# List Scheduling Algorithms for Open Distributed Real-Time Embedded Systems

DISSERTATION

To Obtain the Degree of Doctor of Engineering

Submitted By

Sarah Amin

Submitted To

Chair of Embedded Systems

Department of Elektrotechnik und Informatik

The University of Siegen

Siegen 2020

Supervisor And First Appraiser

Prof. Dr. Roman Obermaisser

University of Siegen

Second Appraiser

Prof. Dr. Peter Puschner

Vienna University of Technology

Date of the Oral Examination

20. May 2020

Take care of the minutes and the hours will take care of themselves.

*Earl of Chesterfield*

I dedicate this work to my beloved parents, and my aunt, Rubina Shafique, who have always supported my innate thrive for knowledge and have always encouraged me to do my best.

## ACKNOWLEDGEMENTS

I would like to thank Allah Almighty for giving me the opportunity, determination and strength to do my research. His continuous grace and mercy was with me throughout my life and ever more during the tenure of my research.

I would like to express my deep and sincere gratitude to my supervisor Prof. Dr Roman Obermaisser for his continuous support, guidance and encouragement. In addition to being an excellent supervisor, he is a man of principles and has immense knowledge of research in general and his subject in particular. I appreciate all his contributions of time, support and ideas.

I would also like to thank the staff of the Chair of Embedded Systems, specifically Manuela Popp and Stefan Otterbach. I am grateful to my colleagues and friends, specially Nadra, Maryam, Adele and Setareh, for their continued guidance, support and friendship.

Very special gratitude goes out to my eternal Cheerleaders, my siblings, who sacrificed a lot to help me fulfil my dream. Finally, I am grateful to my aunt, and my parents who have provided me through emotional and moral support in my life.

# ABSTRACT

In recent years, the field of embedded systems has evolved towards application areas that combine stringent real-time constraints, reliability requirements and a need for an open-world assumption. Such systems are called Open Distributed Real-time Embedded (ODRE) systems. These systems are based on an open-world assumption where new components enter at runtime to dynamically realise emerging services. At the same time, reliable operation and support for stringent real-time requirements are essential to support closed-loop control and guaranteed response times. In such systems, time-triggered scheduling of the services before executing them on the system ensures the correct temporal behaviour of the applications and guarantees support for stringent real-time constraints. However, scheduling a real-time application in an ODRE system is complicated because of its dynamic nature. The structure of the ODRE system is continuously changing, which means that a fixed schedule generated at the design time cannot be used throughout the lifetime of the system. The execution times of prevalent scheduling algorithms, e.g, evolutionary algorithms, mixed integer linear programming, satisfiablity modulo theories, are not suitable for invocation at runtime. Moreover, these algorithms make assumptions, e.g., assuming a bus-based communication network instead of multi-hop communication networks or considering the same period for all the application tasks, that are unrealistic for the stringent real-time requirements of ODRE systems. Therefore, there is a need for a scheduling algorithm that computes a feasible schedule for ODRE systems at runtime whenever there are changes in the system. This thesis proposes list scheduling algorithms that consider both stringent timing constraints and openness of the ODRE system while computing a feasible schedule with short scheduling delay at runtime. The proposed algorithms are generic and support the computation of a time-triggered schedule for any system application on an ODRE system. In addition, this thesis demonstrates the models and algorithms for scheduling diagnostic services for fault detection and diagnosis in ODRE systems. The scheduling algorithms are evaluated based on different parameters such as, the size of the system application, the number of available resources, the network topology used in the system, the number of modifications in the scheduled application, and reconfiguration cost of the system. The results show that the algorithms can compute a feasible schedule provided that there are enough resources in the system. The results for the proposed incremental scheduling algorithm show that the computed schedules are scalable to changes in the system and the revalidation efforts can be reduced by minimising the changes in the already scheduled application. Furthermore, the time complexity and the observed runtime of the algorithms shows that they can be invoked during runtime when the scheduler is triggered by a change in the system configuration.

## Zusammenfassung

In den letzten Jahren hat sich der Bereich der eingebetteten Systeme zu Anwendungsbereichen entwickelt, die strenge Echtzeitbeschränkungen, Zuverlässigkeitsanforderungen und die Notwendigkeit einer Open-World-Annahme vereinen. Solche Systeme werden als Open Distributed Real-time Embedded (ODRE) Systeme bezeichnet. Diese Systeme basieren auf einer Open-World-Annahme, bei der neue Komponenten zur Laufzeit auftreten, um neu entstehende Dienste dynamisch zu realisieren. Gleichzeitig sind ein zuverlässiger Betrieb und die Unterstützung von strengen Echtzeitanforderungen für die Unterstützung von geschlossenen Regelkreisen und garantierten Reaktionszeiten unerlässlich. In solchen Systemen sorgt ein zeitgesteuertes Scheduling der Dienste vor der Ausführung auf dem System für das korrekte zeitliche Verhalten der Anwendungen und garantiert die Unterstützung von strengen Echtzeitbedingungen. Das Scheduling einer Echtzeitanwendung in einem ODRE-System ist jedoch aufgrund seiner dynamischen Natur kompliziert. Die Struktur des ODRE-Systems ändert sich ständig, was bedeutet, dass ein zur Designzeit generierter fester Zeitplan nicht über die gesamte Lebensdauer des Systems verwendet werden kann. Die Ausführungszeiten gängiger Scheduling-Algorithmen, z.B. evolutionäre Algorithmen, gemischt-ganzzahlige lineare Programmierung, Satisfiablity-Modulo-Theorien, sind für die Verwendung zur Laufzeit nicht geeignet. Darüber hinaus machen diese Algorithmen restriktive Annahmen, z.B. die Annahme eines busbasierten Kommunikationsnetzes anstelle von Multi-Hop-Kommunikationsnetzen oder die Berücksichtigung der gleichen Periode für alle Anwendungsaufgaben, die für die Echtzeitanforderungen von ODRE-Systemen unrealistisch sind. Daher besteht ein Bedarf an einem Scheduling-Algorithmus, der zur Laufzeit einen realisierbaren Zeitplan für ODRE-Systeme berechnet, wenn es Änderungen im System gibt. Diese Arbeit schlägt Listen-Scheduling-Algorithmen vor, die sowohl strenge Zetbedingungen als auch die Offenheit des ODRE-Systems berücksichtigen, während sie einen Zeitplan mit kurzer Planungsverzögerung zur

Laufzeit berechnen. Die vorgeschlagenen Algorithmen sind generisch und unterstützen die Berechnung eines zeitgesteuerten Zeitplans für verschiedene Systemanwendungen auf einem ODRE-System. Darüber hinaus werden in dieser Arbeit die Modelle und Algorithmen für das Scheduling von Diagnosediensten zur Fehlererkennung und Diagnose in ODRE-Systemen demonstriert. Die Scheduling-Algorithmen werden auf der Basis verschiedener Parameter, wie z.B. der Größe der Systemanwendung, der Anzahl der verfügbaren Ressourcen, der im System verwendeten Netzwerktopologie, der Anzahl der Änderungen in der geplanten Anwendung und der Rekonfigurationskosten des Systems, bewertet. Die Ergebnisse zeigen, dass die Algorithmen einen korrekten Zeitplan berechnen können, sofern genügend Ressourcen im System vorhanden sind. Die Ergebnisse für den vorgeschlagenen inkrementellen Scheduling-Algorithmus zeigen, dass die berechneten Zeitpläne auf Änderungen im System skalierbar sind und der Aufwand für die Revalidierung durch Minimierung der Änderungen in der bereits eingeplanten Anwendung reduziert werden kann. Darüber hinaus zeigt die Analyse der Zeitkomplexität und der beobachteten Laufzeit der Algorithmen, dass sie zur Laufzeit aufgerufen werden können, wenn der Scheduler durch eine Änderung der Systemkonfiguration ausgelöst wird.

# TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

THIS THESIS deals with the specific challenges related to the system-level design of open distributed real-time embedded systems implemented with time-triggered task sets that communicate over time-triggered communication protocols. The specific focus is on proposing an efficient heuristic that can compute a feasible schedule during runtime whenever there are changes in the system.

## 1.1 Background

In recent years, the field of embedded systems has evolved towards application areas that combine stringent real-time constraints, reliability requirements and a need for an open-world assumption. These systems are called Open Distributed Real-Time Embedded (ODRE) systems [1]. The open-world assumption means that the system structure is dynamic, and components may enter or leave at runtime. In other words, the structure of the system is not completely known at the design time, and changes may occur during the functioning of the system. Examples of ODRE systems are Ambient-Assisted Living (AAL) for elderly care [2], networked medical devices and health management systems [2], command/control systems [3, 4], and applications for electrical power distribution [5]. In such systems, new components are integrated at runtime to realise emerging global services. At the same time, reliable operations and stringent real-time requirements are essential to support closed-loop control and guaranteed response times. For example, a physician needs to dynamically integrate medical devices into an in-home AAL system for emergency treatment. In this scenario, the device should be integrated to the in-home AAL system such that it gives a predictable response time and can interact reliably with the in-home medical devices (e.g., sensors) and remote sites (e.g., the hospital).

In safety-critical systems, an embedded computer system has to provide its services with dependability that is better than the dependability of any of its constituent components [6]. If the failure rate data of available electrical components is considered, then this level of dependability can only be achieved if the system supports fault-tolerance. An essential step in fault treatment is fault diagnosis, which determines the causes of failures in terms of nature and localisation. Root cause analysis is used to trace back the abnormal behaviours and states to the originating fault. A component can fail due to a design fault in software or hardware, a transient or permanent physical hardware fault or an operator mistake. The faults must be detected and diagnosed, and recovery actions must be proposed within a predictable time to avoid disastrous and life-threatening situations. ODRE systems can be safety-critical, i.e., failure or malfunction of such systems may lead to death or serious injuries, loss/damage of the property or environmental harm. For example, consider an in-home AAL medical device that monitors the oxygen input to a critical elderly patient. In this case, if the device is unable to detect a malfunction in one of its components within the systemFLs defined time-bound and the oxygen pressure is increased beyond the recommended value, then the patient can die due to oxygen toxicity in his/her blood. Therefore, it is essential that ODRE systems also offer diagnostic services to diagnose and detect faults at runtime. The diagnostic method should cope with dynamic system structures, should follow stringent timing constraints and should be reliable.

Scheduling the diagnostic service before executing it ensures a predictable temporal behaviour of the diagnostic application and also bounds the time to detect and diagnose faults. Preplanned execution of the diagnostic application infers the maximum time to detect and diagnose faults in the system. However, scheduling a real-time application in an ODRE system is complicated by its dynamic nature. The structure of the ODRE system is not entirely known before runtime, which means that a fixed schedule generated at the design time cannot be used throughout the functioning of the system. The tasks and communication messages need to be reallocated and rescheduled whenever there is a change in the system structure or system application. After that, the system can switch to this new schedule, and the message and task dispatching can proceed according to the new schedule. The scheduling algorithm should, therefore, be fast enough to recompute a schedule during runtime and should ensure continuity of service to avoid interruption of services that are not affected by the changes and to minimise changes in the system to preserve previous validation results and safety arguments. Dynamic scheduling is not desirable due to the higher runtime overhead in particular with task dependencies and usage of shared resources. In contrast, time-triggered schedules can exploit implicit synchronisation to satisfy precedence constraints, avoid resource conflicts and race conditions. In addition, dynamic scheduling can result in the loss of temporal determinism [7]. Therefore, a time-triggered scheduling algorithm is required that recomputes a feasible schedule, whenever there are changes in the system, within a predictable time. There should also be a fail-safe mechanism in the system that discards the changes if no feasible schedule is found.

## 1.2 Research Statement and Thesis Contributions

In the state-of-the-art, there are few time-triggered scheduling algorithms that support both stringent timing constraints and the open-world assumptions of distributed real-time embedded systems. Existing algorithms for time-triggered systems (cf. Chapter 3) support dynamic changes by pre-computing the schedules for all potential changes offline rather than during runtime of the system. The computation time of most of these algorithms is not suitable for invocation at runtime, or they compute schedules for the real-time tasks with unrealistic assumptions (e.g., without considering the communication constraints or considering the same period for all the application tasks). This thesis proposes list scheduling algorithms that consider both stringent timing constraints and the openness of the ODRE system while computing a schedule. The proposed algorithms are designed to schedule any system application on an ODRE system. In addition, this thesis exploits the scheduling of a diagnostic application for fault detection and diagnosis in ODRE systems to test the validity and effectiveness of the proposed algorithms.

Given an ODRE system and a system application, we compute a feasible static schedule for the time-triggered execution of application tasks and message communications at runtime. The schedule is scalable to changes in the ODRE system, and ensures continuity of service and minimises changes in the system to preserve previous validation results and safety arguments. The faults are detected and diagnosed in the ODRE system using diagnostic multi-queries that are realised in a real-time database and executed on pervasive SQL [8, 9]. If a feasible schedule is not found, then the system should keep running the old schedule and discard the changes.

The main contributions of this thesis are as follows,

- This thesis proposes semi-static list scheduling algorithms that are executable during runtime and recompute a schedule whenever there are changes in the system.

- This thesis proposes list scheduling algorithms that consider both processor and communication bandwidth constraints while mapping and allocating tasks onto processors, i.e., the communication overhead and routing constraints of the network are considered while scheduling tasks onto processors. Furthermore, there is no bandwidth sharing between communication messages and the conflicts are solved by scheduling them onto paths.

- The proposed incremental list scheduling algorithm creates an execution and communication plan for the changes in the system while minimising the modifications in the already scheduled application. The minimisation of the modifications maximises the continuity of services and preserves existing validation and certification results.

- The evaluation and demonstration of the algorithms is done by scheduling the proposed diagnostic application to detect faults in ODRE systems. Our diagnostic application uses diagnostic multi-queries to identify and locate faults in the system. The diagnostic application graph is an enhanced form of the traditional scheduler input, i.e., the directed acyclic graph. In addition to depicting the characteristics of the diagnostic tasks, i.e., time-periods, computation costs, and the precedence constraints between them, the application graph also depicts the relationship between multiple iterations of two dependent tasks. The multi-iteration relationships between the tasks in the application determine what data must be discarded from the real-time database that bounds the total data consumption. It also depicts the real-time requirements by providing the maximum delay between the execution of two dependent tasks.

## 1.3 Thesis Overview

The remaining part of the thesis is structured as follows. Chapter 2 presents the basic concepts used throughout the thesis. In Chapter 3, the reader is acquainted with the state-of-the-art already presented in the considered field. Chapter 4 presents the characteristics of the system model, i.e. the system architecture and the system application. Chapter 5 and 6 propose list scheduling algorithms to schedule diagnostic graphs in homogeneous and heterogeneous distributed real-time embedded systems, respectively. Chapter 7 presents an incremental list scheduling algorithm that schedules the changes in the system while minimising the modifications to the already scheduled application. Chapter 8 presents a test case scenario to diagnose and detect faults in heat, ventilation and cooling systems using the list scheduling algorithm proposed in Chapter 5. Finally, Chapter 9 draws some conclusions and gives a summary of the presented work.

# CHAPTER 2
# BASIC CONCEPTS

THIS CHAPTER presents the basic concepts used throughout this thesis. It presents the framework of distributed real-time embedded systems, fault detection and diagnosis (FDD) in real-time embedded systems, characteristics of open distributed real-time embedded systems (ODRE), and finally it explains different real-time scheduling problems along with scheduling in time-triggered systems.

