b_{m,n}$ becomes $|P| \times (|P|-1)/2 \times \overline{B}$. Similarly, the output message size $data_{i,j}$ for each edge $e_{i,j}$ in the task graph is sampled from a normal distribution $\mu = \overline{data}_{DAG}$ and $\sigma = 0.2 \times \overline{data}_{DAG}$ and then scaled such that $\sum data_{i,j}$ over all edges in the task graph become equal to $|E| \times \overline{data}_{DAG}$.

5. **Security parameters**: The minimum security demands of different service types for each message in the application are determined by randomly selecting the demand values from the range $[0, 0.5]$. The relative securing priority $w_{i,j}^x$ for each service type x associated with every message $e_{i,j}$ is determined randomly, such that $\sum_x^X w_{i,j}^x = 1$. Execution time overheads associated with the available security protocols of different service types on a particular processor are obtained from the work in [101] and presented in Table 6.1. Table 6.1 lists the data-dependent overheads $\chi_2^{x,y}$ of each protocol y for confidentiality (x=1) service, and integrity (x=2) service as well as the data-independent overheads $\chi_1^{x,y}$ for authentication service (x=3) [100, 101]. Using the values of the overheads presented in Table 6.1 as the mean values of normal distributions, the corresponding overheads associated with other heterogeneous processors are obtained by scaling the standard deviation with the degree of heterogeneity factor β. As an example, for the authentication service (x=3), the values of execution time overheads on different processors are obtained using a normal distribution with $\mu = \chi_1^{3,y}$ and $\sigma = \chi_1^{3,y} \times \beta$.

6. **Deadline extension rate**: It may be noted that *HSMS* delivers the shortest *makespan* schedule while satisfying the minimum security demands of each message in the application. *SHIELD* returns with *failure*, if *HSMS* delivers a *makespan* which overshoots

the given deadline. Thus, *SHIELD* is expected to generate a security-aware schedule if the given application deadline is greater than *HSMS*'s *makespan*. Further, for any given application, feasible schedules with progressively higher security strengths should be achieved by *SHIELD*, as the deadline is relaxed more and more with respect to the *makespan* of *HSMS*. We define *deadline extension rate* (∂) as the ratio of the specified deadline of a DAG-structured application 'G' with respect to the *HSMS* delivered *makespan*. The performance of *SHIELD* in terms of maximizing application security protection has also been evaluated by deciding the average normalized security utility for different values of *deadline extension rate*. Different values of ∂ used in our experiments are: $\partial = \{1, 1.2, 1.4, 1.6, 1.8, 2\}$.

Simulation Framework: The simulation framework is written in *C* and is executed on a system having the following configuration: (i) Intel® Core™ i7-8550U CPU @ 1.80GHz ×8, (ii) 8 GiB Memory, and (iii) Ubuntu 20.04 LTS OS.

6.4.2 Performance Metrics

Performance of *SHIELD* has been evaluated using the following two metrics, namely (1) *Normalised Security Utility (NSU)* and (2) *Run-time*.

1. **Normalised Security Utility (NSU)**: For a given task graph, the *Normalised Security Utility* of an application is the ratio of the security utility achieved by the algorithm to the maximum achievable security utility obtained by assigning all service types of all messages at the highest available security strength levels.

$$NSU = \frac{T_{SU}}{\sum_{e_{i,j} \in E} \sum_{x=1}^{X} w_{i,j}^{x} * s_{i,j}^{x,Y_x}} \times 100 \tag{6.13}$$

   where $s_{i,j}^{x,Y_x}$ represents the maximum available security strength of service type x on an edge $e_{i,j}$. It may be noted that higher the value of achieved NSU, better is the performance of a scheduling algorithm.

2. **Run-time**: This metric determines the average run-time (in ms) taken by a scheduling algorithm for data sets generated employing a fixed set of input parameters.

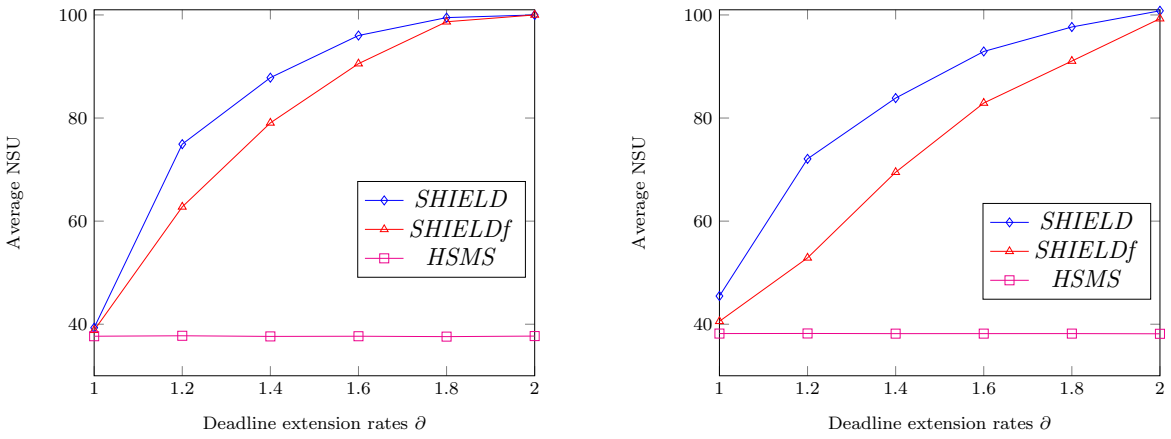
6.4.3 Performance Results

In this subsection, we conduct $five$ experiments to analyze the performance of *SHIELD*, *SHIELDb*, *SHIELDf* and *HSMS* using two benchmarks: Gaussian Elimination [85] and

Cybershake [39]. Each data point in these experiments is obtained as the average over solutions generated with 250 different task graph data corresponding to a fixed set of input parameter values.

6.4.3.1 Experiment-1: Effect of variation in deadline extension rates

In this experiment, we analysed the *Normalised Security Utility (NSU)* achieved by the schedulers *HSMS*, *SHIELD*, *SHIELDf* for different deadline extension rates (∂). The parameters $|P|$, CCR and β are set to 32, 0.5 and 0.75, respectively. Number of tasks $|V|$ for Gaussian Elimination (Fig. 6.4a) is set to 152 and for Cybershake (Fig. 6.4b) is set to 150. It can be observed from Fig. 6.4 that the average NSU achieved by the schedulers *SHIELD* and *SHIELDf* monotonically increase as the deadline increases. This signifies that *SHIELD* and *SHIELDf* are able to efficiently harness higher available slacks to achieve higher security utilities. In the Figs. 6.4a and 6.4b, the plots of *HSMS* depict baseline NSUs obtained by assigning for all messages with the minimum allowable security strengths.



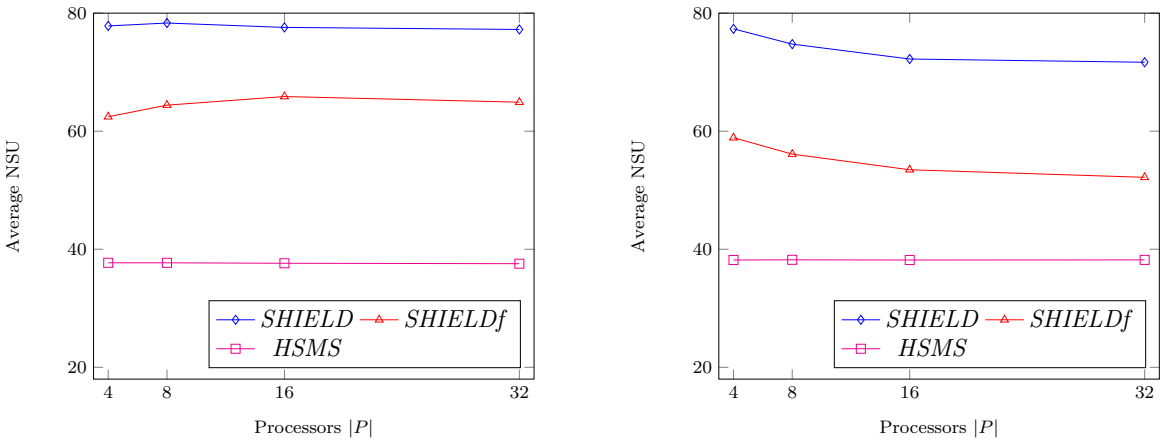
(a) Gaussian Elimination (b) Cybershake

Figure 6.4: *Average NSU for varying Deadline.*

6.4.3.2 Experiment-2: Effect of variation in #processors

Through this experiment, we measure average NSUs against variation #processors, while fixing the values for ∂, CCR, β and $|V|$ to 1.2, 0.5, 0.75 and 150. Obtained results for both Gaussian Elimination and Cybershake are presented in Fig. 6.5. It can be observed that *SHIELD* produces consistently better results in comparison to *SHIELDf* and *HSMS*,

for all the data points. As an example of *SHIELD's* performance, for $|P| = 16$, the average NSU of *SHIELD* is higher than *SHIELDf* and *HSMS* by ∼15% and ∼52% for Gaussian Elimination. Similarly for Cybershake, *SHIELD* performs better by ∼26% and ∼47% w.r.t. *SHIELDf* and *HSMS*.



(a) Gaussian Elimination (b) Cybershake

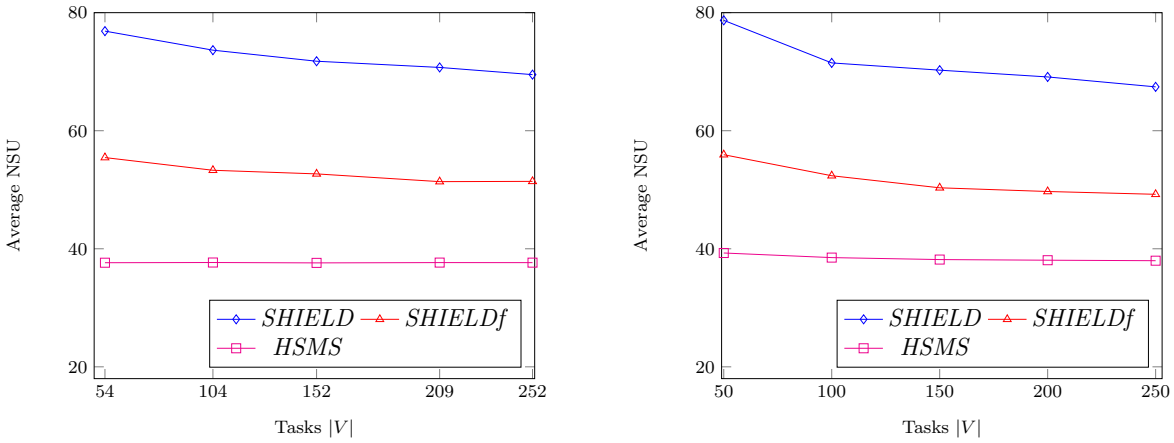
Figure 6.5: *Average NSU for varying number of processors.*

6.4.3.3 Experiment-3: Effect of variation in varying tasks

This experiment measures the *Normalised Security Utility* (NSU) (Fig. 6.6) and *run-times* (Table 6.5) of different algorithms, as the number of tasks is varied from 50 to 252. In this, the parameters ∂, $|P|$, CCR and β are fixed at 1.2, 32, 0.5 and 0.75. Fig. 6.6 shows the variation in average NSU for different number of tasks. For any fixed value of #tasks, it can be observed from the figure that for both the benchmarks *SHIELD* consistently achieves a higher NSUs compared to *SHIELDf* and *HSMS*.

Table 6.5: *Run-times (ms) for varying #tasks using Gaussian Elimination*

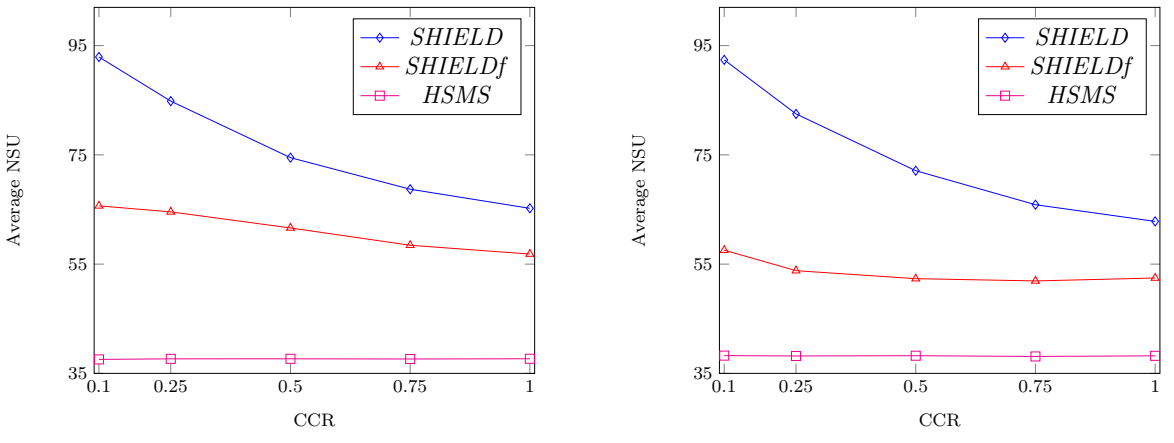
$\|V\|$	54	104	152	209	252
SHIELD	1.14	8.03	22.04	50.54	86.42
SHIELDb	4.61	21.18	53.76	124.56	206.21
SHIELDf	1.97	8.69	22.20	52.54	91.45
HSMS	1.24	3.93	7.92	14.84	21.47



(a) Gaussian Elimination (b) Cybershake

Figure 6.6: *Average NSU for varying number of tasks.*

Table 6.5 depicts average run-time as #task is varied from 50 to 252. It can be observed that, run-time increases with increase in the number of tasks, for all the algorithms; *SHIELD* runs faster compared to *SHIELDb* and *SHIELDf*. As an example, we see from Table 6.5 that for 252 tasks, *SHIELD* runs ∼58% faster than *SHIELDb*. For all the benchmarks, the run-times of *HSMS* are upper bounded by ∼22 *ms*. The results for Cybershake have not been presented as they exhibit trends very similar to Gaussian Elimination.



(a) Gaussian Elimination (b) Cybershake

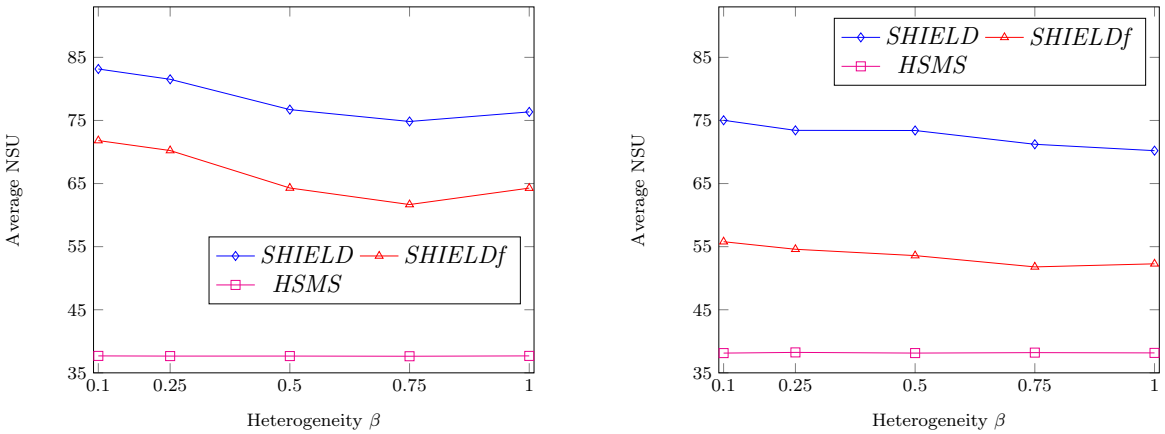
Figure 6.7: *Average NSU for varying CCR.*

6.4.3.4 Experiment-4: Effect of variation in CCR

This experiment measures NSUs achieved by the schedulers *HSMS*, *SHIELD* and *SHIELDf* as communication-to-computation ratios is varied from .1 to 1. The values for ∂, $|P|$, β, $|V|$ are set to 1.2, 32, 0.75, 150 respectively. For both the benchmarks, it may be seen from Fig. 6.7 that the average NSUs achieved by the algorithms decrease with increasing CCR. As relative communication data sizes increase with higher CCR values, any enhancement in the security strength will incur relatively higher security overheads at the source and destination processor of the messages.

6.4.3.5 Experiment-5: Effect of variation in heterogeneity

Through this experiment, we have analysed the average NSUs achieved by *HSMS*, *SHIELD* and *SHIELDf* for varying the degrees of processor heterogeneity. Parameters ∂, $|P|$, CCR and $|V|$ are fixed at 1.2, 32, 0.5 and 150. As an example of *SHIELD's* performance, it may be observed in Fig. 6.8a (Gaussian Elimination) that for $|\beta| = 0.75$, the average normalized security utility of *SHIELD* is higher than *SHIELDf* by about ∼18%. Similarly for Cybershake (Fig. 6.8b), *SHIELD* outperforms *SHIELDf* by ∼27%.