## 2.1 Distributed Real-Time Embedded Systems

A distributed real-time embedded system is a computer system that is designed to perform dedicated functionality and is embedded in a larger device whose components are connected through a network, and that must guarantee a response within strict time constraints [10]. A distributed real-time embedded system is the combination of embedded, real-time and distributed domain. This section explains the concepts and characteristics of these domains.

### 2.1.1 Embedded Systems

In the computing domain, the term embedded system refers to an electronic system that is designed to perform dedicated functionality and is often embedded within a larger system [11]. An embedded system is generally a combination of computer hardware and software components. In contrast to general-purpose computing devices, an embedded system is designed for a specific function and is usually built together with the software intended to run on it. This model of developing hardware and software collectively is called hardware-software co-design [11].

Embedded systems are widely used in safety-critical applications such as aerospace/avionics, railway, automotive industry and medical equipment. Such systems are quite often also called real-time embedded systems since they are obliged to perform certain tasks in a limited amount of time. The failure to comply with the timing constraints has consequences whose gravity varies from a gradual loss of quality in an MPEG decoder to catastrophic events, e.g., fatal car crashes when the braking system fails to react in time [12].

### 2.1.2 Real-Time Systems

Real-time is a quantitative measurement of time measured using a physical clock. For example, consider a chemical power plant, where the system shuts down the heater within 25 ms when the temperature reaches 260° C. The 25 ms are measured using a physical clock present on the plant. A real-time system means that the system is subjected to real-time i.e. the correctness of the system behavior not only depends upon the logical results of the system but also on the physical time when these results are produced.

#### *2.1.2 Classification of Real-Time Systems*

Depending upon the nature of the timing constraints, real-time systems are classified into *hard* and *soft* real-time systems [11].

- *Soft real-time systems:* A timing constraint is termed *soft* if the consequences of missing a deadline are undesirable but tolerable. A soft real-time system offers *best-effort* services, i.e. it completes the service of a request within a known finite time but with an occasional missed deadline that is considered tolerable. Although missing a deadline does not have any dire consequences, but the overall quality of service is degraded. For example, a GPS in a car must remind the driver about a waypoint at a latency of 1.5s. If the system misses this deadline, the driver simply misses the waypoint without any dire consequences [11].

- *Hard real-time systems:* A timing constraint is termed *hard* if the consequences of missing a deadline are fatal. A hard real-time system offers *guaranteed* services, i.e. its service of a request is guaranteed to be completed within a strict deadline. In a hard real-time system, missing a deadline could result in catastrophic effects such as safety hazards or serious financial consequences. For example, a pacemaker is a small device that is placed inside the chest to help control abnormal heart rhythms. It uses electrical pulses to prompt the heart to beat at a normal rate. The deadline for generating a pulse for the ventricular beat after detecting an abnormal atrial beat is at least 0.1s and at most 0.2s. The consequence of missing this deadline is the loss of human life [11].

### *2.1.2 Characteristics of Hard Real-Time Embedded Systems*

A real-time system embedded within a larger system is considered a real-time embedded system. This thesis focuses on hard real-time embedded systems, also called safety-critical RT systems. These systems have the following characteristics [13]:

- *Response time:* The response time requirements for such systems are typically in the range of milliseconds or less and may result in a catastrophe if not met.

- *Peak load performance:* The peak load performance of such systems must be predictable and should not violate the predefined deadlines.

- *Concurrency:* Such systems are highly complex since they control multiple sensors and actuators at a time. Therefore, to avoid system failure, the system must remain synchronous with its environment at all times.

- *Safety and reliability:* Violating a deadline in such systems may result in loss of life, severe injury or environmental damage. Therefore, the components in such systems should be highly reliable with respect to critical failure modes to ensure safety.

- *Data integrity:* Temporal accuracy is a great concern in such systems. If the result is obtained after the deadline, then it is useless and is considered incorrect.

- *Fault tolerance:* Due to the stringent timing and safety requirements, these systems must support fault tolerance and the system must complete the execution of the tasks within their deadline even in the presence of faults.

### 2.1.3 Distributed Systems

A distributed system is classified as a collection of autonomous computers that appear as a single coherent system to its user who could either be an application or a human [14]. In a distributed system, there are multiple control units and each control unit is connected to a set of sensors and actuators that are local to it meaning that a control unit only handles its local sensors and actuators. Each control unit, combined with its sensors and actuators, is termed as a node. Multiple nodes are connected together to create a distributed system where each node provides services for the other nodes to use the nodeFLs resources, e.g., local sensors and actuators [15].

In order to support different control units and networks while offering a single system view, distributed systems are often organized by means of a layer of software that is placed between the higher-level layer of applications and the lower-level layer of operating systems and communication facilities. Such a software layer is termed as middleware, and it provides services beyond those provided by the operating system that enables various components of the distributed system to communicate and manage data. Although the hierarchical approach in these systems provides better scalability and reconfiguration, it also complicates the connection and communication between nodes. Since there is no shared memory pool between the nodes, the communication is usually performed by message passing that limits the choice of usable tools. Moreover, each node may have a different communication protocol or may run on a different operating system which complicates the coordination done by the middleware [15].

### *2.1.3 Characteristics of Distributed Systems*

While designing a distributed system, the following characteristics are considered [14]:

- A distributed system must make it easy for the applications to remotely access resources and must share them in a controlled and efficient manner, i.e., the resource conflicts should be solved without the application knowing about it.
- A distributed system that is able to present itself to applications as if it were a single system is called transparent distributed system. Although complete transparency is preferred for distributed systems, there are cases in which attempting to hide all the distributed aspects from the users is not a good idea. For example, a wide-area distributed system that connects a process from San Francisco to Berlin cannot hide the fact that the laws of physics will not allow it to send a message in less than 35 ms.
- A distributed system that offers services according to certain rules that describe the semantics of the offered services is termed as an open distributed system. In this case, the services are usually provided through interfaces described in Interface Definition Language (IDL). If correctly implemented, an interface definition allows an arbitrary process, that requires a certain interface, to communicate with a process that provides said interface. Proper specifications are complete and neutral, i.e., all the specifications that are needed for implementation are completely defined, and the specifications do not dictate what an implementation should look like. Completeness and neutrality are important for interoperability and portability. Interoperability characterizes the extent by which a component interacts with a component from a different manufacturer (implementation or access) without any restriction. Portability characterizes the extent to which an application designed for a system $A$ runs on a system $B$ without modification if system $B$ implements the same interface as $A$. Another important aspect for open distributed systems is that it should be easy to reconfigure the system, i.e., addition or removal of components should not affect the

untouched ones.

- A distributed system must be scalable. Scalability of a system can be measured in three different ways. A system can be scalable in size, i.e., more resources and application can be easily added to the system. Secondly, it can be administratively scalable, i.e., it is easy to manage the system even if it spans across multiple independent administrative organizations. Lastly, the system can be geographically scalable, i.e., it may have users or resources that may lie far apart.

## 2.2 Fault Diagnosis and Detection (FDD) in Safety-Critical Distributed Embedded Systems

In safety-critical systems, an embedded computer should provide its services with reliability that is higher than the reliability of any of its constituent components. Adding redundant components is one way to increase the reliability of such systems, but it is a costly solution. The other and more applicable solution is to make the systems fault-tolerant using implicit redundancy. An essential aspect of fault treatment is fault diagnosis that determines the cause, nature and locality of the occurring fault [16]. There are different ways faults can occur in a component. A component can fail because of a software or hardware design fault, a transient or permanent physical failure of the hardware or an operator mistake. The fault can be traced back to its source through root-cause analysis. There are two ways to diagnose and detect faults in distributed embedded systems.

### 2.2.1 Passive and Active FDD

- *Passive Diagnosis:* In passive fault diagnosis, the input-output data of the monitored system is measured, analyzed for faulty behaviour, and consequently, a decision for the fault is taken [17]. In such diagnosis techniques, the unusual behaviours and states of the components in the system are usually saved in an offline format and then analyzed by model-based, data-based or knowledge-based techniques. It means that

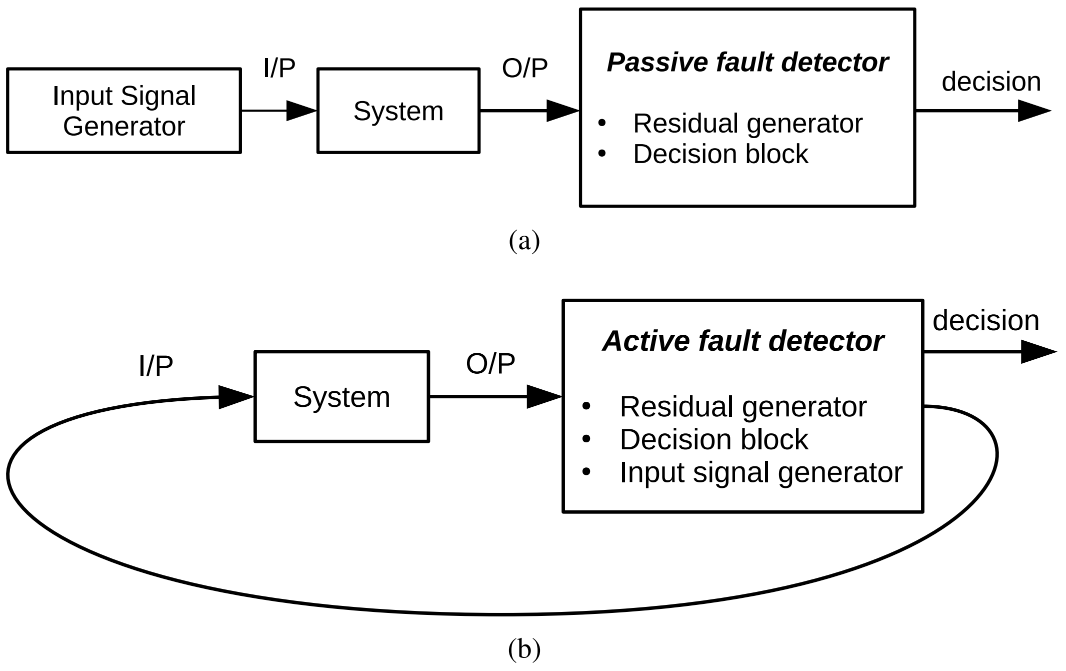


Figure 2.1: Block diagrams for passive and active fault diagnosis

the system is not informed about the fault directly after the failure of a component. Thus it does not raise the alarm or suggests a recovery action. Passive diagnosis is mostly used for maintenance and may not be feasible for fault detection and isolation in systems that have stringent-timing constraints and the possibility of complete failure within seconds of occurrence of the fault.

- *Active Diagnosis:* Contrary to passive diagnosis, an active diagnosis detector interacts with the monitored system by injecting a suitably designed auxiliary input signal to probe it for faults [17]. Such a technique monitors the system at runtime and proposes a suitable recovery action against the faults. This technique is better applicable in systems with stringent timings constraints to prevent their complete failure.

Fig. 2.1 shows the block diagrams for the two mentioned fault diagnosis and detection techniques [17].

### 2.2.2 Diagnosis requirements for Open Distributed Real-time Embedded (ODRE) systems

Since Open Distributed and Real-Time Embedded (ODRE) Systems support stringent timing constraints, continuous monitoring through active diagnosis is a more beneficial approach for ODRE systems than the passive diagnosis. ODRE systems have following fault detection and diagnosis requirements:

- *Open-world assumption:* Diagnosis needs to cope with the dynamic structures of the system where the components leave or enter at runtime, interact with each other at different levels and implement different global services. In this context, fixed variables cannot express diagnostic information. Upon the addition or extraction of a component from the system, the diagnostic technique should be able to express the relationships and change the diagnostic information accordingly. Similarly, the following recovery actions should also consider the dynamic nature of the system.

- *Real-time:* Active diagnosis must support the analysis of the diagnostic information and the subsequent recovery from a fault within a predictable time. Many control applications tolerate the loss of control inputs for only a few cycles; longer outages close off the system completely. For example, the maximum transient outage time for an automotive steer-by-wire application is 50 ms [18]. It means that the diagnostic approach should detect the fault, identify its location and propose a recovery action within a predictable time. Scheduling the diagnostic tasks before executing them on the system depicts the possible temporal behaviour of the diagnostic application. This apriori knowledge of the temporal behaviour bounds the time required to infer faults.

- *Reliability:* In case of active diagnosis, diagnostic information is used to achieve fault-tolerance by directly intervening in the system behaviour. Therefore, the diagnosis technique should be reliable enough to ensure that a fault affecting the diagnostic mechanisms does not cause incorrect recovery actions, e.g. restart the components that are working correctly. In this work, it is assumed that the proposed recovery actions are always correct.

## 2.3 Real-time Scheduling Problems

A real-time scheduling problem explains the conflicts due to the simultaneous access of processors by the tasks. A real-time scheduler usually takes its decisions based on the timing parameters of the *ready* tasks. A task is said to be *ready* when all its parent tasks have completed their execution [19]. The scheduling function is a service provided by an operating system, which allocates processors across time following the sequence of the ready tasks. The scheduler usually employs one or more scheduling algorithms to select the tasks for execution. Generally, the performance criteria of all such algorithms are to maximize the success of meeting the deadline. The arrangement of the tasks constructed by a scheduling algorithm is called a *task sequence*, or a *schedule* [20] and is usually represented by a Gantt chart.

### 2.3.1 Classification of Real-time Scheduling Algorithms

Real-time scheduling is divided into different categories depending upon various criteria, e.g. the scheduling rule and where to apply the scheduling rule. These characteristics can be classified as follows [20].

- *Preemptive vs Non-preemptive:* A scheduling algorithm is said to be preemptive if a running task can be interrupted any time to assign the processor to another task in the ready state according to the predefined scheduling function. Whereas, if the task once assigned frees the processor only upon its completion, then the scheduling algorithm is non-preemptive.

- *Dynamic vs Static:* A static (offline) scheduling algorithm generates the sequence on the system application before the system starts operation. Whereas, in a dynamic environment, the characteristics of the application (e.g., start-time of tasks) are not known apriori. Therefore, scheduling decisions are made during the execution of the task set.

- *Mono-processor vs Multiprocessor:* An algorithm is said to be mono-processor if it runs all the tasks on a single processor and multiprocessor if it schedules the tasks on multiple processors.
- *Idling vs Non-idling:* An algorithm is said to be idling if a processor waits to execute a task for a period even if it is free. Whereas if the processor starts executing the highest priority task when it is ready without insertion of idle time, then it is said to be non-idling.
- *Optimal vs Heuristic:* An algorithm is said to be optimal if it can find a feasible solution if one exists. Whereas the heuristic algorithm aims to provide a solution in a reasonable time frame that is good enough to solve the problem at hand. A heuristic solution is not necessarily always the optimal solution.
- *Centralized vs Distributed:* A centralized algorithm makes the scheduling decisions for the whole system at a central node irrespective of the fact that the system is distributed itself or not. In a distributed algorithm, the scheduling decisions are taken locally at each node.

In this thesis, a non-preemptive, semi-static, multiprocessor and centralized scheduling heuristic is proposed that searches for the best possible solution for the entire task set. The scheduler is semi-static because whenever there are changes in the system, the schedule is recomputed online, but for the whole task application at once.