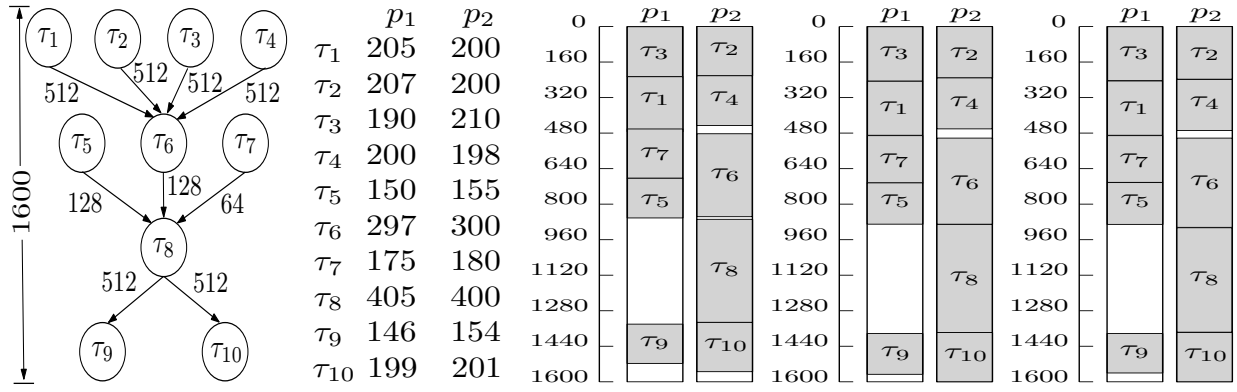
(a) Gaussian Elimination

(b) Cybershake

Figure 6.8: *Average NSU for varying Heterogeneity.*

6.5 Case Study: Traction Control System

To exhibit the practical adaptability of *SHIELD* to real-world settings, we present a case study on the *Traction Control (TC)* application in automotive systems. TC aims to improve a car's stability when road conditions are slippery [69]. Fig. 4.7a depicts the layout diagram of TC as adapted from [41] and Fig. 6.9a displays its corresponding DAG model. This DAG consists of 10 nodes that need to be scheduled on a distributed platform having two processors $\{p_1, p_2\}$. The communication bandwidths between the processors are considered to be 500 KB/s. The end-to-end application deadline is assumed to be 1600 *ms*. Table 6.6 depicts the minimum security demands and relative priority for three security service types, associated with each message of the TC DAG in Fig. 6.9a. Table 6.3 lists the performance of alternative security service protocols on the available processors $\{p_1, p_2\}$. Fig 6.9b depicts the worst case execution times of each task associated with TC on the two processors.



(a) *TC DAG* **(b)** *TC WCET* **(c)** *HSMS* **(d)** *SHIELD* **(e)** *SHIELDf*

Figure 6.9: *Traction Control (TC): (a) TC DAG; (b) Execution times (in ms) of tasks in TC DAG on two processors; Gantt charts depicting the schedules: (c) HSMS (makespan* = 1556, T_{SU} = 3.84*); (d) SHIELD (makespan* = 1600, T_{SU} = 7.0*); (e) SHIELDf (makespan* = 1600, T_{SU} = 4.89*), for the DAG in Fig. 6.9a.*

Employing *HSMS*, *SHIELD* and *SHIELDf*, we generate three separate schedules for the TC application, and the respective Gantt charts are presented in Figs. 6.9c, 6.9d and 6.9e. It may be observed that *SHIELD* is able to achieve higher security utility ($T_{SU} = 7.0$) compared to *SHIELDf* ($T_{SU} = 4.89$) and *HSMS* ($T_{SU} = 3.84$), while satisfying the stipulated deadline for the traction control application.

Table 6.6: *Messages security requirements for the TC DAG in Fig. 6.9a*

Messages	$s_{i,j}^{1,\min}$	$s_{i,j}^{2,\min}$	$s_{i,j}^{3,\min}$	$w_{i,j}^{1}$	$w_{i,j}^{2}$	$w_{i,j}^{3}$
$data_{1,6}$	0.2	0.1	0.4	0.3	0.3	0.4
$data_{2,6}$	0.2	0.2	0.4	0.3	0.5	0.2
$data_{3,6}$	0.2	0.5	0.3	0.2	0.6	0.2
$data_{4,6}$	0.3	0.4	0.2	0.2	0.2	0.6
$data_{5,8}$	0.4	0.3	0.1	0.2	0.3	0.5
$data_{6,8}$	0.4	0.2	0.4	0.7	0.1	0.2
$data_{7,8}$	0.4	0.1	0.3	0.7	0.1	0.2
$data_{8,9}$	0.3	0.1	0.2	0.7	0.1	0.2
$data_{8,10}$	0.3	0.2	0.1	0.7	0.1	0.2

6.6 Summary

In this chapter, we have presented a low-overhead security-aware real-time list scheduling algorithm named *SHIELD* for DAG-structured applications on distributed heterogeneous systems. *SHIELD* first calls *HSMS* to generate a *makespan* minimizing schedule while satisfying the minimum security demands of each message in the application. *SHIELD* returns with failure if the *makespan* generated by *HSMS* violates the given deadline. Otherwise, *SHIELD* attempts to enhance the security strengths of all messages such that the total security utility of the system is maximized. Experimental evaluation using two benchmark task graphs: *Gaussian Elimination* and *Cybershake*, reveal that *SHIELD* significantly outperforms greedy baseline strategies *SHIELDb* in terms of solution generation times (runtimes) and *SHIELDf* in terms of achieved security utility, over various input test scenarios. Finally, the practical applicability of *SHIELD* is exhibited through a case study on the *Traction Control* application in automotive systems.

In the next chapter, we develop a mechanism to construct temperature-aware minimum *makespan* schedules for applications modeled as DAGs with known thermal characteristics on a heterogeneous processing platform.

Chapter 7

TMDS: A Temperature-aware Makespan Minimizing DAG Scheduler for Heterogeneous Systems

7.1 Introduction

The introduction of sub-micron VLSI advancements has led to highly dense multi-million gates per chip, where power dissipation rates and thermal management have become critical design issues [82]. Unconfined surges in temperature not only increase cooling costs but may also reduce system performance and life expectancy. Authors in [107] have shown that a chip's life span could be decreased by up to 50% with a $10°C - 15°C$ temperature rise beyond normal operating temperature. Therefore, modern multi-core embedded systems generally come with a specified temperature threshold that must be adhered to for safe and efficient system operation. *Dynamic Thermal Management* (DTM) techniques that enforce performance throttling to alleviate temperature hotspots, are set off whenever the operating temperature exceeds the predefined threshold temperature in the system [52]. The DTM triggers lead to operations like powering-down processors [98], clock or fetch gating [81], dynamic voltage and frequency scaling (DVFS) [99], etc. Almost all modern processors manage these methods at the hardware level having multiple power domains and operating at different frequency/voltage settings. However, the DTM techniques typically focus solely on the lowering of chip temperature, thus impacting application performance leading to the delivery of degraded Quality of Service (QoS). Therefore, there is a need for platform resource man-

agers which conduct task-to-processor assignments with suitably tuned configurations such that system-level temperature requirements are met. *Given a DAG-structured application with known thermal characteristics on a heterogeneous processing platform, successfully satisfying execution requirements of application tasks while ensuring that all processors operate within a specified temperature threshold is ultimately a scheduling problem.*

The contributions of this chapter are summarized as follows:

1. We proposed a generic temperature management strategy which can be easily employed to adapt existing state-of-the-art *makespan* minimizing DAG scheduling algorithms so that schedules generated by them never violate the processors' threshold temperatures.

2. The temperature-aware DAG scheduling problem has been mathematically formulated as a constrained optimization problem, whose solution is shown to be prohibitively expensive in terms of computational overheads.

3. Based on insights observed through the constraint optimization formulation, we endeavor to design a low-overhead list-based heuristic algorithm called, "*Temperature-aware Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems (TMDS)*", for the problem at hand.

4. Experimental evaluation using real-world benchmarks shows that the *TMDS* delivers lower schedule lengths compared to temperature-aware version of four *makespan* minimizing algorithms, *HEFT* [85], *PEFT* [5], *PPTS* [24] and *PSLS* [112], over various test scenarios.

5. A case study on an adaptive cruise controller in automotive domains is used to exhibit the relevance of *TMDS* in realistic environments.

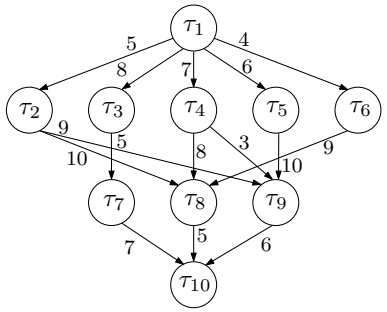
The rest of the chapter is organized as follows. Section 7.2 presents system models and Section 7.3 describes the problem formulation. Section 7.4 discusses the proposed scheduling policy. Section 7.5 presents the detailed experimental results. Section 7.6 exhibits a real-world case study on an automotive control system. Finally, the conclusion of this chapter is presented in Section 7.7.

7.2 System Models

In this section, we present the application and platform model, followed by temperature model.

7.2.1 Application and Platform Model

Directed Acyclic Graph (DAG) $G(V, E)$ is a common way to represent an application where the set of vertices $V = \{\tau_1, \tau_2..., \tau_{|V|}\}$ denotes tasks and the set of directed edges E represents the precedence-constraints between task pairs. Each edge $e_{i,j}$ is marked with a positive weight $data_{i,j}$ indicating the size of the message to be transmitted.



(a) A DAG $G(V, E)$

Tasks	p_1	p_2	p_3
τ_1	16	15	14
τ_2	16	17	18
τ_3	18	16	13
τ_4	19	10	17
τ_5	19	18	20
τ_6	15	16	13
τ_7	20	18	12
τ_8	16	12	14
τ_9	20	19	17
τ_{10}	15	12	7

(b) WCET Table

Figure 7.1: *(a) A DAG with* 10 *tasks; (b) WCETs of* 10 *tasks on* 3 *processors.*

The platform $P = \{p_1, p_2, ..., p_{|P|}\}$ consists of $|P|$ fully interconnected heterogeneous processors. On these heterogeneous processors, each task possibly has a different worst case execution time ($WCET$) on different processors. For example, the WCET of each task of the DAG in Fig. 7.1a on three heterogeneous processors are shown in Fig. 7.1b. An element $\omega_{j,n}$ stores the WCET of task τ_j on p_n. A matrix B of size $|P| \times |P|$ is defined to store heterogeneous bandwidths of links between all pairs of processors. An element $b_{m,n} \in B$ represents the data transfer rate between processors p_m and p_n. As the communication links are bidirectional, i.e., $b_{m,n}$ equals to $b_{n,m}$. The communication time between tasks τ_i (executing on processor p_m) and τ_j (executing on p_n; $\tau_j \in succ(\tau_i)$) can be determined as:

$$c_{i,j}^{m,n} = \begin{cases} 0, & if\, m = n \\ data_{i,j}/b_{m,n}, & otherwise \end{cases} \tag{7.1}$$

The above equation reveals that the communication time is considered to be negligible when both the tasks τ_i and τ_j are allocated to the same processor.

7.2.2 Temperature Model

The rate of change in temperature (in $^\circ C$) of a processor is modeled similar to [34, 57]:

$$\frac{d\ \Gamma^t}{dt} = \frac{pow^t}{C} - \frac{\Gamma^t - \Gamma^a}{C\ R} \tag{7.2}$$

where (i) Γ^t is the processor's temperature at time instant t, (ii) Γ^a is the ambient temperature, (iii) pow^t represents the power dissipation (in W) at time t, (iv) C and R indicate the thermal capacitance (in $J/^\circ C$) and resistance (in $^\circ C/W$) of a processor, respectively. Scaling Γ^t such that Γ^a is $zero$, equation 7.2 gets reduced to equation 7.3.

$$\frac{d\ \Gamma^t}{dt} = a\ pow^t - b\ \Gamma^t \tag{7.3}$$

where $a = 1/C$ and $b = 1/CR$. Here, pow^t comprises of two factors: dynamic power dissipation (pow^d) which is independent of temperature and the leakage power dissipation (pow^l) which is dependent on the processor's temperature [34, 57, 96].

$$pow^t = pow^d + pow^l = C_0 f^3 + (C_1 f + C_2 f\ \Gamma^t) \tag{7.4}$$

where C_0, C_1, and C_2 are constants that are dependent on processor's frequency f (proportional to the voltage [55]). Here, we assume that each processor can run only at a single frequency say, f. Plugging equation 7.4 into equation 7.3, the rate of change in processor temperature can be formulated as:

$$\frac{d\ \Gamma^t}{dt} = a[C_0 f^3 + C_1 f + C_2 f\ \Gamma^t] - b\ \Gamma^t = \mathcal{A} - \mathcal{B}\Gamma^t \tag{7.5}$$

where $\mathcal{A} = a(C_0 f^3 + C_1 f)$ and $\mathcal{B} = (b - aC_2 f)$. For a time interval $[t^0, t^e]$ in which task τ_j is executing on processor p_n, if the processor temperature at the start of the interval is Γ^0 and Γ^e at the end of the interval, then equation 7.5 can be rewritten as:

$$\Gamma^e = \mathcal{A}/\mathcal{B} + \left(\Gamma^0 - \mathcal{A}/\mathcal{B}\right) e^{-\mathcal{B}(t^e - t^0)} \tag{7.6}$$

We now define a function $T_{heat}()$ to determine the temperature Γ^e of a processor p_n at time t^e when a task τ_j continually executes on it starting from time t^0 with temperature Γ^0.

$$T_{heat}(\Gamma^{ss}_{j,n}, \Gamma^0, t^e - t^0) = \Gamma^{ss}_{j,n} + (\Gamma^0 - \Gamma^{ss}_{j,n}) e^{-\mathcal{B}_n (t^e - t^0)} \tag{7.7}$$

where $\mathcal{B}_n = (1 - RC_2 f)/CR$, represents the thermal characteristics of processor p_n and $\Gamma^{ss}_{j,n}$ is the steady state temperature of a task τ_j on p_n. $\Gamma^{ss}_{j,n}$ denotes the temperature that a

processor may reach when τ_j executes continuously on p_n for a sufficiently long time. Thus, $\Gamma^{ss}_{j,n} = \Gamma^a + \mathcal{A}_n/\mathcal{B}_n$. A very similar function can be defined to obtain the temperature of p_n when it remains idle for a time interval $[t^0, t^e]$, starting from its initial temperature Γ^0 at time t^0.

$$T_{cool}(\Gamma^a, \Gamma^0, t^e - t^0) = \Gamma^a + (\Gamma^0 - \Gamma^a)e^{-\mathcal{B}_n(t^e - t^0)} \tag{7.8}$$

Some of the important notations and their meanings have been listed in Table 7.1.

Table 7.1: *List of important notations used and their meanings*

Symbols	Descriptions	Acronyms	Full Forms
τ_j	j^{th} task	DTM	Dynamic Thermal Management
p_n	n^{th} processor	WCET	Worst Case Execution Time
$\omega_{j,n}$	WCET of task τ_j on processor p_n	EST	Effective Start Time
t	Current time	EFT	Effective Finish Time
Γ^a	Ambient room temperature	AST	Actual Start Time
Γ^0	Initial temperature	AFT	Actual Finish Time
Γ^q	Final temperature	OFT	Optimistic Finish Time
$\Gamma^{ss}_{j,n}$	Steady state temperature of task τ_j on processor p_n	TETT	Task Execution Time and
$\Gamma^{st}_{j,n}$	Starting temperature of task τ_j on processor p_n		final Temperature
$\mathcal{B}_n$	Thermal characteristics of processor p_n	MDE	Maximum Duration of
Γ^{lim}_n	Upper bound on temperature associated with p_n		continuous Execution
$\Gamma^c_{j,n}$	Cutoff temperature of τ_j on p_n	SCT	Schedule Completion Time
Γ^t_n	Temperature of a processor p_n at any time step t	OV	Overhead
$\Gamma^t_{j,n}$	p_n's temperature at time t when τ_j is mapped on it	SLR	Schedule Length Ratio
T	Upper bound on possible schedule length	CCR	Communication-to-
$\xi_{j,n}$	Duration in which task τ_j remains allocated on p_n		Computation Ratio
S_j	Start time of task τ_j	ACC	Adaptive Cruise Controller

Problem Statement: *The objective of this work is to design a DAG scheduling algorithm which attempts to minimize makespan, while ensuring that processor temperatures never overshoot their respective threshold cutoff values. In systems where the steady state temperatures of one or more tasks (on some or all processors) are higher than their threshold*

cutoffs, the above objective is achieved by opportunistically inserting idle intervals (to allow processor cooling) within the schedule.

7.3 Constraint Satisfaction Problem Formulation

In this section, we formally model the above-discussed problem as a *Constraint Optimization Problem*. The formulation uses two sets of decision variables: (i) $X = \{x^t_{j,n} : j = 1, 2, \ldots, |V|; n = 1, 2, \ldots, |P|; t = 0, 1, \ldots, T\}$. Here, $T = \sum_{j=0}^{|V|} \max_{p_n \in P} \omega_{j,n}$ denotes the upper bound on possible schedule length. The term $\omega_{j,n}$ denotes the WCET of a task τ_j on a processor p_n and is defined as the maximum time taken when the task runs standalone in a control setup when the processor temperature can never exceed its stipulated threshold upper bound. $x^t_{j,n} = 1$, indicates that task τ_j is mapped to processor p_n and starts execution from the t^{th} time step; otherwise, $x^t_{j,n} = 0$, and (ii) $Z = \{z^\alpha_{j,n} : j = 1, 2, \ldots, |V|; n = 1, 2, \ldots, |P|; \alpha = 0, 1, \ldots, T\}$. $z^\alpha_{j,n} = 1$ indicates that task τ_j should be executing on processor p_n at time step α subsequent to the start of τ_j; $z^\alpha_{j,n} = 0$ means one of two possible cases: (i) Processor p_n is idling with task τ_j is allocated on p_n. This case happens when $\alpha < \xi_{j,n}$. Here, $\xi_{j,n} = \max_{\alpha=1}^{T} \alpha \; z^\alpha_{j,n}$. The value of $\xi_{j,n}$ represents the duration during which task τ_j remains allocated on processor p_n. (ii) Task τ_j is not allocated on p_n. This happens when $\alpha > \xi_{j,n}$. Next, we present the objective function and the necessary constraints on the binary decision variables to model the scheduling problem.