### 2.3.2 Real-time Feasibility and Schedulability

Schedulability or feasibility analysis of a real-time system consists of checking whether all the tasks finish their execution within their deadline. The following properties of a scheduling algorithm are used to determine its feasibility [20].

- *Validity:* A schedule is said to be *valid* if the deadlines of all the tasks in the system application are met.
- *Optimal:* A scheduling algorithm is *optimal* if it finds a schedule that fulfills all the system requirements, i.e. the schedule is completely *valid*, within an estimated amount of time.
- *System feasibility:* A system is said to be *feasible* if there exists at least one schedule for the application aimed to be implemented on it.
- *Computational Complexity:* Computational or Time complexity of a scheduling algorithm describes the amount of time it takes to run the algorithm. It is used to measure the efficiency of the algorithm and the worst-case overhead incurred if the scheduler is executed at runtime.
  - An algorithm is said to be in *polynomial* time if its complexity function is $O(p(n))$ where $p$ is a polynomial function of a size $n$ task set e.g., $O(n^k)$ where $k$ is a constant. When $p$ is linear, the complexity is termed as linear complexity e.g., $O(n)$.
  - An algorithm is said to be in *pseudo-polynomial* time if its computation time is expressed as a polynomial function in terms of not only the size of the input but also the length of the input e.g., the *knapsack problem* has a time complexity of $O(n*W)$ where $n$ is the total number of items and $W$ is the maximum weight capacity of the knapsack.

– An algorithm is said to be in *exponential* time if its complexity function is *O(n!)* or $O(k^n)$, where $k > 1$ or $O(n^{log(n)})$, e.g., finding all distinct subsets of a set with *n* number of elements has a time complexity of $O(2^n)$.

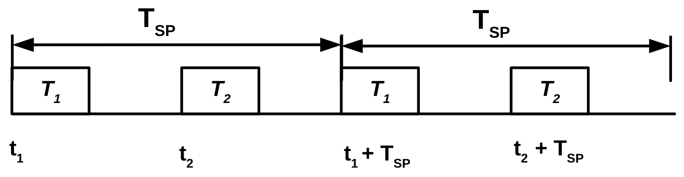
The higher is the algorithmic complexity, the greater is the overhead incurred by the on-line execution of the scheduler.

## 2.4 Time-triggered (TT) Systems

There are two fundamental paradigms for the design of embedded systems. In an event-triggered design, the system waits for triggers from the controlled objects, e.g. sensors to carry out particular activities. Whereas, in a time-triggered system, the system activities are carried out in predefined slots in a periodic manner. Time-triggered systems are preferred in safety-critical environments because they provide a high level of predictability and determinism [7]. With predefined time slots and the knowledge of the sequence of execution, the designer is ensured that all the stringent timing constraints are fulfilled. The fault diagnosis and detection model used for this work depends upon the diagnosis of faults in predictable time with the fulfilment of all deadlines; therefore, the safety-critical system is defined using a time-triggered design model. Thus, only the time-triggered execution of computation and communication tasks is considered.

### 2.4.1 TT Task Execution

A task is defined as ”a computation that is executed by the CPU in a sequential fashion” [21]. In this thesis, a task is a non-divisible sequential code segment or more specifically, a non-divisible diagnostic query. In a time-triggered system, the task execution starts at predetermined points in time. The main component of the real-time kernel is the timer interrupt routine, and the primary control signal is the clock of the system that is synchronized across all segments. The information needed for task execution is stored in a data structure called *schedule table* that contains the assigned start time of each task obtained through a static scheduling algorithm. This algorithm computes the schedule before it is deployed and thus eliminates any possible conflict between tasks by imposing appropriate start times [12].



| Task | Start-time | Time Period |
|---|---|---|
| $T_1$ | $t_1$ | $T_{SP}$ |
| $T_2$ | $t_2$ | $T_{SP}$ |

Figure 2.2: Time-Triggered Execution of Tasks

Consider Fig. 2.2, where two periodic tasks $T_1$ and $T_2$ are scheduled, each with a period $T_{SP}$ on a single processor. The table on the right side in the figure represents the *schedule table* and gives the start-times of each task. The start time of a task is computed offline such that it finishes its execution before the next start-time in the table. After a certain time $T_{SP}$ called the period of the static cyclic schedule, the kernel again performs the same set of decisions. If the tasks have different periods, then $T_{SP}$ is the least common multiple of the periods of all the individual tasks in the system. In such a case, the tasks have a different number of iterations within one $T_{SP}$ and the size of the *schedule table* is increased. This is discussed in later chapters.

### 2.4.2 Static Communication (TT Message Execution)

Similar to the time-triggered execution of tasks, static communication activities are initiated at predetermined moments. The communication activities are usually termed as messages between system nodes. For consistency, the clocks in all the nodes in such multiprocessor distributed systems must be synchronized to provide a global notion of time [7]. This synchronization is achieved through a defined communication protocol. In this section, the time-triggered communication mechanism is discussed as an example as it appears in a TDMA bus.

In a TDMA bus, the bandwidth is divided into different time-slots. Each such slot is

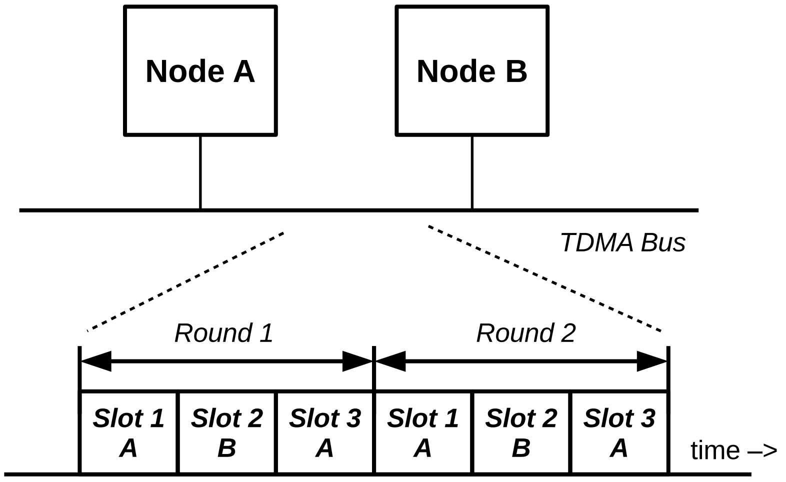


Figure 2.3: TDMA Bus Communication

then assigned offline to a node in the system. The node can only send messages on the bus during its assigned time-slot(s). The slots are ordered in a periodic sequence (i.e. the kernel repeats the assignment) called a TDMA round [12] e.g., in Fig. 2.3 shows TDMA rounds for a two-node system. The bandwidth is divided into three time-slots that collectively form a TDMA round, *slot 1*, *slot 2* and *slot 3*. Node A can transmit messages over the TDMA bus only during *slot 1* and *slot 3*, whereas node B can only transmit during *slot 2*. This way, it is guaranteed that only one node transmits on the bus at a time, and there are no conflicts.

| **Message ID** | **Start Time** | **Length** | **Sender** | **Receiver** |
|---|---|---|---|---|
| $m_{AB}$ | $t_1$ | $L_1$ | Node A | Node B |
| $m_{BA}$ | $t_2$ | $L_2$ | Node B | Node A |

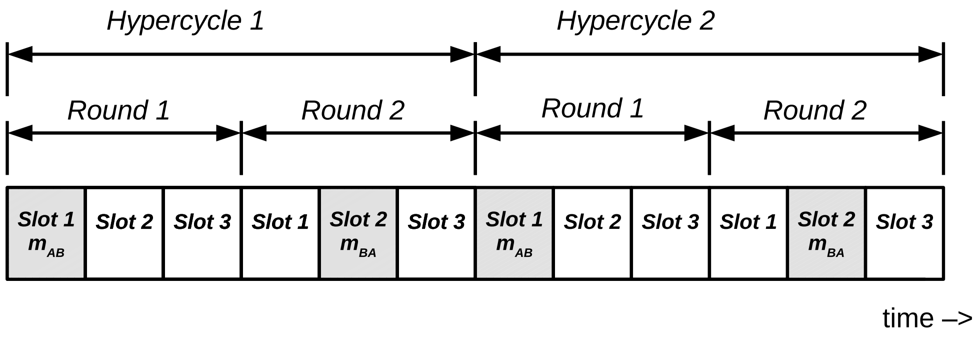


Figure 2.4: Static Time-Triggered (TT) Communication

A widely used TDMA based communication protocol is the Time-Triggered Protocol

(TTP) [22]. In the case of TTP, each node stores the information related to each of the messages in the system, e.g., sender/receiver nodes, the starting time of transmission and message length. A node will send a message whenever the current global time reaches one of the start times stored locally. For example, in Fig. 2.4 according to the information stored locally, Node A starts sending a message to Node B at $t_1$ during its predefined slot in the first bus round. At the same time, the communication controller at Node B knows from its local table that at time $t_1$ it has to start reading message $m_{AB}$. In the second bus round, at time $t_2$, a message from Node B to Node A is scheduled. The illustrated static schedule expands across two bus rounds, and the sequence of consecutive TDMA rounds is called a hypercycle. The static schedule stored locally at each node is repeated periodically with a period equal to the length of one hypercycle [12].

There are various advantages of time-triggered communication, e.g., timing properties of the system are guaranteed, and composability of the system is simple when extensions are planned [7]. This thesis considers time-triggered communication, e.g., TDMA, over different network topology and uses a heuristic to solve the conflicts by scheduling the application at run-time before its deployment.

## 2.5 Static Task Scheduling

When the characteristics of the system application, e.g., the execution time of the tasks are known apriori, then the scheduling problem is represented through a static model. The algorithm generates a schedule for the application before the start of the system, and the tasks are executed following this static schedule. Static task scheduling algorithms are divided into two groups (Fig. 2.5) [23], heuristic-based and guided random-search-based algorithms. Heuristic-based algorithms are further classified into, list scheduling, clustering and task duplication heuristics.

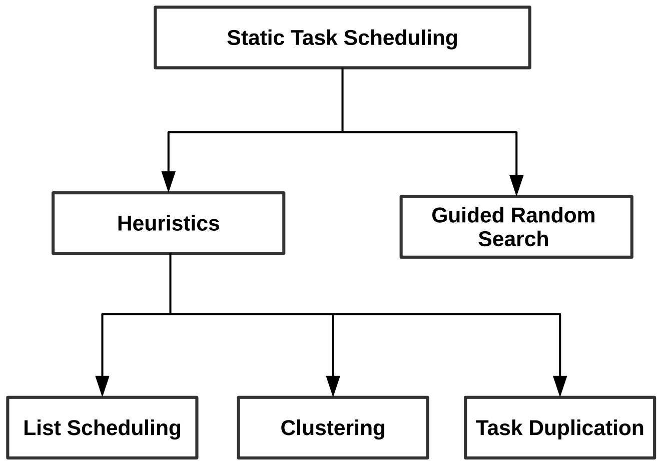


Figure 2.5: Classification of static task scheduling

### 2.5.1 List Scheduling

The most commonly used static heuristic to schedule tasks/messages in distributed systems is the traditional list scheduling algorithm [24]. It was first used with the assumption that the communication cost between tasks is zero, e.g., [25, 26, 27, 28]. In later works, it was modified to include non-zero communication cost between tasks, e.g., [29, 30, 31, 32]. List scheduling algorithm consists of two phases. In the first phase, the tasks are assigned priorities using a defined criterion, e.g. static level of the task in the application graph. The tasks are sorted into a list using the calculated priorities while respecting their precedence constraints. In the second phase, the algorithm schedules the highest priority ready task from the list to the processor that minimizes a defined cost function, e.g., the earliest start time of the task. In this general definition, it is assumed that the communication cost between tasks is zero. Most of the list scheduling algorithms are designed for a bounded number of fully connected homogeneous processor systems. They are generally more practical and provide better performance results at a lower computation time than the other algorithms in the group [23]. Since this thesis considers scheduling at runtime that requires lower computation time by the scheduler, therefore list scheduling is used as the starting heuristic. Some of the examples include Dynamic Level Scheduling [33], Modified Critical Path [34], Mapping Heuristic [35], Earliest Time First [29], Insertion-Scheduling Heuristic [30] and Dynamic Critical Path [36].

### 2.5.2 Clustering Heuristics

In an algorithm in this group, the tasks in the graph are mapped to an unlimited number of clusters. It is not necessary for the task selected for clustering to be a *ready* task. Each iteration refines the previous clustering by merging some clusters. All the tasks in a cluster are assigned to the same processor for scheduling. After clustering, the algorithm merges the clusters so that the number of clusters equals the number of processors. Each cluster is then assigned to a processor, and tasks within a processor are arranged according to their precedence constraints. Clustering heuristics are generally expensive to implement since their complexity and the workload on each processor increases with the increase in the number of tasks in each cluster. Some examples for clustering heuristics are, Linear Clustering Method [37], Mobility Directed [34], Dominant Sequence Clustering (DSC) [38], and Clustering and Scheduling System (CASS) [39].

### 2.5.3 Task Duplication Heuristics

In this algorithm, the interprocess communication overhead is reduced by mapping some of the tasks redundantly, i.e. a task is replicated and assigned to more than one processor. The algorithm avoids the transmission of the output to a successor task by executing the replication of the predecessor task on the same processor. The algorithms in this group have much higher complexity than the algorithms in the other groups and are usually used for an unbounded number of homogeneous processors [23]. A few examples are Critical Path Fast Duplication (CPFD) [40], Duplication Scheduling Heuristic (DSH) [30], Bottom-Up Top-Down Duplication Heuristic [41] and Duplication First and Reduction Next [42].

#### 2.5.4 Guided Random Search Techniques

Guided random search techniques use random choices to guide themselves through the problem space. These techniques combine the knowledge gained from previous search results with some randomized features to generate new results. Genetic Algorithms (GAs) [43, 44, 45] are the most widely used algorithms in this group. GAs generate optimal schedules. However, their computation time is much higher than the heuristic-based techniques. Also, the control parameters in a GA should be determined appropriately. Since a set of control parameters that work for one task graph may not give the best results for another graph. Other techniques in this group are simulated annealing [46, 47] and a local search method [48, 49]. Guided random search techniques generally require a lot of computation time and are thus not advisable for scheduling at runtime in safety-critical applications.

## 2.6 Incremental Design Process

Complex embedded systems with multiple processing elements have become a common occurrence in different fields like automotive electronics, telecommunication networks, networked medical devices, health management systems, command/control systems, nuclear power plants, and electrical power distribution. Such systems demand high performance, reliability, and cost-efficiency [50]. The design of these systems includes the selection of a distributed platform with different resources, such as processors and communication networks, and allocation (spatial and temporal) and scheduling of the system application onto the distributed platform. The process usually comprises an iterative execution of these steps until a solution is found that satisfies all the design constraints [51], [52], [53].

In the literature, several notable methods have been reported to facilitate the designers with the hardware/software design and co-synthesis of the embedded systems. The research mostly concentrates on designing a new system from scratch to accommodate a particular application. However, in many fields, it is more likely that a base system is modified to provide new functionality and specifications [50]. For example, in the automotive industry, companies usually develop different car models on the same technological platform. Models like Audi A3, Volkswagen Golf, and SEAT Leon share the same Modularer QuerBaukasten (MQB) Platform [54]. Each car model usually has multiple variations, i.e. different configuration settings depending upon the targeted market. In [55], the Volkswagen group proclaimed that they have 340 different model variants in their product portfolio. Therefore, it seems highly logical to design the system on the same technological platform to avoid extensive design costs and to reduce development and testing times.