Objective Function: Our objective is to minimize the overall schedule length while satisfying all scheduling constraints. The objective function can be defined as:

$$Minimize \sum_{n=1}^{|P|} \sum_{t=0}^{T} x^t_{exit,n} \times (t + \xi_{exit,n}) \tag{7.9}$$

Subject to satisfaction of the constraints in equations 7.10 - 7.14.

Unique Start Time Constraints: The start time of each task on a specific processor should be unique; that is,

$$\forall j \in [1, |V|], \sum_{n=1}^{|P|} \sum_{t=0}^{T} x^t_{j,n} = 1 \tag{7.10}$$

The above equation ensures that each task τ_j must start its execution at a unique time step t exactly in one processor p_n. The start time of τ_j may be obtained as $S_j = \sum_{t=0}^{T} t \times x^t_{j,n}$.

Dependency Constraints: This enforces the satisfaction of precedence relationships among tasks within a DAG. That is, the execution start time of a task τ_j must be greater or

equal to the arrival time of output messages among all the predecessors of τ_j. For any given predecessor τ_i, its arrival time is determined as the summation of τ_i's execution completion time (that includes cooling time) and the data transmission time from τ_i to τ_j.

$$\forall(\tau_i,\tau_j)\in E,\ \sum_{n=1}^{|P|}\sum_{t=0}^{T} t\times x_{j,n}^{t} \geqslant \sum_{n=1}^{|P|}\sum_{t=0}^{T}\sum_{m=1}^{|P|}\sum_{t'=0}^{T} x_{i,m}^{t'}\times x_{j,n}^{t}(t'+\xi_{i,m}+c_{i,j}^{m,n}) \tag{7.11}$$

Resource Constraints: Resource constraints enforce that a processor p_n can be allocated to at most one task at any given time. This constraint may be represented as:

$$\forall n\in[1,|P|],\forall t\in[0,T],\ \lambda_n^t \leqslant 1 \tag{7.12}$$

It may be noted that a task τ_j can only be mapped on p_n for execution at time instant t, if it has started at most '$t-\xi_{j,n}+1$' time steps earlier. In the above equation, $\lambda_n^t = \sum_{j=1}^{|V|}\rho_{j,n}^t$ and $\rho_{j,n}^t = \sum_{l=t-\xi_{j,n}+1}^{t} x_{j,n}^l$. The term $\rho_{j,n}^t$ assumes a value of '1' when τ_j starts on p_n between time intervals $[t-\xi_{j,n}+1,t]$.

Execution Demand Constraints: Over the duration $\xi_{j,n}$, the total number of time slots in which τ_j should be executing on processor p_n must be exactly equal to $\omega_{j,n}$. That is:

$$\forall j\in[1,|V|],\forall n\in[1,|P|],\sum_{\alpha=1}^{\xi_{j,n}} z_{j,n}^{\alpha} = \omega_{j,n} \tag{7.13}$$

Processor Temperature Constraints: The operating temperature (Γ_n^t) of a processor p_n at any time step t should never exceed a stipulated upper bound on temperature (Γ_n^{lim}) associated with p_n. That is,

$$\forall t\in[0,T],\forall n\in[1,|P|],\ \Gamma_n^t \leqslant \Gamma_n^{lim} \tag{7.14}$$

Here, Γ_n^t is determined as follows:

$$\Gamma_n^t = \begin{cases} \Gamma^a\ , & if\ t=0 \\ \Gamma^a + (\Gamma_n^{t-1}-\Gamma^a)e^{-\mathcal{B}_n}, & if\ t>0\ \&\ \lambda_n^t=0 \\ \sum_{j=1}^{|V|}\Gamma_{j,n}^t\ , & otherwise \end{cases} \tag{7.15}$$

In the above equation, Γ^a represents the ambient room temperature. If no task is executing (indicated by $\lambda_n^t=0$) on processor p_n at any time t (>0), Γ_n^t is represented by the second line of equation 7.15. Given Γ_n^{t-1} ($<\Gamma_{j,n}^{ss}$), the temperature of processor p_n at time $t-1$, the expression in the right-hand side (RHS) produces the temperature Γ_n^t of p_n when it idles

(thus cools-off) at the t^{th} time slot (refer equation 7.8). In the scenario when some task τ_j is allocated on p_n, two possible cases may arise: (i) p_n is executing τ_j, or (ii) p_n remains idle (for cool-off). In the third line of equation 7.15, $\Gamma^t_{j,n}$ indicates the temperature of p_n at time t when task τ_j is mapped on it. equation 7.16 below determines $\Gamma^t_{j,n}$ as:

$$\Gamma^t_{j,n} = \begin{cases} 0, & \text{if } \rho^t_{j,n} = 0 \\ \{\Gamma^a + (\Gamma^{t-1}_n - \Gamma^a)e^{-\mathcal{B}_n}\} \times (1 - z^\alpha_{j,n}) \ + \{\Gamma^{ss}_{j,n} + (\Gamma^{t-1}_n - \Gamma^{ss}_{j,n})e^{-\mathcal{B}_n}\} \times z^\alpha_{j,n}, & \text{otherwise} \end{cases} \tag{7.16}$$

The first line in the above equation says that $\Gamma^t_{j,n}$ assumes the value *zero*, when τ_j is not mapped on p_n (in this case, $\rho^t_{j,n} = 0$). When τ_j is mapped on p_n (2^{nd} line of equation 7.16), the first expression in the RHS depicts the temperature of processor p_n when τ_j is mapped on it but not executing at time t. The next expression in the RHS denotes the temperature of p_n when τ_j executes on it at time step t. Here, $\alpha = t - S_j$.

Complexity Analysis: The overhead of the above formulation can be analysed in terms of the total number of variables employed considering all constraints of the problem:

- *Unique start time constraints* use $O(|V|)$ constraints and $O(|P| \times T)$ variables per constraint.
- *Dependency constraints* employ $O(|E|)$ constraints and $O(|P| \times T)$ variables per constraint.
- *Resource constraints* use $O(|P| \times T)$ constraints and $O(|V|)$ variables per constraint.
- *Execution Demand Constraints* employ $O(|P| \times |V|)$ constraints and $O(T)$ variables per constraint.
- *Processor temperature constraints* utilize $O(|P| \times T)$ constraints. The number of variables per constraint can be determined (refer equation 7.15; when $t \neq 0$ & $\lambda^t_n \neq 0$) as: For $t = 1$, the number of variables required to compute Γ^t_n is $O(|V|)$. Similarly, for $t = 2$, the number of variables is $O(|V|^2)$, and so on. For $t = T$, the number of variables is $O(|V|^T)$. Thus, the total variables per constraint becomes $O(|V| + |V|^2 + \cdots + |V|^T) = O(\frac{|V|^{T+1} - |V|}{|V|-1}) = O(|V|^T)$.

Finally, the total number of variables in all constraints of the proposed formulation becomes: $O(3 \times |V| \times |P| \times T + |E| \times |P| \times T + |V|^T \times |P| \times T) = O(|V|^T \times |P| \times T)$.

An estimate of the size of the state space associated with the problem may also be derived as follows. To determine an optimal solution there may be a need to explore: (i) all possible orders in which the set of tasks may be considered for possible allocation which is $O(|V|!)$, (ii) given any such order for task allocation, each task may be allocated to any processor; the number of possibility becomes $O(|P|^{|V|})$, (iii) for each mapping of a task on a certain processor an upper bound on the number of processor temperature management schedules are obtained as $O(2^T)$. Hence, an upper bound estimate on the size of the state space may be obtained as $O(|V!| \times |P|^{|V|} \times 2^T)$.

It can be observed from the above analysis that the problem is highly exponential with respect to the number of tasks, processors, and temporal bounds in the system. Therefore, an attempt to optimally solve this scheduling problem through standard off-the-shelf solvers or exhaustive state space search, is prohibitively expensive because of the inherent enormity of the state space. Hence, we have designed a list-based heuristic solution strategy for the problem at hand. This heuristic strategy attempts to deliver good and acceptable solutions while intelligently restricting the solution search space in the following ways:

- The heuristic strategy attempts to solve the overall optimization problem by partitioning it into three separate phases, namely (i) *task prioritization*, (ii) *processor allocation* and (iii) *temperature-aware scheduler*.

- A fast mechanism to obtain a fixed task priority order that may allow the generation of low *makespan* schedules, has been designed as the first phase. This phase is a pre-processing step which is called prior to the commencement of processor allocation and schedule generation for the given set of tasks.

- The second phase determines a suitable processor for the highest priority unallocated task, at a given intermediate point during partial schedule generation. In order to obtain this processor, a temperature-aware evaluation on the allocation-goodness of the selected task on all available processors, is conducted.

- For each processor on which the highest priority task may possibly be assigned, the second phase must construct a temperature-aware schedule corresponding to evaluate the goodness of this tentative task-to-processor allocation. This temperature-aware schedule generation strategy for any given task processor pair, comprises the third phase of the algorithm.

In the next section, we describe in detail this list-based heuristic algorithm called "*Temperature-aware Makespan minimizing DAG Scheduler for Heterogeneous Distributed Systems* (*TMDS*)".

7.4 TMDS: The Proposed Scheduler

The proposed list scheduling algorithm called *Temperature-aware Makespan minimizing DAG Scheduler* (*TMDS*) takes a precedence-constrained task graph $G(V, E)$ with known thermal characteristics and a platform P, as inputs. *TMDS* consists of three phases, namely (i) *task prioritization*, (ii) *processor allocation* and (iii) *temperature-aware scheduler*. Algorithm 16 presents the pseudocode of *TMDS*. Steps 1-3 of function *TMDS()* comprises the *task prioritization phase*, while the *processor allocation phase* consists of steps 5-14. Step 10 of the Algorithm 16 calls function $TETT(\tau_j,\ p_n,\ curT,\ \Gamma_n^{cur})$ (*Task Execution time and final Temperature*; refer Algorithm 17) to generate a tentative temperature management schedule associated with the possible allocation of task τ_j on p_n. In the processor selection phase, the temperature-aware schedule returned by *TETT()* plays a significant role in the determination of the actual processor on which τ_j is allocated. We now discuss these three phases in detail.

7.4.1 Task Prioritization

Step 2 of *TMDS* (refer Algorithm 16) calculates two different parameters (adapted from [73]) namely, (1) *Optimistic Finish Time (OFT)* for each task-processor pair, and (2) A *Rank* value for each task.

OFT$[\tau_j, p_n]$: The value of $OFT[\tau_j, p_n]$ is a lower bound estimate of the total additional time required to complete the schedule when task τ_j is assigned on processor p_n. This duration includes the execution time of task τ_j on p_n along with the maximum time required to schedule all remaining unallocated tasks on the given platform. The sink node (τ_{exit}) has an OFT equal to its execution time since it has no successors. The OFT for all other tasks is calculated recursively from τ_{exit} to τ_j as detailed below:

$$OFT[\tau_j, p_n] = \begin{cases} \omega_{j,n}, & \text{if } \tau_j = \tau_{exit} \\ \max\limits_{\tau_k \in succ(\tau_j)} \big[\min\limits_{p_r \in P} \{OFT[\tau_k, p_r] + \omega_{j,n} + c_{j,k}^{n,r}\} \big], & otherwise \end{cases} \tag{7.17}$$

During the processor allocation phase, this estimated time ($OFT[\tau_j, p_n]$) value is employed to obtain a more informed decision on which processor if allocated to a given task, may

possibly lead to the least makespan (refer function $SCT[\tau_j, p_n]$; equation 7.20).

Rank$[\tau_\mathbf{j}]$: The rank of a task τ_j ($Rank[\tau_j]$) is obtained as the average of its OFTs over all processors. Due to this ranking strategy, given a set of ready tasks, their rank values provide comparative estimates of the amounts of workload that must be completed subsequent to their execution start times. In the endeavor to achieve low makespans, a set of ready tasks is considered for processor allocation in the order of their ranks, so that tasks that need to execute comparatively larger workloads subsequent to their starts may be prioritized.

$$Rank[\tau_j] = \frac{\sum_{n=1}^{|P|} OFT[\tau_j, p_n]}{|P|} \tag{7.18}$$

Step 3 initializes ready list $taskList$ with the entry task τ_{entry} in it.

7.4.2 Processor Allocation

The while loop (steps 5-14) generates a schedule obtained by sequentially scheduling the task with the highest $Ranked$ unscheduled ready task in $taskList$. A task becomes ready when all its predecessors have already been scheduled. After selecting the task τ_j in step 6, *TMDS* determines *Effective Start Time* ($EST[\tau_j, p_n]$; refer equation 7.19), *Task Execution Time and final Temperature* ($TETT(\tau_j, p_n, curT, \Gamma_n^{cur})$: refer temperature-aware scheduler below) and *Schedule Completion Time* ($SCT[\tau_j, p_n]$; refer equation 7.20) with respect to the execution of τ_j on each processor p_n (steps 7-11). **Effective Start Time** (**EST**$[\tau_j, p_n]$): The EST of the entry task τ_{entry} on each processor p_n is $zero$. For all other tasks, the effective start time $EST[\tau_j, p_n]$ of τ_j on p_n is determined as:

$$EST[\tau_j, p_n] = \max\{avail[n], \max_{\tau_i \in pred(\tau_j)} (AFT[\tau_i] + c_{i,j}^{m,n})\} \tag{7.19}$$

where $avail[n]$ stores the earliest time at which processor p_n is available to execute a task and $AFT[\tau_i]$ records the *actual finish time* of τ_j's predecessor task τ_i. The inner $\max_{\tau_i \in pred(\tau_j)} \{\dots\}$ block determines the time at which all outputs from the predecessors of τ_j have arrived at processor p_n.

Step 9 updates Γ_n^{cur} to obtain the temperature of p_n at time $EST[\tau_j, p_n]$ (refer equation 7.8), when the processor remains idle for a time interval $[avail[n], EST[\tau_j, p_n]]$, starting from its initial temperature Γ_n^{cur} at time $avail[n]$. Step 10 calls *TETT()* to determine Effective Finish Time $EFT[\tau_j, p_n]$, a scheduling event queue $Sch_{j,n}$ and the temperature (Γ_n^{fin}) of p_n at time $EFT[\tau_j, p_n]$.

7. TMDS: A TEMPERATURE-AWARE MAKESPAN MINIMIZING DAG SCHEDULER FOR HETEROGENEOUS SYSTEMS

Algorithm 16: $TMDS(G, P)$

Input: *Application DAG* $G(V, E)$, *processor set* P
Output: *A temperature-aware schedule that minimizes makespan*

$\forall_{p_n \in P}$, Set $avail[n] = 0$;
Determine OFTs, $Rank$s of all tasks using equations 7.17, 7.18;
Initialize ready list $taskList = \{\tau_{entry}\}$;
// *Let* Γ_n^{cur} *be a temporary variable which holds the temperature of processor* p_n *at any time during partial schedule generation*
while $taskList \neq \phi$ **do**
    τ_j = Extract the task with highest $Rank$ from ready list $taskList$;
    for *each processor* p_n *in* P **do**
        Compute $EST[\tau_j, p_n]$ using equation 7.19; // *Start time of* τ_j *if selected for execution on* p_n
        $\Gamma_{j,n}^{est} = T_{cool}(\Gamma^a, \Gamma_n^{cur}, EST[\tau_j, p_n] - avail[n])$; // *Tentatively determine temperature of* p_n *at the instant* $EST[\tau_j, p_n]$ $(\Gamma_{j,n}^{est})$, *given: (i) its temperature at* $avail[n]$ (Γ_n^{cur}), *and (ii) that* p_n *idles in the interval* $[avail[n], EST[\tau_j, p_n]]$
        $\langle EFT[\tau_j, p_n], Sch_{j,n}, \Gamma_n^{fin} \rangle = TETT(\tau_j, p_n, EST[\tau_j, p_n], \Gamma_{j,n}^{est})$; // *The TETT() function returns: (i) effective finish time* $(EFT[\tau_j, p_n])$ *of* τ_j *on* p_n, *(ii) Schedule* $(Sch_{j,n})$ *of execution and cooling intervals w.r.t. task* τ_j *on* p_n *and (iii) temperature of* p_n *at* $EFT[\tau_j, p_n]$
        $SCT[\tau_j, p_n] = EFT[\tau_j, p_n] + OFT[\tau_j, p_n] - \omega_{j,n}$; // *Estimate schedule completion time of* τ_j *on* p_n
    Determine: $p_{n'} = \min_{p_r \in P} SCT[\tau_j, p_r]$; // *Find the processor* $p_{n'}$ *for which* $SCT[\tau_j, p_{n'}]$ *is minimum*
    Assign τ_j on processor $p_{n'}$;
    Set $AFT[\tau_j] = avail[n'] = EFT[\tau_j, p_{n'}]$, $\Gamma_{n'}^{cur} = \Gamma_{n'}^{fin}$; // *Update: (i) actual finish time of* τ_j $(AFT[\tau_j])$, *(ii)* $avail[n']$ *and (iii) temperature* $\Gamma_{n'}^{cur}$ *of* p'_n *at* $avail[n']$ Insert all ready to execute successor tasks of τ_j in $taskList$;

Effective Finish Time (**EFT**$[\tau_j, p_n]$): $EFT[\tau_j, p_n]$ is the effective execution finish time of task τ_j on p_n. The effective time at which τ_j can start execution on p_n is given by $EST[\tau_j, p_n]$ (refer equation 7.19). This time instant ($EST[\tau_j, p_n]$) is marked by a distinct temperature value of p_n, which may be the ambient temperature (if p_n has been idle for a long time) or a higher temperature (due to earlier task executions on p_n). Starting from this initial temperature at $EST[\tau_j, p_n]$, τ_j may be able to complete its execution demand $\omega_{j,n}$ in one continuous interval or may need one or more intervening cooling intervals (in order to prevent p_n from breaching the threshold temperature Γ_n^{lim}). Thus in general, the total duration over which τ_j executes on p_n is characterized by alternating sub-phases of execution and cooling, such that the aggregate length of these execution sub-phases is equal to $\omega_{j,n}$. Thus, the value of $EFT[\tau_j, p_n]$ is obtained by adding $EST[\tau_j, p_n]$ with this total duration over which τ_j executes on p_n.

Scheduling event queue ($Sch_{j,n}$): $Sch_{j,n}$ is an ordered event queue marking all the time instances corresponding to the start of execution and cool-off sub-phases within the total duration over which τ_j remains allocated on p_n for completing its execution demand ($\omega_{j,n}$). Corresponding to our running example (refer to the paragraph "Example continued" below), Let $Sch_{7,3} = \langle 47.00, 55.64, 56.09, 56.70, 57.15, 57.76, 58.21, 58.82, 59.27, 59.88, 60.33, 60.94, 61.29, 61.60\rangle$ represents the event queue related to the execution of task τ_7 on processor p_3. The schedule Gantt chart produced by this event queue may be seen in Fig. 7.4 and reproduced here in Fig. 7.2, for convenience. It may be observed that this Gantt chart has seven alternating phases of execution and cooling. Among these, $\omega^1_{7,3}(= 55.64 - 47) = 8.64$, $\omega^2_{7,3} = \omega^3_{7,3} = \omega^4_{7,3} = \omega^5_{7,3} = \omega^6_{7,3} = 0.61$ and $\omega^7_{7,3} = 0.31$, denote the execution sub-phases. The first phase starts at $EST[\tau_7, p_3] = 47.00$. The WCET of task τ_7 on p_3 is obtained as $\omega_{7,3} = \sum_{k=1}^{7} \omega^k_{7,3} = 12$. The total duration over which τ_7 remains allocated on p_3 for completing its execution demand ($\omega_{7,3} = 12$) is: 14.60. Hence, $EFT[\tau_7, p_3] = 47.00 + 14.60 = 61.60$.

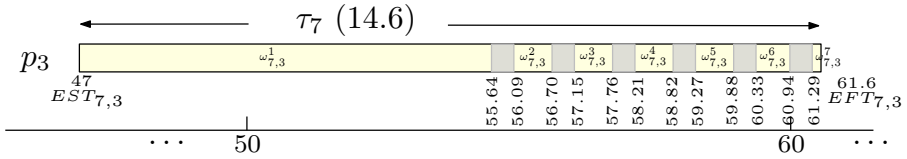


Figure 7.2: *$Sch_{7,3}$ - schedule of task τ_7 on processor p_3*

Step 11 computes the Schedule Completion Time ($SCT[\tau_j, p_n]$) of task τ_j on p_n.

Schedule Completion Time (**SCT**$[\tau_j, p_n]$): Given that τ_j is allocated on p_n, $SCT[\tau_j, p_n]$ is computed as:

$$SCT[\tau_j, p_n] = OFT[\tau_j, p_n] + EFT[\tau_j, p_n] - \omega_{j,n} \tag{7.20}$$

The R.H.S. of the above expression can be viewed as a combination of two terms: (i) The first term $EFT[\tau_j, p_n]$ is the effective execution finish time of task τ_j on p_n. (ii) The second term $OFT[\tau_j, p_n] - \omega_{j,n}$ essentially provides an estimate of the total time required to complete the execution of all dependant tasks of τ_j, assuming τ_j to be allocated for execution on processor p_n.

Finally in steps 12-13, each selected task τ_j is assigned to that processor p_n for which $SCT[\tau_j, p_n]$ is minimal. After τ_j is scheduled on p_n, EST and EFT of τ_j on p_n become the *Actual Start Time* $AST[\tau_j]$ and *Actual Finish Time* $AFT[\tau_j]$ of task τ_j. Step 14 updates the finish time of each task and captures the current temperature of p_n as well as updates the *avail* time of processor p_n. *Makespan* of the generated schedule is equal to the finish time $AFT[\tau_{exit}]$ of the sink task τ_{exit}.

7.4.3 Temperature-aware Scheduler

The processor allocation phase calls function $TETT()$ for each possible processor on which the currently highest priority task τ_j can be assigned. *TETT()*'s objective is to generate a minimal length non-preemptive execution schedule for τ_j while ensuring that the temperature of p_n is always maintained below its stipulated threshold value. To achieve this, the temperature-aware scheduler determines the intervals within the execution schedule during which τ_j actually executes on p_n. The processor is kept idle in order to enable cool-off, in the remaining intervals. However, each transition from the *active* to *idle* state of a processor (or *vice versa*), has an overhead (OV) associated with the *saving* (or *restoration*) of the executing task's intermediate state. Over diverse embedded processing platforms having varying capabilities, this overhead may differ from about $\approx 10\mu s$ up to $\approx 150\mu s$ [43,46,110]. Throughout the remaining chapter, we have considered the value of OV to be $= 80\mu s$ [43].

TETT() adheres to two design principles for minimizing the total execution length of τ_j on p_n in the presence of OV, the *active-to-idle* transition overhead. (i) An interval in which task τ_j actually executes on processor p_n is never interrupted/terminated until the threshold temperature of p_n is reached, (ii) Whenever the temperature of p_n reaches its threshold value, it is switched to idle mode. Thereafter, p_n continuously cools-off in this

idle state for a minimum duration which either allows: (a) continuous execution of τ_j up to completion (while not violating p_n's threshold) in the execution interval which follows this cool-off interval, or (b) cool down to the ambient room temperature. If τ_j's execution time is relatively high, or if p_n's threshold temperature is low, multiple idle cool-off intervals may be required before τ_j is able to complete its execution. In all these intermediate cool-off intervals (except the final interval), p_n is forced to idle until ambient room temperature is reached. We now show that this obvious cool down strategy may have a very adverse effect on *TETT()*'s objective towards achieving a minimum length schedule for τ_j.

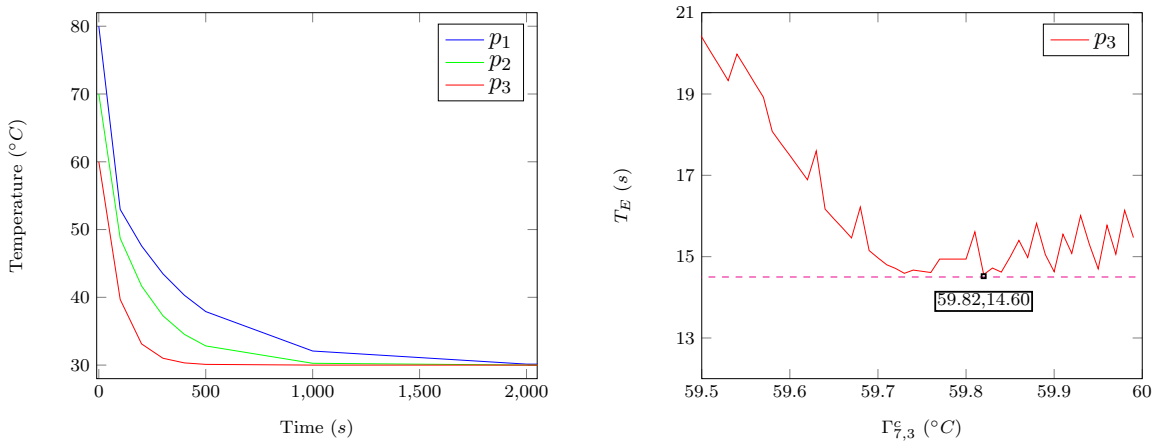
Table 7.2: *Thermal parameters of processors [57, 71]*

	f	C_0	C_1	C_2	R	C	Γ^{lim}	Γ^{cur}	Γ^{a}
p_1	2.6	2.332	13.1568	0.1754	0.68	380	$80°C$	$30°C$	$25°C$
p_2	3.4	2.138	5.0187	0.1942	0.487	295	$70°C$	$50°C$	$25°C$
p_3	3.0	4.556	15.6262	0.1942	0.238	320	$60°C$	$40°C$	$25°C$

Fig. 7.3a shows the cooling characteristic curves plotted using equation 7.8, for three heterogeneous processors with different thermal properties, as detailed in Table 7.2. It can be observed from Fig. 7.3a that a processor p_n must suffer an exponentially long duration of idleness if it is forced to cool down from the threshold temperature (Γ_n^{lim}) to a *cutoff* temperature which is close to the ambient room temperature (Γ^a). On the other hand, if the *cutoff* temperature is too close to the threshold Γ_n^{lim}, p_n can not cool down sufficiently so that adequate execution progress can be made before temperature reaches Γ_n^{lim} again (when p_n must transit back to idle mode). Thus, we see that the total duration of execution of a task τ_j on p_n gets elongated (beyond an optimal value) both when the *cutoff* ($\Gamma_{j,n}^c$) temperature is very close to either Γ_n^{lim} or Γ^a. This indicates the existence of one or more optimal values of $\Gamma_{j,n}^c$ (which we refer to as $\Gamma_{j,n}^{oc}$; $\Gamma_n^{lim} > \Gamma_{j,n}^{oc} > \Gamma^a$) for which the total execution duration of τ_j on p_n gets minimized.

As discussed above, the temperature of a processor p_n (on which task τ_j is considered for assignment) at the beginning of τ_j's execution (at the instant $EST[\tau_j, p_n]$) is obtained as Γ_n^{est} in step 9 of Algorithm 16. Now, the total execution length of τ_j on p_n ($T_E(\tau_j, p_n)$) can be expressed as:

$$T_E(\tau_j, p_n) = MDE(\tau_j, p_n, \Gamma_{j,n}^{est}) + \lfloor I \rfloor \times [coolT(\Gamma_{j,n}^c, \Gamma_n^{lim}) + MDE(\tau_j, p_n, \Gamma_{j,n}^c)] + remET + coolT(\Gamma_{j,n}^{cur}, \Gamma_n^{lim}) + 2 \lceil I \rceil \times OV \quad (7.21)$$



(a) Cooling characteristic curves

(b) τ_7 execution lengths

Figure 7.3: *(a) Cooling characteristic curves of processors w.r.t. time; (b) Task execution lengths w.r.t. cutoff temperatures.*

The above equation can be explained by describing the important terms in its R.H.S. Starting with the initial temperature $\Gamma^{est}_{j,n}$, $MDE(\tau_j, p_n, \Gamma^{est}_{j,n})$ (*MDE: Maximum Duration of continuous Execution*) represents the execution progress that τ_j makes by continuously running on p_n until the temperature reaches p_n's threshold value Γ^{lim}_n. The function $MDE()$ can be derived from equation 7.7 above, as:

$$MDE(\tau_j, p_n, \Gamma^{ini}) = \begin{cases} \infty, & if\ \Gamma^{ss}_{j,n} \leqslant \Gamma^{lim}_n \\ \frac{-1}{\mathcal{B}_n} \times log(\frac{\Gamma^{lim}_n - \Gamma^{ss}_{j,n}}{\Gamma^{ini} - \Gamma^{ss}_{j,n}}), & otherwise \end{cases} \tag{7.22}$$

Next, $coolT(\Gamma^c_{j,n}, \Gamma^{lim}_n)$ denotes the idle interval that is necessary for p_n to cool-off from Γ^{lim}_n to $\Gamma^c_{j,n}$. By replacing Γ^{fin} and Γ^{ini} by $\Gamma^c_{j,n}$ and Γ^{lim}_n, a generalised definition of $coolT()$ may also be obtained from equation 7.7, as:

$$coolT(\Gamma^{fin}, \Gamma^{ini}) = \frac{-1}{\mathcal{B}_n} \times log\Big(\frac{\Gamma^{fin} - \Gamma^a}{\Gamma^{ini} - \Gamma^a}\Big) \tag{7.23}$$

The term I denotes the number of cooling intervals needed for τ_j to complete execution on p_n.

$$I = \frac{\omega_{j,n} - MDE(\tau_j, p_n, \Gamma^{est}_{j,n})}{MDE(\tau_j, p_n, \Gamma^c_{j,n})} \tag{7.24}$$

$remET$ represents the last remaining execution chunk of τ_j on p_n before completion, and is obtained as:

$$remET = \omega_{j,n} - MDE(\tau_j, p_n, \Gamma^{est}_{j,n}) - \lfloor I \rfloor \times MDE(\tau_j, p_n, \Gamma^c_{j,n}) \tag{7.25}$$

The maximum temperature ($\Gamma_{j,n}^{cur}$) necessary to continuously execute τ_j for its last remaining $remET$ time units, while not violating the threshold temperature Γ_n^{lim}, is defined as:

$$\Gamma_{j,n}^{cur} = \Gamma_{j,n}^{ss} + \frac{\Gamma_n^{lim} - \Gamma_{j,n}^{ss}}{e^{-\mathcal{B}_n \times remET}} \tag{7.26}$$

Finally, OV denotes the *active* to *idle* (or *idle* to *active*) transition overhead associated with the *saving* (or *restoration*) of task τ_j intermediate state. As discussed above, the value of OV has been assumed to be $80\mu s$.

Example - 1: Let us consider the 10 task DAG in Fig. 7.1a. The DAG is to be executed on three heterogeneous processors. The WCETs of each task on the three processors are given in Fig. 6.1b. The processors' thermal parameters have been adopted from [57, 71] and listed in Table 7.2. Let the *steady state temperatures* corresponding to these tasks for the given processors be as listed in Fig 7.1b. Given OV, the optimal *cutoff* temperature of τ_j on p_n ($\Gamma_{j,n}^{oc}$) corresponds to that value of $\Gamma_{j,n}^{c}$ for which the lowest minima (having the longest *cutoff* interval) of function $T_E(\tau_j, p_n)$ (refer equation 7.21) is obtained, as $\Gamma_{j,n}^{c}$ is varied in the range $[\Gamma^a, \Gamma_n^{lim}]$. By putting $\Gamma_{j,n}^{oc}$ as the first argument of $coolT()$ (refer equation 7.23), we get the optimal *cool-off* interval size ($I_{j,n}^{oc}$).

Table 7.3: $\Gamma_{j,n}^{ss}$ *(in °C),* $\Gamma_{j,n}^{oc}$ *(in °C) and* $I_{j,n}^{oc}$ *(in s) of three processors*

Tasks	$\Gamma_{j,n}^{ss}$			$\Gamma_{j,n}^{oc}$			$I_{j,n}^{oc}$		
	p_1	p_2	p_3	p_1	p_2	p_3	p_1	p_2	p_3
τ_1	107	94	84	79.71	69.84	59.49	1.98	0.75	1.30
τ_2	105	98	83	79.63	69.60	59.72	2.53	1.89	0.71
τ_3	100	93	82	79.73	69.89	59.62	1.84	0.52	0.97
τ_4	101	95	82	79.88	69.76	59.69	0.82	1.13	0.79
τ_5	104	94	81	79.78	69.87	59.65	1.50	0.61	0.89
τ_6	103	94	85	79.69	69.81	59.62	2.12	0.90	0.97
τ_7	104	93	86	79.78	69.75	**59.59**	1.50	1.18	**1.04**
τ_8	102	92	83	79.52	69.77	59.69	3.28	1.09	0.79
τ_9	99	85	84	79.77	69.77	59.60	1.57	1.09	1.02
τ_{10}	107	98	83	79.71	69.60	59.34	1.98	1.89	1.68

Fig. 7.3b depicts τ_7's execution lengths on processors p_3, as the $\Gamma_{j,n}^{c}$ is progressively lowered from $\Gamma_3^{lim} = 60°C$. The plots in Fig. 7.3b validate the existence of an optimal cutoff temperature (for each distinct task-processor pair) for which the total execution duration T_E, is minimal. For example, $\Gamma_{7,3}^{oc}$, $I_{7,3}^{oc}$, and corresponding T_E's for τ_7 are obtained as $(\langle \Gamma_{7,3}^{oc}, I_{7,3}^{oc}, T_E \rangle)$; $\langle 59.82°C, 0.45s, 14.60s \rangle$ (indicated by black dot in the figure).

TETT(): The pseudocode of function *TETT()* is presented in Algorithm 17. The objective of *TETT()* is to generate an execution/idling schedule for a task τ_j on p_n, while ensuring that the temperature of p_n remains below the specified threshold value. *TETT()* returns this schedule along with the effective finish time $EFT[\tau_j, p_n]$ and the processor temperature at $EFT[\tau_j, p_n]$. Given the initial temperature Γ_n^{cur} of p_n, step 1 of Algorithm 17 determines the Maximum Duration of continuous Execution ($MDE()$; using equation 7.22) of τ_j before p_n's temperature reaches Γ_n^{lim} (and stores it in $allowET$). When $\omega_{j,n} \leqslant allowET$ is true (step 2), function *TETT()* can directly return (step 3) with the *schedule* $Sch_{j,n}$ being a single continuous execution interval (with no idling), $EFT[\tau_j, p_n]$ being $EST[\tau_j, p_n] + \omega_{j,n}$ and the *temperature* of p_n at time $EFT[\tau_j, p_n]$ calculated using $T_{heat}(\Gamma_{j,n}^{ss}, \Gamma^0, \omega_{j,n})$. Otherwise (steps 4-6), temperature of p_n reaches Γ_n^{lim} after $allowET$ units of continuous execution. Steps 5 and 6 respectively update the current values of τ_j's remaining execution requirement ($remET$) as well as the time up to which it must remain assigned on p_n.