Similarly, in the railway industry, the ongoing digitization of all technical systems has led to an increased number of functionalities being implemented in software on standard platforms rather than dedicated hardware domains. These software systems are connected, raising new demands of usability and security. Consequently, the technical systems get affected by changes in the technological infrastructure or user behaviour that necessitates re-validation or re-certification of the whole system [56]. An incremental verification approach is used to decrease the cost and effort of re-verification, where only the modified part of the system is validated. In this scenario, it is fundamental that there are no or as few modifications in the execution of the already running system application.

The system validation is a time-consuming and costly process, e.g., in the automotive industry, in case of a power-train, the product goes through the validation process in 5 out of 24 months of production [57]. However, the testing of solely the added functionality rather than the whole system considerably reduces the time required to introduce the product to the market, thus decreasing the development cost. Such an incremental design process is vital for current and future industrial practices. Since the time interval between the successive generations of products is reducing, but the complexity of the product is rapidly growing because of the increasing demand for new functionality and security.

The concept of incremental design process coincides with the open-world assumption of ODRE systems where the components leave or enter at runtime. This thesis proposes an incremental scheduling technique that deals with the dynamic nature of embedded systems intending to optimize the design cost and time for reconfiguring the system. The proposed method is applicable in all the areas of complex embedded systems that support the open-world assumption, i.e. dynamic integration of components at runtime.

# CHAPTER 3
# RELATED WORK

THIS CHAPTER gives an overview of the state-of-the-art related to the work presented in this thesis. The first three sections of the chapter introduce the literature review on scheduling tasks and communication messages in distributed embedded systems. The fourth section gives an overview of the work done in the field of incremental scheduling. The final section presents the different techniques used to detect and diagnose faults in embedded systems.

## 3.1 Static Task and Message Scheduling

One can distinguish dynamic and static scheduling algorithms for multi-core platforms. In static scheduling, tasks are statically allocated to cores during the design phase [58]. This activity results in decreasing the utilisation of computing resources as execution times may be lower than worst-case execution times (WCET). Static scheduling statically determines the sequences of tasks and is only applicable to Dataflow Graphs (DFGs) that have either Synchronous Dataflow (SDF) or Cyclo-Static Dataflow (CSDF) [59]. In static scheduling, decisions such as, selection of paths for message transmission, selection of processors for task executions, and start time for task executions, are made at compile-time [60]. For this purpose, a static schedule needs complete prior knowledge of the characteristics of the tasks, e.g., maximum execution time, precedence constraints, deadlines, and mutual exclusion constraints. [61, 62, 63]. However, in dynamic scheduling, all the decisions are made at runtime, and the algorithm considers only the current ready task set while making a decision. Dynamic schedulers are flexible and adapt to an evolving task scenario. However, the runtime effort involved in finding a feasible schedule is substantial [62, 63].

Many distributed real-time embedded systems assign computational activities statically to the processors. In these systems, constructing a schedule of the application statically before executing it depicts its temporal behaviour and guarantees fulfilment of deadlines and timing constraints. According to [63], static scheduling is widely used in safety-critical systems where missing a deadline leads to catastrophic consequences. According to [61], static scheduling can be divided into guided random search-based algorithms and heuristic-based algorithms. Heuristic-based scheduling algorithms use greedy heuristics that restrict the solution space to a smaller portion of the search space [60]. As explained previously, heuristic-based scheduling algorithms provide feasible solutions, without guaranteeing an optimal schedule, and exhibit a time complexity that is polynomial. The guided random search-based algorithms, on the other hand, use random choices to guide themselves throughout the solution space. These algorithms have a robust performance on a variety of scheduling problems. However, they are less efficient and generate much higher computational costs than heuristic-based algorithms [64].

Heuristic-based scheduling algorithms can be categorised into (i) cluster-based heuristics, (ii) task duplication heuristics and (iii) priority-based (list scheduling) heuristics. Priority-based heuristics are simplistic as they do not consider the inter-process communication. Each task has an assigned priority which is used to assign it to the processor. These algorithms generally provide better performance results at lower computation time [23]. Cluster-based heuristics put related tasks in one cluster, and this cluster is assigned to one processor. In this manner, the communication cost is minimised, although it increases the complexity of the algorithms and makes them more expensive to implement [23]. Task duplication heuristics reduce the interprocess communication overhead by executing the replication of a predecessor task on the processor assigned to the successor task. Task duplication is also used to handle failures and crashes in the system. Duplicate copies of the same task are executed on multiple processors so that in case of a failure, the application still completes its execution [65]. The replication of the tasks, however, increases the complexity of these heuristics and introduces significant overhead at runtime [23].

Different characteristics of the scheduling problem, e.g., scheduling length, deadline of the tasks, and reliability of the system, are considered by researchers while proposing an algorithm. Scheduling techniques that consider both aspects of algorithmic design and architectural design (bi-criteria scheduling mechanism) are typically more effective as compared to other methods. The scheduling algorithm presented in [66] gives priority to the reliability of the model, while its modified version [67] also considers the deadline of the tasks while scheduling the application. Sun et al. [68] proposed a scheduling algorithm based on Breadth-First Scheduling (BFS) known as BFS* that schedules the tasks with precedence constraints in an OpenMP framework to reduce the sequential execution of dependent OpenMP tasks. The performance of the algorithm is highly dependent on the order of the tasks in the ready list [68].

Authors in [61] have proposed a method known as Predict Earliest Finish Time (PEFT) for task scheduling in heterogeneous distributed systems. The proposed algorithm has the same complexity as other state-of-the-art methods. Still, it optimises the schedule length by introducing a look ahead feature and computes an Optimistic Cost Table (OCT). Optimistic cost is denoted as optimistic because the availability of processors is not considered in the computational cost. OCT is used to select the processors and rank the tasks. PEFT in comparison to the Heterogeneous Earliest Finish Time (HEFT) gives better schedulability and makespan. Heuristic-based scheduling algorithms with low complexities provide better schedulability in case of heterogeneous real-time systems. Authors in [69] have compared 20 algorithms and concluded that Heterogeneous Earliest Finish Time (HEFT) gives the best results (robustness and shortest schedule length in safety-critical systems) in case of random graphs. Colin and Chretienne have proposed a scheduling algorithm which gives good results in case of homogeneous processing nodes. The task graphs that are scheduled in this technique are arbitrary in size while short communication time is also considered between the nodes of task graphs [70]. The mentioned algorithms either only concentrate on scheduling the tasks onto processors without any consideration to communication across the network or consider a simplistic model for the communication rather than the multi-hop network used in real-time distributed systems.

According to [71], the communication architecture is essential for an embedded system to exchange or share data in a timely and efficient manner. A communication architecture consists of nodes that are linked via the communication channels that facilitate nodes to send messages to one another. As mentioned by [72], there are two types of scheduling models for communication, one in which communication contention is considered and the other where communication activities are contention-free. In a contention-free communication model, communication networks or processors are fully connected and have the potential to perform concurrent communications. Also, each processor has an independent communication sub-system. However, in a communication model that considers communication conflicts, a message has to wait for a communication source to be available for transmission, which in turn impacts the execution time of the application [72]. Moreover, task scheduling and message scheduling processes are isolated in a communication contention environment, i.e. they are not scheduled concurrently. However, synchronisation of tasks and messages is critical for end-to-end worst-case response time and has an impact on the overall system properties like the ease of maintenance, cost, and performance [73, 12, 72].