Table 7.4: *Schedule produced by TMDS in each iteration for DAG in Fig. 7.1*

τ_j	R	OFT			EST			EFT			SCT			Allocation		
		p_1	p_2	p_3	p_1	p_2	p_3	p_1	p_2	p_3	p_1	p_2	p_3	$AFT[\tau_j]$	p_n	Γ_n^{cur}
τ_1	44.8	46.5	44.0	44.0	0.0	0.0	0.0	16.0	15.0	14.0	62.50	59.0	58.0	14.0	p_3	46.4°C
τ_5	28.4	32.0	29.33	24.0	17.0	16.0	14.0	36.0	34.0	34.0	68.0	63.33	58.0	34.0	p_3	53.4°C
τ_2	28.2	31.5	29.0	24.0	16.50	15.67	34.0	32.50	32.67	52.0	64.0	61.67	76.0	32.67	p_2	52.1°C
τ_4	26.5	28.5	27.0	24.0	17.50	32.67	34.0	36.50	42.67	51.0	65.0	69.67	75.0	36.50	p_1	33.3°C
τ_3	25.1	29.5	26.67	19.0	36.50	32.67	34.0	54.50	48.67	47.0	84.0	75.33	66.0	47.0	p_3	57.3°C
τ_6	23.5	27.5	22.0	21.0	36.50	32.67	47.0	51.50	48.67	63.62	79.0	70.67	84.62	48.67	p_2	55.1°C
τ_9	11.3	15.0	12.0	7.0	41.67	48.67	47.0	61.67	67.67	69.78	76.67	79.67	76.78	61.67	p_1	36.6°C
τ_7	11.3	15.0	12.0	7.0	61.67	48.67	**47.0**	81.67	66.67	61.84	96.67	78.67	68.84	**61.84**	p_3	60.0°C
τ_8	11.3	15.0	12.0	7.0	61.67	48.67	61.84	77.67	60.67	85.16	92.67	72.67	92.16	60.67	p_2	57.1°C
τ_{10}	0.0	0.0	0.0	0.0	65.67	67.67	64.67	80.67	79.67	74.01	80.67	79.67	74.01	74.01	p_3	60.0°C

Now, in order to complete $remET$ units of execution *TETT()*, must enter into a repetitive loop (steps 9-18). At each iteration of the loop, depending on the value of $remET$ (step 10), p_n must cool down either up to the optimal cutoff temperature $\Gamma_{j,n}^{oc}$ (step 12) or to a specific temperature from which τ_j can be continuously executed to completion while not breaching the threshold temperature Γ_n^{lim} (step 15). Step 7 calculates the interval that is necessary for p_n to cool-off from Γ_n^{lim} to $\Gamma_{j,n}^{c}$, while step 8 determines the continuous execution duration that will result in p_n's temperature to increase from $\Gamma_{j,n}^{c}$ to Γ_n^{lim}. The remaining steps of the algorithm are self-explanatory. In step 19, *TETT()* finally returns

with a schedule $Sch_{j,n}$ (that includes all the idle and execution intervals of τ_j), effective finish time $EFT[\tau_j, p_n]$ and the temperature (Γ_n^{lim}) of p_n at time $EFT[\tau_j, p_n]$.

Algorithm 17: $TETT(\tau_j, p_n, curT, \Gamma_n^{cur})$

Input: *Task* τ_j, *processor* p_n, *time* $curT$, *temperature* Γ_n^{cur}
Output: *Schedule Sch and effective finish time EFT of a task* τ_j *on* p_n *and temperature of* p_n *at EFT*

1 $allowET = MDE(\tau_j, p_n, \Gamma_n^{cur})$ refer equation 7.22;
2 **if** $\omega_{j,n} \leqslant allowET$ **then**
3 **return** $\langle Sch_{j,n}, curT + \omega_{j,n}, T_{heat}(\Gamma_{j,n}^{ss}, \Gamma^0, \omega_{j,n})\rangle$;
4 **else**
5 $remET = \omega_{j,n} - allowET$;
6 $curT = curT + allowET$;
7 $idleT = coolT(\Gamma_{j,n}^{c}, \Gamma_n^{lim})$ refer equation 7.23;
8 $allowET = MDE(\tau_j, p_n, \Gamma_{j,n}^{c})$ refer equation 7.22;
9 **while** $remET > 0$ **do**
10 **if** $remET \geqslant allowET$ **then**
11 // p_n *should idle for an interval* [$curT$, $curT + idleT + 2 \times OV$]
12 $curT = curT + idleT + allowET + 2 \times OV$;
13 // τ_j *should execute for an interval* [$curT - allowET$,$curT$]
14 **else**
15 Calculate $\Gamma_{j,n}^{cur}$ (using equation 7.26), the max temperature necessary to continuously execute τ_j for its last remaining $remET$ time units, while not breaching Γ_n^{lim};
16 $idleT = coolT(\Gamma_{j,n}^{cur}, \Gamma_n^{lim})$ refer equation 7.23;
17 $curT = curT + idelT + remET + 2 \times OV$;
18 $remET = remET - allowET$;
19 **return** $\langle Sch_{j,n}, curT, \Gamma_n^{lim}\rangle$;

Example Continued: Using the same example system (refer Example - 1), let us assume that the communication links between each pair of processors have the following bandwidths: $b_{1,2} = b_{2,1} = 1$, $b_{2,3} = b_{3,2} = 3$, $b_{3,1} = b_{1,3} = 2$, and $b_{1,1} = b_{2,2} = b_{3,3} = \infty$. The rows in Table 7.4 show the values of important terms, at each iteration during partial schedule generation by *TMDS()* (*while* loop steps 5-14, Algorithm 16). The Gantt chart in Fig. 7.4 depicts the schedule generated by *TMDS()* (*makespan* 73.62s).

In the Gantt chart, the light yellow coloured intervals denote phases when a task actually executes on a processor. On the other hand, the gray-coloured intervals denote phases when a processor idles with a certain task assigned on it. For example, when the task τ_7 is allocated

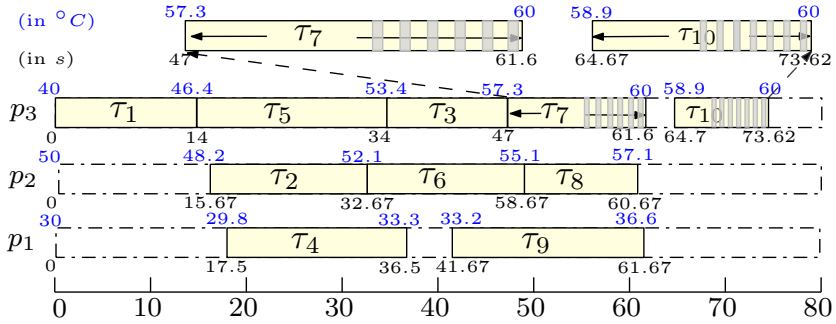


Figure 7.4: *Gantt chart of the schedule generated by TMDS (makespan = 74.01s) for the DAG in Fig. 7.1a.*

on processor p_3 at time $EST[\tau_{7,3}] = 47s$ (refer Table 7.4), the operating temperature of p_3 is $\Gamma^{est}_{7,3} = 57.3°C$. At this time, p_3 is allowed a continuous execution interval of $8.64s$ before reaching p_3's threshold ($\Gamma^{lim}_3 = 60°C$). To complete τ_7's remaining execute of $3.36s$ ($= 12-8.64s$), p_3 must cool down either up to the optimal cutoff temperature $\Gamma^{oc}_{7,3} = 59.70°C$ (with interval size $I^{oc}_{7,3} = .45s$) or to a specific temperature from which τ_7 can be continuously executed to completion, while not breaching the threshold temperature. To do so, processor p_3 remains idle in the following six intervals $\langle$(*start time, start temperature*), (*end time, end temperature*)$\rangle$: $\langle(55.64s, 60°C), (56.09s, 59.70°C)\rangle$, $\langle(56.7s, 60°C), (57.15s, 59.70°C)\rangle$, $\langle(57.76s, 60°C), (58.21s, 59.70°C)\rangle$, $\langle(58.82s, 60°C), (59.27s, 59.70°C)\rangle$, $\langle(59.88s, 60°C), (60.33s, 59.70°C)\rangle$ and $\langle(60.94s, 60°C), (61.29s, 59.89°C)\rangle$. Therefore, the total allocation duration of task τ_7 on p_3 is $14.6s$. This primarily includes τ_7's execution duration ($12s$) and p_3's idle intervals ($5 \times .45s + 0.35s = 2.6s$). It may be noted from Table 7.4 that the start time of τ_7 on p_3 is $47s$ and its finish time is $61.6s$.

7.4.4 Complexity Analysis

Step 2 of *TMDS* computes two parameters: *OFT* and *Rank*. *OFT* (refer equation 7.17) considers each edge exactly once and iterates over $|P|$ processors. Hence, the overhead of creating the *OFT* table is $O(|P|(|V| + |E|))$. The overhead of determining *Rank* (refer equation 7.18) of a task is $O(|P|)$ and for all tasks it becomes $O(|P| \times |V|)$. The complexity of the *while* loop (steps 5-14) is primarily governed by the overhead of calculating $EST[\tau_j, p_n]$ (refer equation 7.19; step 8) for all task-processor pairs. Computation of $EST[\tau_j, p_n]$ for a task τ_j on p_n requires $O(1)$ time over all predecessors of τ_j and hence has a complexity of $O(\#predecessors)$. As the sum of the predecessors of all tasks is equal to the total number of edges in the task graph, the amortized complexity of determining $EST[\tau_j, p_n]$ for τ_j on p_n is

$O(|P|(|V|+|E|)/(|P|\times|V|)) = O((|V|+|E|)/|V|) = O(|E|/|V|)$. Hence, the complexity for computing $EST[\tau_j, p_n]$ on all task-processor pairs is $O(|E|/|V| \times |V| \times |P|) = O(|E| \times |P|)$. Thus, the overall complexity of *TMDS* can be expressed as: $O(|P|(|V|+|E|)+|P|\times|V|+|E|\times|P|) \approx O(|E|\times|P|)$. It may be noted that the time complexity of the *TETT()* function is $O(1)$.

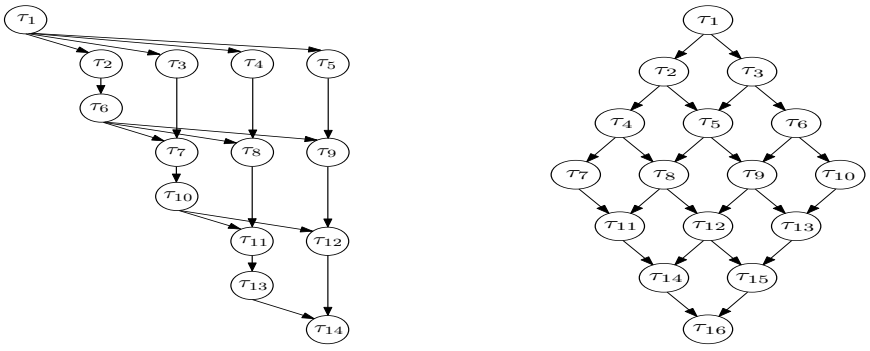
7.5 Experiments and Results

In this section, we experimentally assess the performance of *TMDS* against *four* existing strategies as discussed in Section 7.5.3. We first describe the experimental setup, the performance metrics and the comparison with related works in the following subsections, followed by detailed experimental results.

7.5.1 Experimental Setup

The experimental evaluation has been conducted using two real-world benchmark DAGs: Gaussian Elimination [74] and Laplace [69]. The structural representations of these task graphs are depicted in Fig. 7.5.

Gaussian Elimination is a method used to solve a system of linear equations. The DAG structure of this graph is characterized by *matrix-size* (ν). A Gaussian Elimination DAG has $(\nu^2+\nu-2)/2$ tasks and $(\nu^2-\nu-1)$ edges. For example, Fig. 7.5a depicts the DAG structure of Gaussian Elimination for *matrix-size* $\nu = 5$. The DAG has 14 tasks and 19 edges.



(a) *Gaussian Elimination* **(b)** *Laplace*

Figure 7.5: *Benchmark Task Graphs.*

The **Laplace** graph is derived from the Laplace equation solver algorithm. The size of this task graph is influenced by the size of the input parameter φ. The total number of tasks and edges in a Laplace DAG is equal to φ^2 and $2\varphi(\varphi-1)$. For example, Fig. 7.5b shows a Laplace task graph for $\varphi = 4$. The task graph consists of 16 tasks and 24 edges.

Data Generation Framework: We have generated an exhaustive set of random input test cases by carefully varying the following set of parameters to conduct our experiments.

1. **Task graph size**: For Gaussian Elimination, the *matrix-sizes* $\nu = \{8, 12, 15, 18, 20\}$, respectively produces task graphs having $|V| = \{35, 77, 119, 170, 209\}$ tasks and $|E| = \{55, 131, 209, 305, 379\}$ edges. Similarly for Laplace, $\varphi = \{6, 9, 11, 13, 14\}$ results in task graphs having $|V| = \{36, 81, 121, 169, 196\}$ and $|E| = \{60, 144, 220, 312, 364\}$.

2. **Number of processors**: $|P| = \{4, 8, 16, 32\}$.

3. **Task execution times**: Our experiments used a three-step approach to produce the execution time of each task on different heterogeneous processors. First, we fixed the average execution time ($\overline{\omega_{DAG}}$) of all tasks in the graph. $\overline{\omega_{DAG}} = \{10, 15, 20, 25\}$ has been considered in our experiment. Given $\overline{\omega_{DAG}}$, the average execution time ($\overline{\omega_j}$) of task τ_j over all processors is generated. Values of $\overline{\omega_j}$ for different tasks are generated from normal distributions ($\mathcal{N}(\mu, \sigma)$) with mean $\mu = \overline{\omega_{DAG}}$ and various standard deviation values $\sigma = \{1, 2, 3\}$. Finally, we obtain the execution times of task τ_j on each processor p_n at $\omega_{j,n} \sim \mathcal{N}(\overline{\omega_j}, \overline{\omega_j} \times \beta)$. Here, $\beta = \{0.1, 0.25, 0.5, 0.75, 1\}$ is the heterogeneity factor that indicates the skewness among execution times of a task on different processors.

4. **Communication-to-Computation Ratio (CCR)**: This parameter determines the ratio of relative overhead between message transmission and task execution. A higher CCR indicates that the system spends relatively more time in transmission of data between processors than task execution. Values of CCR chosen in this work are: $CCR = \{0.1, 0.25.0.5, 0.75, 1\}$. The mean inter-task message size ($\overline{data_{DAG}}$; in bytes) for a task graph is given by: $\overline{data_{DAG}} = CCR \times \overline{\omega_{DAG}} \times \overline{B}$. Here, $\overline{B}$ ($= \frac{1}{|P| \times (|P|-1)/2} \sum b_{m,n}$ $(1 \leqslant m \leqslant |P|; 1 \leqslant n < m)$) is the average communication bandwidth randomly sampled from $\overline{B} = \{1\ Gbps, 5\ Gbps\}$. An element $b_{m,n} \in B$ denotes the actual bandwidth of each link between p_m, p_n and is sampled from a normal distribution such that $b_{m,n} \sim \mathcal{N}(\overline{B}, 0.2 \times \overline{B})$. These $b_{m,n}$ values are further scaled appropriately such that $\sum_{m=1}^{|P|} \sum_{n=1}^{m-1} b_{m,n}$ becomes $|P| \times (|P|-1)/2 \times \overline{B}$. The output message size ($data_{i,j}$) for each edge (τ_i, τ_j) in the task graph is sampled from a normal distribution $data_{i,j} \sim \mathcal{N}(\overline{data_{DAG}}, 0.2 \times \overline{data_{DAG}})$ and then scaled such that $\sum data_{i,j}$ over all edges in the task graph equals $|E| \times \overline{data_{DAG}}$.

5. **Temperature parameters**: Similar to [57], the steady state temperature ($\Gamma^{ss}_{j,n}$) of a task τ_j on p_n is proportional to its intensity. A more intensive task will cause the processor to heat up to higher temperatures. The value of $\Gamma^{ss}_{j,n}$ for each task-processor pair is generated by sampling uniformly from the range $[40°C, 120°C]$. Similarly, the temperature threshold (Γ^{lim}_{n}) for each processor p_n is picked uniformly from the range $[60°C, 100°C]$. The thermal constant ($\mathcal{B}_n$) determines the thermal characteristics of the processor. A processor with higher $\mathcal{B}_n$ tends to quickly heat (cool) up (down). Values of $\mathcal{B}_n$ are sampled from a normal distribution with $\mu = 0.008$ and $\sigma = 0.004$. The ambient temperature (Γ^a) is the temperature of the environment where the system operates and is fixed to $25°C$ for all our experiments.

Simulation Framework: The simulation framework is written in *C* and is executed on a system having the following configuration: (i) Intel® Core™ i7-8550U CPU @ 1.8GHz ×8, (ii) 8 GiB Memory, and (iii) Ubuntu 20.04 LTS OS.

7.5.2 Performance Metrics

Performance of *TMDS* has been evaluated using the following three metrics:

1. **Schedule Length Ratio (SLR)**: The *makespan/schedule length* is the most commonly used performance measure of a DAG scheduling algorithm. Since, a large set of DAGs with various input parameters are used, we used a normalized schedule length measure named *Schedule Length Ratio (SLR)*. This metric compares the performance of the proposed temperature-aware *makespan* minimizing scheduler *TMDS*, against *THEFT*, *TPEFT*, *TPPTS* and *TPSLS*. Formally, SLR is determined as:

$$SLR = \frac{X_{ms}}{\sum_{\tau_j \in CP_{AVG}} \overline{\omega_j}} \tag{7.27}$$

   where X_{ms} symbolizes the *makespan* delivered by a scheduler like *TMDS*, *THEFT*, *TPEFT*, *TPPTS* and *TPSLS*. Considering execution time of each task to be its average value over all processors, the denominator depicts sum of the execution times of all tasks in the critical path (CP_{AVG}) of the DAG. It may be observed that lower the achieved SLR, better is the performance of the scheduling algorithm.

2. **Number of Improved Makespans**: We used this metric to pair-wise compare the *makespan* of two scheduling algorithms in a tabular format. In the table, we show

the percentages of cases in which one strategy has performed better, similar, or worse compared to the other.

3. **Run-time**: This metric determines the average run-time taken by a scheduling algorithm for data sets generated using a fixed set of input parameter values

.

7.5.3 Comparison with Related Works

To the best of our knowledge, the work proposed herein (*TMDS*) is the first one to deal with the temperature-aware scheduling of precedence-constrained DAG-structured applications on distributed heterogeneous processing platforms. That is, none of the well-known DAG scheduling strategies for heterogeneous platforms such as *HEFT* [85], *PEFT* [5], *PPTS* [24] and *PSLS* [112] are temperature-aware. Temperature-aware counterparts for any of these strategies are also not available in the literature. In this regard, it may be highlighted that the temperature-awareness mechanism presented in this chapter is generic and adaptable. Hence, the proposed mechanism can be easily employed to extend the DAG scheduling strategies mentioned above and obtain their temperature versions as follows. In general, all these strategies have a two-level design. The first level among them is concerned with task priority generation, while the second level deals with the allocation of processors-to-tasks in the order prescribed by the first level. While considering a processor for a task the temperature-awareness mechanism (refer *TETT()*) determines and takes into account the time required in complete the task on the given processor while never breaching the temperature cap associated with the processor. Following this approach, we have implemented temperature-aware versions (namely, *THEFT*, *TPEFT*, *TPPTS* and *TPSLS*) of the prominent existing algorithms *HEFT*, *PEFT*, *PPTS*, *PSLS*.

7.5.4 Performance Results

In this subsection, we present detailed experimental results using two benchmarks: Gaussian Elimination [74] and Laplace [69].

7.5.4.1 Experiment-1: Pair-wise *makespan* comparison of algorithms

Table 7.5 shows pair-wise *makespan* comparisons of *TMDS* with the algorithms *THEFT*, *TPEFT*, *TPPTS* and *TPSLS*. Specifically, the result corresponding to the $(row\ i, column\ j)^{th}$-entry in the table depicts the percentages of test cases for which the algorithm corresponding to the i^{th} row performs better, equal or worse than the algorithm in column j. A total of 100000 test cases using Gaussian Elimination task graphs have been considered by varying values of #tasks ($|V|$) and #processors ($|P|$), heterogeneity (β) and Communication-to-Computation Ratios (CCR) for each pair of algorithms. For example, the $(1,1)^{th}$- entry in Table 7.5 shows that *TMDS* performs better, equal and worse in 66.7%, 6.5% and 26.8% test cases respectively, compared to *TPSLS*. The results for Laplace have not been presented as they exhibit trends very similar to Gaussian Elimination.

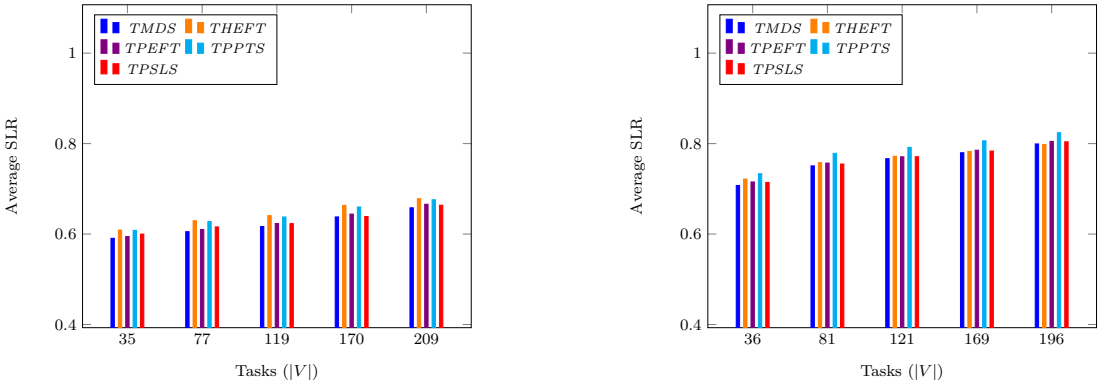
Table 7.5: *Pair-wise comparison of algorithms using Gaussian Elimination*

		TPSLS	*TPPTS*	*TPEFT*	*THEFT*
TMDS	better	66.7%	62.2%	54.0%	65.6%
	equal	6.5%	1.3%	14.2%	1.3%
	worse	26.8%	36.5%	31.8%	33.1%
THEFT	better	43.7%	40.5%	39.3%	
	equal	1.1%	1.7%	1.6%	
	worse	55.2%	57.8%	59.1%	
TPEFT	better	56.3%	54.9%		
	equal	12.5%	1.8%		
	worse	31.2%	43.3%		
TPPTS	better	52.1%			
	equal	1.4%			
	worse	46.5%			

7.5.4.2 Experiment-2: Comparison of schedule length ratios

This experiment measures the *schedule length ratio*s (SLR) of *TMDS*, *THEFT*, *TPEFT*, *TPPTS* and *TPSLS* for varying values of #tasks ($|V|$). We have also conducted the experiments for varying #processors ($|P|$), heterogeneity (β), and Communication-to-Computation Ratios (CCR). However, the results for varying $|P|$, β and CCR have not been presented here as the result trends are very similar. Each data point in the experiments below is obtained as the average over solutions generated with 200 different task graph data corresponding to a fixed set of parameter values. Obtained results for both Gaussian Elimination

and Laplace are presented in Fig. 7.6.



(a) Gaussian Elimination

(b) Laplace

Figure 7.6: *Schedule Length Ratios for varying number of tasks.*

Fig. 7.6 shows average SLR values for the Gaussian Elimination and Laplace task graphs, as the number of tasks vary between ≈ 30 and ≈ 200. Values of the parameters *CCR*, $|P|$ and β have been fixed at 0.5, 8 and 0.75, respectively. For Gaussian Elimination, *TMDS* is seen to deliver better results than *THEFT*, *TPEFT*, *TPPTS* and *TPSLS*, in all cases. As an example of *TMDS's* performance, in Fig. 7.6a (Gaussian Elimination) for $|V| = 119$, the average schedule lengths of *TMDS* is lower than *THEFT*, *TPEFT*, *TPPTS* and *TPSLS* by approximately 3.9%, 1.1%, 3.4% and 1.1%. For Laplace (7.6b), performs of *TMDS* may be seen to be slightly better or comparable to the other algorithms. As is obvious, the *makespans* (SLR values) increase as workloads become higher with increase in the number of tasks.

7.5.4.3 Experiment-3: Comparison of run-times

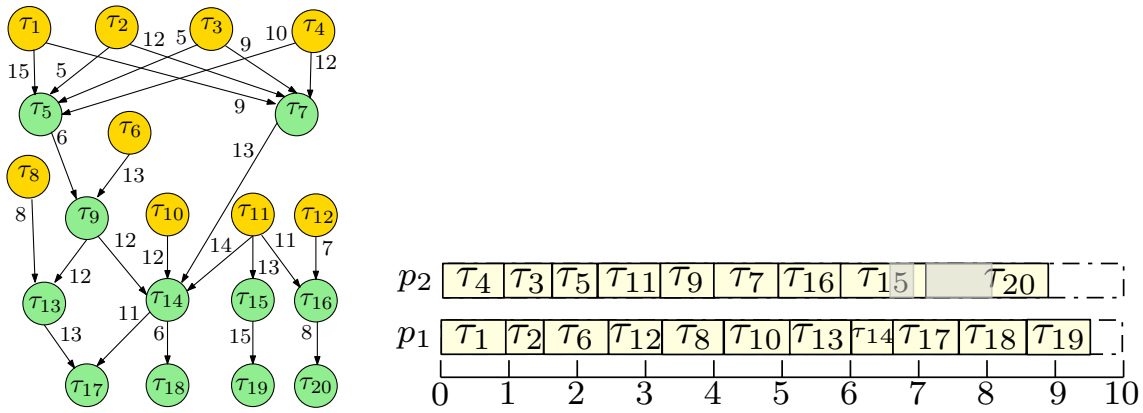
This experiment measures the *run-time*s of *TMDS*, *THEFT*, *TPEFT*, *TPPTS* and *TPSLS* for varying values of #tasks ($|V|$). Obtained results for Laplace are detailed in Table 7.6. The parameters $|P|$, *CCR* and β are set to 8, 0.5 and 0.75, respectively. It is observed that the *run-time*s of all the schedulers strictly increase with the number of tasks. The *run-time*s of *TMDS* is upper bounded by $\approx 0.98ms$ for the considered test cases.

Table 7.6: *Average run-times (in ms) of different Algorithms using Laplace*

τ_j	*TMDS*	*THEFT*	*TPEFT*	*TPPTS*	*TPSLS*
36	0.11	0.06	0.12	0.12	0.18
81	0.29	0.15	0.29	0.30	0.52
121	0.52	0.29	0.52	0.54	1.04
169	0.74	0.45	0.75	0.77	1.60
196	0.98	0.59	0.98	0.99	2.13

7.6 Case Study: Adaptive Cruise Controller

To demonstrate the practical applicability of *TMDS* in real-world settings, we present a case study using an Adaptive Cruise Controller (ACC) in automotive systems. ACC automatically preserves a safe distance between two cars [41]. Fig. 3.11a illustrates the ACC block diagram adapted from [41] and Fig. 7.7a displays its related DAG structure. This ACC application consists of 20 executable tasks that need to be scheduled on a platform with two processors p_1 and p_2 having the thermal properties $\langle \mathcal{B}_n, \Gamma_n^{lim}, \Gamma_n^{cur} \rangle$: $\langle 0.02267, 70, 45 \rangle$ and $\langle 0.0313, 60, 55 \rangle$, respectively.



(a) *ACC: DAG* **(b)** *ACC: TMDS Schedule*

Figure 7.7: *Adaptive Cruise Controller (ACC): (a) ACC DAG, (b) Gantt chart of the ACC schedule generated by TMDS (makespan = 9.5s) for the DAG in Fig. 7.7a.*

Table 7.7 lists the WCETs and the steady state temperature of the ACC task graph on the three heterogeneous processors. The communication bandwidths between the processors are considered to be 500 KB/s. Employing *TMDS*, we generate a schedule for the ACC application and the respective Gantt chart is presented in Fig. 7.7b. It may be observed

Table 7.7: *ACC:* $\omega_{j,n}$ *(in s) and* $\Gamma^{ss}_{j,n}$ *(°C) of* τ_j *on two processors*

		τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}
ω	p_1	.9	.6	.6	1	.8	.9	1	.9	.9	1	1	.8	.9	.6	1	1	1	1	.9	.9
	p_2	.8	.9	.8	.8	.7	1	.9	.8	.8	.8	.9	.9	.7	.5	1	.9	.9	.9	.6	.8
Γ^{ss}	p_1	98	97	99	96	98	99	91	95	93	97	94	99	97	93	94	92	95	96	94	97
	p_2	84	84	88	74	86	86	83	82	86	81	75	78	77	86	81	83	84	81	89	93

that, *TMDS* is able to deliver a schedule with *makespan* ($9.5s$) while satisfying the given thermal constraints.

7.7 Summary

This chapter presented *TMDS*, a low-overhead temperature aware list scheduling algorithm for DAG-structured applications on distributed heterogeneous systems. The objective of *TMDS* is to generate a *makespan* minimizing schedule while satisfying, (i) execution and communication demands of application tasks, (ii) processor capacity and communication bandwidth constraints of the platform, and (iii) temperature threshold bound associated with all processors, over the entire schedule. The proposed temperature management strategy is a generic approach that can be easily used to adapt existing *makespan* minimizing DAG schedulers (for example, *THEFT*, *TPEFT*, etc.), so that the delivered schedules never violate threshold temperature bounds of processors. This generic approach is important because as shown in Table 7.5, although *TMDS* is better or comparable in performance to the other state-of-the-art algorithms in a majority of considered test cases, there still are a significant number of test case scenarios in which one or more existing algorithms deliver slightly better results than *TMDS*. Our generic temperature management scheme can be used to employ *TMDS* or temperature-aware versions of any of the other algorithms as needed, in a given system scenario. Finally, the practical applicability of the proposed scheme has been additionally exhibited using a real-world case study with an adaptive cruise controller from the automotive system's domain.

The next chapter summarizes the contributions of this dissertation and discusses a few possible extensions to this research.

Chapter 8

Conclusions and Future Perspectives

8.1 Summary of Contributions

This dissertation presents a few novel ideas towards the design of low-overhead scheduling techniques for both single DAG or multiple independent DAG applications on distributed heterogeneous RT-CPSs. RT-CPSs integrate sensing, computation, control and networking into physical entities and infrastructure, enabling a revolution of smart gadgets and systems from smart agriculture to smart cars. The thesis which unfolds through the dissertation is as follows: *Given precedence-constrained applications to be executed on distributed platforms, the list-based design philosophy is effective towards obtaining low-overhead but efficient offline real-time scheduling policies for satisfying diverse objectives/constraints related to resource usage efficiency, energy, security, temperature, etc.* The entire thesis work comprises multiple contributions categorized into five contributory chapters. Our first contribution has been the development of an efficient real-time DAG-scheduling framework that attempts to minimize a *generic penalty function*. The designed penalty function can be amicably adapted towards its deployment in various application domains such as real-time cyber-physical systems like automotive and avionic systems, cloud computing, smart grids, etc. In the second contribution, we developed a state-space search guided heuristic scheduling algorithm called *HMDS*, whose objective is to minimize *schedule length*. The design of *HMDS* is generic enough and can be easily adapted to various distributed systems such as industrial assembly line balancing, cloud/fog computing, grid computing, etc., where the applications to be executed are often structured as task graphs. Contributions one and two cover the first three objectives of the thesis. In order to achieve the fourth

objective of the thesis, a mechanism for co-scheduling *multiple independent periodic DAG applications* has been devised within the third contributory chapter. The objective of the scheduling algorithm is to minimize dissipated energy. Subsequently, in the fourth contribution, a security-aware real-time DAG scheduling strategy has been designed. The scheme maximizes total security utility for a given application having known minimum security strength specifications for its messages. Finally, in the last contribution of the dissertation, we have developed a mechanism to construct minimum *makespan* schedules for precedence-constrained task graph applications with known thermal characteristics on a heterogeneous processing platform. The last two contributions are oriented towards fulfilling the thesis's fifth objective. We now present brief synopses of the contributory chapters in more detail.

An important insight obtained through our literature survey (refer Section 2.5 of Chapter 2) was that effective task-message co-scheduling schemes for real-time DAG applications can be designed through an efficient iterative scheme which employs state-of-the-art makespan minimization strategies as their core. Based on this insight, in our first contributory chapter (i.e., Chapter 3), we have proposed a general model and a corresponding heuristic algorithm to schedule real-time DAG applications on heterogeneous platforms. The model's objective is to minimize a generic penalty function, which allows the model to be adapted to different domains such as automotive and avionic systems, cloud computing, smart grids, industrial automation, etc. The proposed heuristic algorithm *PRESTO* comprises two phases: *initialization* and *allocation*. The primary objective of the *initialization* phase is to determine a task priority list, as well as to estimate for each task-processor pair the total time and corresponding penalty associated with the execution of all tasks starting from the execution of itself, up to the sink task node. The *allocation* phase aims to generate a real-time static schedule obtained by sequentially determining a *processor allocation* and an *actual start time* for each task, in the order prescribed by the priority list obtained in the previous step, such that the overall *penalty* is minimized. The allocation phase is a bounded iterative process that continues until a valid schedule is successfully generated in a specific iteration or finally exits with a makespan minimizing schedule, which we refer to as *MMSH*. Here, the *MMSH* algorithm acts as the core of the proposed real-time scheduler *PRESTO*. Experimental analysis using two benchmark task graphs reveals that *MMSH* is able to outperform state-of-the-art *makespan* minimization policies including *HEFT*, *PEFT*, *PPTS*, *PSLS* and *PALG*, in most cases. To illustrate the generic applicability of *PRESTO* in real-world designs, we presented two case studies. In the first case study, *PRESTO* has

been used to schedule and map an (automotive) *adaptive cruise control* application such that energy consumption is minimized. In the second case study, *PRESTO* has been used to minimize the total monetary cost involved in the execution of an *intelligent surveillance application* running in a fog environment. It may be noted that the generic penalty function has been adapted through minimize energy in the first case study, while it has been used for monetary cost minimization in the second case study. this shows the designed penalty function is quite flexible and can be amicably molded as different objective functions as needed in particular application scenarios. We have also shown that *PRESTO* requires less than 25 ms to generate a valid schedule for benchmark task graphs with up to 250 tasks on platforms consisting of 32 heterogeneous processors.

While designing *MMSH*, we realized that there is ample scope for improving *MMSH* as well as other state-of-the-art makespan minimizing DAG scheduling strategies mentioned above, by systematically applying principles of any time heuristic search. Based on this insight, in Chapter 4, we have designed two makespan minimizing algorithms for DAG-structured applications. The first algorithm *HMDS-Bl* is a list-based scheduler whose task prioritization and processor selection phases depend on a matrix called Predicted Finish Time (PFT), which provides a lower bound on the remaining execution time of the successors of a given task, provided the task is scheduled on a particular processor. The second algorithm, *HMDS* is a branch-and-bound extension of *HMDS-Bl* whose run-time is kept in check by restricting the branching factor and terminating the search once the search time exceeds the baseline by a given factor. *HMDS* allows the designer to obtain a judicious balance between performance (makespan) and solution generation times. The results obtained in Table 4.3 (refer Experiment-6 of Chapter 4) show that *HMDS* achieves a steady improvement in performance as more time is allowed in its search for better solutions. The proposed heuristics were evaluated through extensive simulations using two real-world task graphs. The results show that *HMDS* comprehensively outperforms *HEFT*, *PEFT*, *MMSH* and other state-of-the-art schemes under various conditions with different parameters. Finally, the practical adaptability of *HMDS* is shown using a prototype real-platform implementation and a real-world case study on traction control application.

The works are done in Chapters 3 and 4 dealt with single DAG-structured applications on heterogeneous distributed platforms. As discussed in the motivation section (refer Section 1.2 of Chapter 1), large systems which constitute multiple control subsystems typically follow a federated resource allocation policy as it allows simpler design, albeit,

at the cost of significantly lower resource utilizations, in many cases. In order to improve the usage efficiency of available processing and network resources, in Chapter 5, we thought to extend *PRESTO* for the co-scheduling of multiple independent task graphs. As *PRESTO* takes a single real-time task graph as input, the set of independent periodic task graphs to be co-scheduled must first be merged into a single task graph which can be applied as input to *PRESTO*. Let a set of independent persistently executing *periodic* real-time DAG applications $G = \{G^1, G^2, \ldots, G^r, \ldots, G^{|G|}\}$ having respective deadlines $D = \{D^1, D^2, \ldots, D^r, \ldots, D^{|G|}\}$ (refer Fig. 5.1). The applications being periodic, the time interval, say D^0 (referred to as hyperperiod), after which arrival times of all application instances synchronize, is given by $D^0 = LCM(D)$. In each hyperperiod, any application G^r sequentially invokes I^r $(= D^0/D^r)$ instances. It may be appreciated that sequential execution of the I^r instances of any application DAG G^r can be modeled as the execution of a single composite DAG, where (i) end-to-end deadline of the composite DAG is D^0, (ii) the I^r DAG instances are arranged such that the sink node of any instance say G^r_{n-1} is connected to the source node of G^r_n, through a dummy edge to enforce precedence relationship between them, (iii) the execution start time of the source node of any instance say G^r_n happens on or after $AT[G^r_n] = (n-1) \times D^r$, (iv) the completion time of any instance say G^r_n happens on or before $n \times D^r$. All these composite DAGs can further be modeled as a single merged DAG G^0 by connecting their individual source and sink nodes to a single dummy source (τ_0) and sink node (τ_{exit}). It may be noted that G^0 has an end-to-end deadline of D^0. *PRESTO* delivers faulty schedules for almost all test cases when a single merged DAG G^0 is used as input. This is because *PRESTO* is not equipped with the necessary critical features that are required to handle multi-application merged DAGs. These features include:

- An enhanced task prioritization mechanism, where *ranks* of the source nodes of all application instances (within the merged DAG) are designed to be aware of the relative arrival times (within the hyperperiod) of the respective application instances. Similarly, *ranks* of the sink nodes of all application instances must be made aware of application instance deadlines. Finally, the *ranks* of all other intermediate nodes within each application instance (within the merged DAG) must be sensitive to the instance's relative deadline.

- In addition to such an enhanced task prioritization mechanism which is necessary for improved task ranking, the allocation phase during partial schedule generation must

be able to recognize and take corrective actions in situations when:

- the scheduler attempts to assign to a source task node a start time that is earlier than the stipulated relative arrival time.
- the scheduler attempts to assign to a sink task node a start time which causes the application instance's relative deadline to be missed.

The extended version of *PRESTO* (we refer to as *DPMRS*) includes all the necessary critical features discussed above for scheduling *multiple independent periodic real-time applications*. The overall objective of *DPMRS* is to minimize total dynamic energy dissipation associated with the execution of multiple independent periodic DAGs using a DVFS approach. For the same problem, this chapter also presented another scheduler named *NDPMRS* (an adaption to *DPMRS*), targeting the platforms which are not DVFS-enabled. Experimental analysis employing four benchmark task graphs, namely CyberShake, Stencil, Gaussian Elimination and Epigenomics acknowledges that *DPMRS* performs appreciably over extensive sets of test scenarios, pointing to the practical effectiveness of the scheme. Compared to state-of-the-art energy-aware single DAG scheduler *NDES&GDES*, the global nature of *DPMRS* allows it to harness significantly improved processor/communication resource sharing among different benchmark DAGs, in addition to better exploitation of task-processor affinities in the heterogeneous environment. Due to such efficient sharing and affinity awareness, *DPMRS* is able to achieve considerably lower energy dissipation, compared to *NDES&GDES*. For example, in the scenario consisting of 16 processors the *Normalized Energy-dissipation* suffered by *DPMRS* is 3.69 W. In comparison, *NDES&GDES* suffer significantly higher dissipation − 70.45 W. Finally, the practical adaptability of *DPMRS* and *NDPMRS* is exhibited through a case study with an automotive control system.

The scheduling strategies proposed in the first three contributory chapters (refer Chapters 3, 4, 5) do not focus towards ensuring the security needs of networked RT-CPS applications. However, data communication between dependent task nodes running on different processing elements is often realized through message transmission over a public network and is hence susceptible to multiple security threats such as *snooping*, *alteration* and *spoofing*. Several alternative security protocols having varying security strengths and associated implementation overheads are available in the market, for incorporating *confidentiality*, *integrity* and *authentication* on the transmitted messages. Hence, given a resource-constrained computation platform, a security-aware RT-CPS scheduler should be able to judiciously choose

appropriate schemes for the three types of security services, so that overall security of the system is maximized while adhering to stipulated timeliness constraints. In Chapter 6, we have presented a low-overhead security-aware real-time list scheduling algorithm named *SHIELD* for DAG-structured applications on distributed heterogeneous systems. *SHIELD* first calls *HSMS* to generate a *makespan* minimizing schedule while satisfying the minimum security demands of each message in the application. *SHIELD* returns with failure if the *makespan* generated by *HSMS* violates the given deadline. Otherwise, it attempts to enhance the security strengths of all messages such that the total security utility of the system is maximized. Experimental evaluation using two benchmark task graphs reveal that *SHIELD* significantly outperforms greedy baseline strategies *SHIELDb* in terms of solution generation times (run-times) and *SHIELDf* in terms of achieved security utility, over various input test scenarios. Finally, the practical applicability of *SHIELD* is exhibited through a case study on the *Traction Control* application in automotive systems.

Recent RT-CPSs often involve the use of intricate micro-architectural designs and very small feature sizes leading to complex chips with multi-million gates. Such ultra-high gate densities often make these chips susceptible to inappropriate surges in core temperatures. Temperature surges above a specific threshold may throttle processor performance, enhance cooling costs and reduce processor life expectancy. In Chapter 7, we have presented a low-overhead temperature aware list scheduling algorithm called *TMDS*, for DAG-structured applications. The objective of *TMDS* is to generate a *makespan* minimizing schedule while satisfying, (i) execution and communication demands of application tasks, (ii) processor capacity and communication bandwidth constraints of the platform, and (iii) temperature threshold bound associated with all processors, over the entire schedule. The proposed temperature management strategy is a generic approach that can be easily used to adapt existing *makespan* minimizing DAG schedulers (for example, *THEFT*, *TPEFT*, etc.), so that the delivered schedules never violate threshold temperature bounds of processors. This generic approach is important because as shown in Table 7.5, although *TMDS* is better or comparable in performance to the other state-of-the-art algorithms in a majority of considered test cases, there still are a significant number of test case scenarios in which one or more existing algorithms deliver slightly better results than *TMDS*. Our generic temperature management scheme can be used to employ *TMDS* or temperature-aware versions of any of the other algorithms as needed, in a given system scenario. Finally, the practical applicability of the proposed scheme has been additionally exhibited using a real-world case

study with an adaptive cruise controller from the automotive system's domain.

DAG application models are considerably general and appear in a variety of safety-critical cyber-physical systems like spacecraft, industry 4.0-enabled factories, healthcare, etc. We purview that our proposed design strategies and the developed design framework will go a long way towards solving important problems related to the efficient design and optimization of RT-CPSs in various domains and also provide future research directions to academia, researchers and scientists.

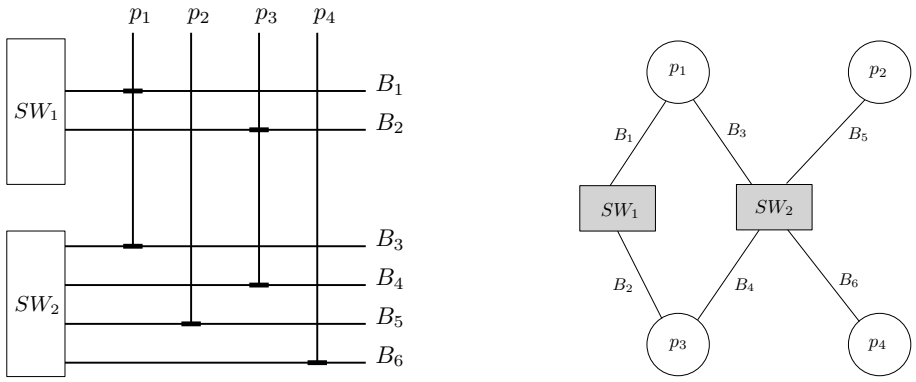
8.2 Scope for Future Research

The works presented in this thesis leave several open directions, and there is ample scope for future research in this area. In this section, we present a few such future perspectives.

- **Deployment of *PRESTO* on real communication frameworks**

  The work named *PRESTO* proposed in Chapter 3, is a generic penalty-aware real-time scheduling strategy targeted to a distributed platform consisting of a set of fully connected heterogeneous processors. Although the '*fully connected processors model*' is widely being used in many practical scenarios such as ZigBee-based wireless sensor networks, WiFi networks, etc., there exists a large class of cyber-physical control systems in diverse domains such as avionics, manufacturing systems, automotive, etc., where the set of processing elements are inter-connected through broadcast shared buses. In these systems, there exists contention for the shared buses, in addition to the contention for processors. Although, *PRESTO* can not be directly applied on these platforms, we purview that the strategy can be extended by enabling concurrent processor bus co-allocations in a real-time setting. In this regard, we first want to conduct research related to the extension of *PRESTO* for CAN-based distributed processing platforms. Further, we also plan to design a processor-bus co-scheduling strategy for applications with futuristic communication frameworks such as Time Sensitive Networking (TSN; IEEE 802.1Qbv; [19, 63, 86]). TSN is a switched Ethernet-based protocol that can support precise real-time communication. TSN is envisioned to be widely used in several application domains ranging from automotive to industrial automation systems. 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. As an example (refer Figure 8.1), messages from processor p_1 can either be passed through switch SW_1 or switch SW_2 to reach processor p_3. 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 *PRESTO*, the corresponding message communication schedule may be considerably more involved. The task-message co-scheduling strategy must concurrently address processor and bus allocation as a single step to achieve potentially disruptive performance gains compared to more ad hoc strategies which are currently being employed.



(a) *Detail View* **(b)** *Abstract View*

Figure 8.1: *Example of a switched network; Here, p_1, p_2, p_3, p_4 are processors, B_1, B_2, B_3, B_4 are buses and SW_1, SW_2 are switches.*

- **Scheduling dynamic applications in the presence of persistent applications**

  Chapter 5 addresses the problem of co-scheduling a set of independent periodic DAG applications executing on heterogeneous platforms. As the solution approach, we have devised a heuristic static real-time scheduling strategy whose objective is to minimize dissipated energy. However, CPSs like automotive systems may consist of both periodic and aperiodic applications and these applications are represented as real-time independent DAGs. Periodic applications are triggered at the start of the system and continue running periodically till the system stops. Hence, these applications have a persistent nature. Examples of periodic applications include anti-lock braking systems, fuel injection, chassis control, traction control, etc., in an automotive system.

On the other hand, aperiodic applications are triggered by the occupant of an event dynamically at run time, and terminate after, the execution of one or more instances. Hence, these applications can be considered to be non-persistent. Examples of such aperiodic automotive applications include central locking systems, break lights, power windows, etc. These applications may have multiple Quality of service (QoS) levels and may be executed on homogeneous or heterogeneous platforms which are centralized or distributed. Here, we plan to address the following problem: *Given a set of persistent and dynamic DAG applications (which may arrive at any time when the system is under operation) executing on heterogeneous platforms, the objective is to generate a real-time schedule that allows guaranteed execution of all persistent DAGs while maximizing the number and performance (QoS) of dynamic applications that can be incorporated.*

- **Design of hybrid offline-online scheduling strategies for enhancing resource usage efficiency**

  The static resource allocation strategies developed as part of this thesis are based on worst-case resource usage estimates of applications. Although these strategies are more predictable as well as provide better performance and timelines for the safety-critical hard real-time cyber-physical systems, they are prone to significant performance degradation when actual resource usage of the applications is 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 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 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, the determination of the exact offline-online strategies to be employed for specific system scenarios at hand is non-trivial and demands considerable research.

8.3 Disseminations out of this Work

Transactions/Journal Papers

1. D. Senapati, A. Sarkar and C. Karfa, "PRESTO: A Penalty-aware Real-Time Scheduler for Task Graphs on Heterogeneous Platforms," in IEEE Transactions on Computers (IEEE TC), vol. 71, no. 2, pp. 421-435, DOI: 10.1109/TC.2021.3052389, February 2022.

2. D. Senapati, A. Sarkar, C. Karfa, "HMDS: A Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems," in ACM Transactions on Embedded Computing Systems (ACM TECS), Special Issue on ESWEEK, 20, 5s, Article 106, 26 pages. DOI: https://doi.org/10.1145/3477037, October 2021.

3. D. Senapati, A. Sarkar and C. Karfa, "Energy-aware Real-time Scheduling of Multiple Periodic DAGs on Heterogeneous Systems," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 8, pp. 2447-2460, doi: 10.1109/TCAD.2022.3228504, December 2022.

4. D. Senapati, P. Bhagat, C. Karfa and A. Sarkar, "SHIELD: Security-aware Scheduling for Real-time DAGs on Heterogeneous Systems," in IEEE Transactions on Dependable and Secure Computing (IEEE TDSC) (Under Revision).

5. D. Senapati, K. Rajesh, C. Karfa and A. Sarkar, "A Temperature-aware Makespan Minimizing DAG Scheduler for Heterogeneous Systems," in ACM Transactions on Design Automation of Electronic Systems (ACM TODAS) (Reviewed).

Conference Papers

1. D. Senapati, A. Sarkar, C. Karfa, "Performance-Effective DAG Scheduling for Heterogeneous Distributed Systems," in Doctoral Symposium of 23rd International Conference on Distributed Computing and Networking (ICDCN 2022), January 2022.

2. D. Senapati, A. Sarkar, C. Karfa, "HMDS: A Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems," in ACM SIGBED International Conference On Embedded Software (EMSOFT 2021), July 2021.

Doctoral Committee

Chairperson	Prof. Hemangee K. Kapoor Professor Department of Computer Science and Engineering Indian Institute of Technology Guwahati
Supervisors	Dr. Arnab Sarkar Associate Professor Advanced Technology Development Centre Indian Institute of Technology Kharagpur
	Dr. Chandan Karfa Associate Professor Department of Computer Science and Engineering Indian Institute of Technology Guwahati
Indian External Examiner	Prof. Preeti Ranjan Panda Professor Department of Computer Science and Engineering Indian Institute of Technology Delhi
External External Examiner	Prof. Maryline Chetto Professor Laboratory of Digital Sciences of Nantes Nantes University France
Members	Prof. Purandar Bhaduri Professor Department of Computer Science and Engineering Indian Institute of Technology Guwahati
	Dr. John Jose Associate Professor Department of Computer Science and Engineering Indian Institute of Technology Guwahati

Bibliography

[1] Cplex. *https://www.ibm.com/in-en/analytics/cplex-optimizer*. [Pg.40]

[2] S. Abrishami, M. Naghibzadeh, and D. H. Epema. Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1400–1414, 2011. [Pg.2]

[3] S. AlEbrahim and I. Ahmad. Task scheduling for heterogeneous computing systems. *The Journal of Supercomputing*, 73(6):2313–2338, 2017. [Pg.5], [Pg.29], [Pg.36], [Pg.50], [Pg.54], [Pg.87], [Pg.89]

[4] H. Arabnejad and J. Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 633–639. IEEE, 2012. [Pg.7], [Pg.31]

[5] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014. [Pg.1], [Pg.2], [Pg.5], [Pg.28], [Pg.29], [Pg.36], [Pg.45], [Pg.50], [Pg.54], [Pg.71], [Pg.87], [Pg.89], [Pg.121], [Pg.162], [Pg.184]

[6] S. Bak and S. Chaki. Verifying cyber-physical systems by combining software model checking with hybrid systems reachability. In *Proceedings of the 13th International Conf. on Embedded Software*, EMSOFT, pages 1–10. ACM, 2016. [Pg.1]

[7] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *TECS*, 15(1):1–34, 2016. [Pg.25], [Pg.30]

[8] E. Bampis, D. Letsios, G. Lucarelli, E. Markakis, and I. Milis. On multiprocessor temperature-aware scheduling problems. *Journal of Scheduling*, 16(5):529–538, 2013. [Pg.33]

[9] S. Baruah. Scheduling dags when processor assignments are specified. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, pages 111–116, 2020. [Pg.4]

[10] M. Birks and S. P. Fung. Temperature aware online scheduling for throughput maximisation: The effect of the cooling factor. *Sustainable Computing: Informatics and Systems*, 4(3):151–159, 2014. [Pg.8], [Pg.33]

[11] M. Bishop. What is computer security? *IEEE Security & Privacy*, 1(1):67–69, 2003. [Pg.133]

[12] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *18th Euromicro Conf. on PDNP*, pages 27–34. IEEE, 2010. [Pg.5], [Pg.28], [Pg.29]

[13] C. Bolchini and A. Miele. Reliability-driven system-level synthesis for mixed-critical embedded systems. *IEEE Transactions on Computers*, 62(12):2489–2502, 2013. [Pg.65], [Pg.66]

[14] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll. Task period selection to minimize hyperperiod. In *ETFA2011*, pages 1–4. IEEE, 2011. [Pg.118]

[15] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. [Pg.1], [Pg.2], [Pg.17], [Pg.19], [Pg.24]

[16] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008. [Pg.5], [Pg.28]

[17] T. Chantem, X. Wang, M. D. Lemmon, and X. S. Hu. Period and deadline selection for schedulability in real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 168–177. IEEE, 2008. [Pg.118]

[18] E. G. Coffman and J. L. Bruno. *Computer and job-shop scheduling theory.* John Wiley & Sons, 1976. [Pg.27]

[19] 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*, pages 183–192, 2016. [Pg.195]

[20] M. I. Daoud and N. Kharma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68(4):399–409, 2008. [Pg.5], [Pg.28], [Pg.29]

[21] 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*, pages 278–283, 2007. [Pg.118]

[22] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011. [Pg.19]

[23] J. Deepakumara, H. M. Heys, and R. Venkatesan. Performance comparison of message authentication code (mac) algorithms for internet protocol security (ipsec). In *Proc. Newfoundland Electrical and Computer Engineering Conf*, 2003. [Pg.133]

[24] H. Djigal, J. Feng, and J. Lu. Task scheduling for heterogeneous computing using a predict cost matrix. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, pages 1–10, NY, USA, 2019. ACM. [Pg.5], [Pg.29], [Pg.36], [Pg.50], [Pg.54], [Pg.87], [Pg.89], [Pg.162], [Pg.184]

[25] H. Djigal, J. Feng, and J. Lu. Performance evaluation of security-aware list scheduling algorithms in iaas cloud. In *2020 20th IEEE/ACM Intl. Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 330–339. IEEE, 2020. [Pg.2], [Pg.29]

[26] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, 1990. [Pg.2]

[27] C. Fidge. Real-time scheduling theory. 2002. [Pg.26]

[28] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979. [Pg.27]

[29] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017. [Pg.65], [Pg.67]

[30] B. Hamidzadeh, L. Y. Kit, and D. J. Lilja. Dynamic task scheduling using online optimization. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1151–1163, 2000. [Pg.28]

[31] M. H. Hilman, M. A. Rodriguez, and R. Buyya. Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 53(1):1–39, 2020. [Pg.7], [Pg.31]

[32] C.-C. Hsu, K.-C. Huang, and F.-J. Wang. Online scheduling of workflow applications in grid environments. *Future Generation Computer Systems*, 27(6):860–870, 2011. [Pg.7], [Pg.31]

[33] M. Hu, J. Luo, Y. Wang, and B. Veeravalli. Scheduling periodic task graphs for safety-critical time-triggered avionic systems. *IEEE Transactions on Aerospace and Electronic Systems*, 51(3):2294–2304, 2015. [Pg.4], [Pg.7], [Pg.31]

[34] H. Huang, V. Chaturvedi, G. Quan, J. Fan, and M. Qiu. Throughput maximization for periodic real-time systems under the maximal temperature constraint. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2s):1–22, 2014. [Pg.33], [Pg.164]

[35] Q. Huang, S. Su, J. Li, P. Xu, K. Shuang, and X. Huang. Enhanced energy-efficient scheduling for parallel applications in cloud. In *CCGRID 2012*, pages 781–786. IEEE, 2012. [Pg.6], [Pg.30], [Pg.31]

[36] E. Ilavarasan and P. Thambidurai. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *Journal of Computer sciences*, 3(2):94–103, 2007. [Pg.5], [Pg.28], [Pg.29]

[37] E. Ilavarasan, P. Thambidurai, and R. Mahilmannan. High performance task scheduling algorithm for heterogeneous computing system. In *International conference on algorithms and architectures for parallel processing*, pages 193–203. Springer, 2005. [Pg.5], [Pg.29]

[38] M. A. Iverson, F. Ozguner, and G. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computing Workshop (HCW'95)*, pages 93–100, 1995. [Pg.5], [Pg.27], [Pg.28]

[39] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013. [Pg.51], [Pg.87], [Pg.102], [Pg.118], [Pg.152], [Pg.155]

[40] N. Kandasamy, J. P. Hayes, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Transactions on Computers*, 52(2):113–125, 2003. [Pg.2]

[41] N. Kandasamy, J. P. Hayes, and B. T. Murray. Dependable communication synthesis for distributed embedded systems. *Reliability Engineering & System Safety*, 89(1):81–92, 2005. [Pg.66], [Pg.97], [Pg.129], [Pg.159], [Pg.187]

[42] H. Kanemitsu, M. Hanada, and H. Nakazato. Prior node selection for scheduling workflows in a heterogeneous system. *Journal of Parallel and Distributed Computing*, 109:155–177, 2017. [Pg.29]

[43] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40. IEEE, 2014. [Pg.174]

[44] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999. [Pg.5]

[45] J. Lee, B. Yun, and K. G. Shin. Reducing peak power consumption inmulti-core systems without violatingreal-time constraints. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):1024–1033, 2013. [Pg.26]

[46] Y. C. Lee and A. Y. Zomaya. Energy conscious scheduling for distributed computing systems under different operating conditions. *IEEE TPDS*, 22(8):1374–1381, 2010. [Pg.174]

[47] K. Li. Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers. *IEEE Transactions on Computers*, 61(12):1668–1681, 2012. [Pg.5], [Pg.6], [Pg.30]

[48] K. Li. Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels. *J. of Parallel and Distrib. comput.*, 95:15–28, 2016. [Pg.5], [Pg.6], [Pg.30]

[49] M. Li, B. J. Liu, and F. F. Yao. Min-energy voltage allocation for tree-structured tasks. *Journal of Combinatorial Optimization*, 11(3):305–319, 2006. [Pg.30]

[50] Z. Li, J. Ge, H. Hu, W. Song, H. Hu, and B. Luo. Cost and energy aware scheduling algorithm for scientific workflows with deadline constraint in clouds. *IEEE TSC*, 11(4):713–726, 2015. [Pg.29], [Pg.30]

[51] J. Liu, K. Li, D. Zhu, J. Han, and K. Li. Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(2):1–25, 2016. [Pg.1]

[52] S. Maity, A. Ghose, S. Dey, and S. Biswas. Thermal-aware adaptive platform management for heterogeneous embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–28, 2021. [Pg.8], [Pg.26], [Pg.34], [Pg.161]

[53] G. M. Mancuso, E. Bini, and G. Pannocchia. Optimal priority assignment to control tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):1–17, 2014. [Pg.71]

[54] A. K. Maurya and A. K. Tripathi. On benchmarking task scheduling algorithms for heterogeneous computing systems. *The Journal of Supercomputing*, 74(7):3039–3070, 2018. [Pg.29]

[55] S. Moulik, R. Chaudhary, and Z. Das. Hears: A heterogeneous energy-aware real-time scheduler. *Microprocessors and Microsystems*, 72:102939, 2020. [Pg.164]

[56] S. Moulik, R. Devaraj, and A. Sarkar. Healers: a heterogeneous energy-aware low-overhead real-time scheduler. *IET Computers & Digital Techniques*, 13(6):470–480, 2019. [Pg.105]

[57] S. Moulik, A. Sarkar, and H. K. Kapoor. Tarts: A temperature-aware real-time deadline-partitioned fair scheduler. *Journal of Systems Architecture*, 112:101847, 2021. [Pg.xxvi], [Pg.8], [Pg.28], [Pg.34], [Pg.164], [Pg.175], [Pg.177], [Pg.183]

[58] W. Munawar, H. Khdr, S. Pagani, M. Shafique, J.-J. Chen, and J. Henkel. Peak power management for scheduling real-time tasks on heterogeneous many-core systems. In *2014 20th IEEE international conference on parallel and distributed systems (ICPADS)*, pages 200–209. IEEE, 2014. [Pg.26]

[59] A. Nadeem and M. Javed. A performance comparison of data encryption algorithms. In *2005 International Conference on Information and Communication Technologies*, pages 84–89, 2005. [Pg.138]

[60] E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards high performance cryptographic software. In *Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 69–72, 1995. [Pg.138]

[61] G. Nelissen. *Efficient optimal multiprocessor scheduling algorithms for real-time systems*. PhD thesis, Université libre de Bruxelles, 2012. [Pg.17]

[62] A. Nelson and K. Goossens. Distributed power management of real-time applications on a gals multiprocessor soc. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 147–156. IEEE, 2015. [Pg.2]

[63] 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)*, pages 13–24. IEEE, 2018. [Pg.195]

[64] J. Pérez Rodríguez and P. Meumeu Yomsi. An efficient proactive thermal-aware scheduler for dvfs-enabled single-core processors. In *29th International Conference on Real-Time Networks and Systems*, pages 144–154, 2021. [Pg.8], [Pg.34]

[65] L. L. Peterson and B. S. Davie. *Computer networks: a systems approach*. Elsevier, 2007. [Pg.3]

[66] I. Ripoll and R. Ballester-Ripoll. Period selection for minimal hyperperiod in periodic task systems. *IEEE Transactions on Computers*, 62(9):1813–1822, 2012. [Pg.118]

[67] S. K. Roy, R. Devaraj, and A. Sarkar. Contention cognizant scheduling of task graphs on shared bus based heterogeneous platforms. *IEEE TCAD*, 2021. [Pg.102], [Pg.118]

[68] S. K. Roy, R. Devaraj, A. Sarkar, K. Maji, and S. Sinha. Contention-aware optimal scheduling of real-time precedence-constrained task graphs on heterogeneous distributed systems. *Journal of Systems Architecture*, 105:101706, 2020. [Pg.102], [Pg.118]

[69] S. K. Roy, R. Devaraj, A. Sarkar, and D. Senapati. Slaqa: Quality-level aware scheduling of task graphs on heterogeneous distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5):1–31, 2021. [Pg.33], [Pg.159], [Pg.181], [Pg.184]

[70] I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118(2):390–412, 1999. [Pg.2]

[71] S. Saha, Y. Lu, and J. S. Deogun. Thermal-constrained energy-aware partitioning for heterogeneous multi-core multiprocessor real-time systems. In *2012 IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 41–50. IEEE, 2012. [Pg.xxvi], [Pg.175], [Pg.177]

[72] A. Sarkar. *New Approaches in Real-Time Proportional Fair Multi-Processor Scheduling*. PhD thesis, IIT Kharagpur, 2012. [Pg.2], [Pg.32]

[73] D. Senapati, A. Sarkar, and C. Karfa. Presto: A penalty-aware real-time scheduler for task graphs on heterogeneous platforms. *IEEE Transactions on Computers*, 71(2):421–435, 2022. [Pg.33], [Pg.170]

[74] D. Senapati, A. Sarkar, and C. Karfa. Energy-aware real-time scheduling of multiple periodic dags on heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(8):2447–2460, 2023. [Pg.181], [Pg.184]

[75] Y. Sharma, S. Chakraborty, and S. Moulik. Eta-hp: an energy and temperature-aware real-time scheduler for heterogeneous platforms. *The Journal of Supercomputing*, 78:1–25, 2022. [Pg.8], [Pg.34]

[76] H. F. Sheikh and I. Ahmad. Fast algorithms for thermal constrained performance optimization in dag scheduling on multi-core processors. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011. [Pg.2], [Pg.8], [Pg.34]

[77] H. F. Sheikh and I. Ahmad. Fast algorithms for simultaneous optimization of performance, energy and temperature in dag scheduling on multi-core processors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1–7, 2012. [Pg.8], [Pg.34]

[78] H. F. Sheikh, I. Ahmad, Z. Wang, and S. Ranka. An overview and classification of thermal-aware scheduling techniques for multi-core processing systems. *Sustainable Computing: Informatics and Systems*, 2(3):151–169, 2012. [Pg.8], [Pg.33]

[79] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained hetero processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993. [Pg.2]

[80] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE heterogeneous computing workshop (HCW'96)*, pages 86–97, 1996. [Pg.27]

[81] K. Skadron. Hybrid architectural dynamic thermal management. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 10–15. IEEE, 2004. [Pg.161]

[82] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *ACM SIGARCH Computer Architecture News*, 31(2):2–13, 2003. [Pg.33], [Pg.161]

[83] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988. [Pg.1], [Pg.17]

[84] 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.*, 14(1):55–74, 2016. [Pg.4], [Pg.30], [Pg.31]

[85] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002. [Pg.1], [Pg.2], [Pg.5], [Pg.18], [Pg.28], [Pg.36], [Pg.50], [Pg.51], [Pg.54], [Pg.71], [Pg.87], [Pg.89], [Pg.102], [Pg.118], [Pg.152], [Pg.154], [Pg.162], [Pg.184]

[86] TSN. Time-Sensitive Networking (TSN). *https://en.wikipedia.org/wiki/Time-Sensitive_Networking*, 2012. [Pg.195]

[87] J. D. Ullman. Np-complete scheduling problems. *J. of Computer and System sciences*, 10(3):384–393, 1975. [Pg.2], [Pg.27], [Pg.29]

[88] O. S. Unsal and I. Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, 2003. [Pg.105]

[89] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. Anytime pack search. *Natural Computing*, 15(3):395–414, 2016. [Pg.71]

[90] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. *Mobile Computing*, pages 449–471, 1994. [Pg.30]

[91] F. Wu, Q. Wu, and Y. Tan. Workflow scheduling in cloud: a survey. *J. Supercomput.*, 71(9):3373–3418, 2015. [Pg.29]

[92] H. Wu, J. Jaffar, and R. Yap. A fast algorithm for scheduling instructions with deadline constraints on risc machines. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 281–290. IEEE, 2000. [Pg.28]

[93] Q. Wu, F. Ishikawa, Q. Zhu, Y. Xia, and J. Wen. Deadline-constrained cost optimization approaches for workflow scheduling in clouds. *IEEE TPDS*, 28(12):3401–3412, 2017. [Pg.2], [Pg.28], [Pg.29]

[94] T. Xiaoyong, K. Li, Z. Zeng, and B. Veeravalli. A novel security-driven scheduling algorithm for precedence-constrained tasks in heterogeneous distributed systems. *IEEE Transactions on Computers*, 60(7):1017–1029, 2011. [Pg.7], [Pg.8], [Pg.28], [Pg.32], [Pg.33]

[95] 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 TII*, 13(3):1068–1078, 2017. [Pg.2], [Pg.4], [Pg.6], [Pg.28], [Pg.30], [Pg.31], [Pg.98], [Pg.99], [Pg.106]

[96] G. Xie, X. Xiao, H. Peng, R. Li, and K. Li. A survey of low-energy parallel scheduling algorithms. *IEEE Transactions on Sustainable Computing*, 7(1):27–46, 2022. [Pg.164]

[97] G. Xie, K. Yang, H. Luo, R. Li, and S. Hu. Reliability and confidentiality co-verification for parallel applications in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1353–1368, 2021. [Pg.133]

[98] G. Xie, G. Zeng, R. Li, and K. Li. Energy-aware processor merging algorithms for deadline constrained parallel apls in heterogeneous cloud computing. *T-SUSC*, 2(2):62–75, 2017. [Pg.30], [Pg.31], [Pg.161]

[99] G. Xie, G. Zeng, X. Xiao, R. Li, and K. Li. Energy-efficient scheduling algorithms for real-time parallel applications on heterogeneous distributed embedded systems. *IEEE Trans. Parallel Distrib. Syst.*, 28(12):3426–3442, 2017. [Pg.4], [Pg.6], [Pg.30], [Pg.66], [Pg.105], [Pg.106], [Pg.121], [Pg.122], [Pg.161]

[100] T. Xie and X. Qin. A new allocation scheme for parallel applications with deadline and security constraints on clusters. *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005. [Pg.153]

[101] T. Xie and X. Qin. Scheduling security-critical real-time applications on clusters. *IEEE Transactions on Computers*, 55(7):864–879, 2006. [Pg.7], [Pg.8], [Pg.28], [Pg.32], [Pg.153]

[102] T. Xie and X. Qin. Improving security for periodic tasks in embedded systems through scheduling. *ACM Trans. Embed. Comput. Syst.*, 6(3):20–es, July 2007. [Pg.7], [Pg.8], [Pg.28], [Pg.32], [Pg.33]

[103] T. Xie and X. Qin. Performance evaluation of a new scheduling algorithm for distributed systems with security heterogeneity. *Journal of Parallel and Distributed Computing*, 67(10):1067–1081, 2007. [Pg.7], [Pg.8], [Pg.28], [Pg.32]

[104] T. Xie and X. Qin. Security-aware resource allocation for real-time parallel jobs on homogeneous and heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):682–697, 2008. [Pg.7], [Pg.8], [Pg.28], [Pg.32]

[105] T. Xie, X. Qin, A. Sung, M. Lin, and L. T. Yang. Real-time scheduling with quality of security constraints. *International Journal of High Performance Computing and Networking*, 4(3-4):188–197, 2006. [Pg.7], [Pg.8], [Pg.28], [Pg.32]

[106] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of IEEE 36th annual foundations of computer science*, pages 374–382. IEEE, 1995. [Pg.30]

[107] L.-T. Yeh, R. Chu, and W. Janna. Thermal management of microelectronic equipment: Heat transfer theory, analysis methods, and design practices. asme press book series on electronic packaging. *Appl. Mech. Rev.*, 56(3):B46–B48, 2003. [Pg.161]

[108] H.-S. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2(3):393–430, 2003. [Pg.30]

[109] B. M. H. Zade, N. Mansouri, and M. M. Javidi. Saea: A security-aware and energy-aware task scheduling strategy by parallel squirrel search algorithm in cloud environment. *Expert Systems with Applications*, 176:114915, 2021. [Pg.138]

[110] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. Carb: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters*, 16(1):6–9, 2016. [Pg.174]

[111] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 14–pp. IEEE, 2006. [Pg.7], [Pg.31]

[112] Y. Zhao, S. Cao, and L. Yan. List scheduling algorithm based on pre-scheduling for heterogeneous computing. In *IEEE Intl Conf on Parallel & Dist. Processing with Appl, BDCloud, SustainCom, SocialCom*, pages 588–595. IEEE, 2019. [Pg.5], [Pg.29], [Pg.36], [Pg.50], [Pg.54], [Pg.87], [Pg.89], [Pg.162], [Pg.184]

[113] J. Zhou, K. Cao, P. Cong, T. Wei, M. Chen, G. Zhang, J. Yan, and Y. Ma. Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms. *Journal of Systems and Software*, 133:1–16, 2017. [Pg.8], [Pg.34]

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India