One can distinguish two types of communication systems. Systems in which communication activities are triggered dynamically in response to an event are called event-triggered systems. In contrast, the systems in which communication activities are triggered at pre-determined moments in time are called time-triggered systems [74, 12]. Therefore, there are communication protocols where message scheduling is performed dynamically on the occurrence of an event, such as CAN [75], LonWorks [76], and Bytefight [77], and then there are communication protocols that schedule the messages statically based on the progression of time, e.g., Time-Triggered Protocol (TTP) [78], SPIDER [79], SAFEbus [80], TTEthernet [81] and TTCAN [82]. Within time-triggered communication, the messages are transmitted during pre-defined time windows leading to the advantage of a quasi-deterministic behaviour during regular operation. Therefore, time-triggered communication systems provide higher dependability since missing messages are easily detected, and the network is guarded against non-authorized message access [74]. Another interesting property of time-triggered communication systems is composability. Since the time windows for the network access are pre-defined, the behaviour along the timeline is decoupled from the actual network load. Thus, it is possible to develop different sub-systems, simulate the exact time behaviour of said sub-systems and subsequently integrate them into the complete system [74]. One drawback of time-triggered communications is the lack of flexibility and the restrictive design process since all the communication messages and their time specifications must be known in advance for an efficient implementation. Moreover, task and message executions must be synchronised during operation to ensure that the real-time system application fulfils all its strict deadlines and timing constraints [74]. Dynamic scheduling of messages in event-triggered systems gives them the advantage to schedule asynchronous communication activities that are not known in advance. Therefore, in comparison to time-triggered systems, event-triggered systems are more flexible [74]. For safety-critical applications, time-triggered systems are preferred because they ensure the temporal behaviour and strict timing constraints of the application.

A significant effort has gone into developing algorithms that find a reliable, bandwidth-efficient, and deterministic communication schedule for different time-triggered communication networks such as TTP, Ethernet or FlexRay [83]. Steiner et al. [84, 85, 86, 87] used SMT Solver, Tabu Search and Network calculus to compute schedules for time-triggered communication activities for TTEthernet and Automotive Ethernet. In [50], Pop et al. proposed an extensible scheduling method for TTP bus traffic. Their method first finds a solution that satisfies the hard real-time constraints of the system and then uses an iterative algorithm to schedule any new traffic into vacant spaces thus improving the availability of the resources for any future use. Authors in [88] presented an offline schedule synthesis model for time-triggered communication flows to improve reliability in large hybrid (i.e., containing both wireless and wired) networks. Authors in [89] have proposed a scheduling algorithm for time-triggered communication flows for TTEthernet. Their algorithm is based on the path-hop of tasks to generate a rational scheduling timetable that is free of conflicts on the physical links.

FlexRay is a hybrid protocol that allows sharing of the network by both time-triggered and even-triggered messages, thus offering the advantages of both worlds [12]. For this reason, a lot of recent work focuses on scheduling communication activities in the FlexRay protocol. For the static (time-triggered) part of the FlexRay, Lukasiewycz et al. [90] are the pioneers to introduce the method for transforming the basic static segment scheduling problem into a two-dimensional bin packing problem. Their main objective is to minimise the number of allocated slots and to obtain such a schedule where further traffic can be accommodated with no need to allocate new slots. In [91], authors proposed an algorithm to schedule periodic signals in the static segment of FlexRay. Their method first packed the communication signals into message frames while maximising the utilisation. In the second step, the message frames are scheduled while using a minimum number of slots. In [92], authors presented a static segment scheduling problem with real-time constraints. Their method consists of two steps, in the first step the signals are packed into message frames and in the second step a frame scheduling algorithm is used to create a schedule. Kang et al. [93] proposed a frame packing algorithm for the static segment of FlexRay that allows packing of communication signals with different periods into a single frame. In [94], authors proposed a fast heuristic to schedule communication signals on the static segment of FlexRay. In [95], authors proposed an Integer Linear Programming (ILP) algorithm to allocate signals to static slots of FlexRay. Zhao et al. [96] proposed a fast heuristic and an efficient Mixed Integer Linear Programming (MILP) algorithm to optimise the scheduling of the static segment of FlexRay. The authors also proposed a rectangle bin packing optimisation approach to schedule communication signals with timing constraints at minimum bandwidth cost [97]. Authors in [83] proposed a multi-variant scheduling algorithm to schedule the communication signals of new vehicle variants. All of the mentioned approaches only focus on communication schedules and do not attempt to schedule the tasks.

Generally scheduling in distributed real-time systems requires a complete system view that takes into account both processors and network devices as well as the task-level execution times and their inter-dependencies. For an optimal resource utilisation, all these parameters should be considered at once. However, with growing system sizes and complexities, the holistic approach for scheduling becomes more and more difficult. The industrial solution for that is to construct the application and communication schedules separately and synchronise them afterwards [84]. The isolated approach, though, seriously limits the flexibility and performance of real-time applications [98]. There are a handful of studies that have considered the scheduling of both communication signals and application tasks. In [99], authors proposed a MILP algorithm that executes joint scheduling of communication messages and application tasks with the aim to reduce network energy consumption. In [100], another ILP-based algorithm is proposed that schedules both communication messages and application tasks while improving communication latency. Authors in [101] proposed a MILP-based optimisation approach to solve the scheduling problem of time-triggered systems communicating over a FlexRay static segment. Another ILP-based optimisation approach for FlexRay-based time-triggered systems is proposed in [102]. Authors in [98] also proposed a MILP-based optimisation approach to solve the scheduling problem of FlexRay based real-time automotive systems subject to both authentication mechanism constraints and traditional design constraints. The objective of their approach is to improve timing performance and extensibility.

Authors in [103] have proposed an ILP-based method for the scheduling problem and a greedy randomised adaptive search procedure-based heuristic for the routing problem of time-triggered flows and AVB flows in time-sensitive networks. Authors in [104] have designed a scheduling algorithm named as Unfixed Start Time (UST). The proposed algorithm deals with both task and message scheduling problems and uses rescheduling and backtracking methodologies named as Rescheduling with Offset Modification (ROM) and Backtracking and Priority Promotion (BPP) to solve the assignment conflicts. Authors in [105] proposed a genetic algorithm for scheduling applications in Time-Sensitive Networks (TSN). They considered joint routing and scheduling optimisation problem in the context of time-triggered traffic for safety-critical systems. Authors in [106] study the scheduling and communication synthesis problem in integrated avionics for satellites and propose a scheduling model based on the mechanism of a time-trigger bus. Although the mentioned approaches consider both task and communication scheduling, their computation time makes them infeasible for runtime scheduling.

## 3.2 List Scheduling

In distributed and parallel real-time systems, an application is usually represented through a Directed Acyclic Graph (DAG). A DAG is an application model that represents the characteristics of the application tasks and the precedence constraints between them and is a standard form of representing an input to the scheduler. An effective scheduling scheme to schedule the DAG not only minimises the worst-case response time but also maximises task concurrency. Scheduling tasks to minimise the overall schedule length of the DAG falls under the NP-hard optimisation problem. In the state-of-the-art, numerous heuristics are proposed to solve this NP-hard optimization problem [107][108][109]. One of the most efficient heuristics in this category is the list scheduling algorithm [19]. List scheduling forgoes the search for an optimal solution in favour of reducing the time complexity of the scheduling problem. Authors in [110] stated that a list scheduling algorithm solves two problems, i.e., (i). how two tasks with no precedence order can be parallelised, and (ii). how the overall schedule length of the application can be reduced. A list scheduling algorithm consists of two phases. In the first phase, the priorities are assigned to the tasks, and the tasks are ordered into a list in ascending or descending order of the priorities. In the second phase, the algorithm allocates the tasks to processors that give them the earliest start time. The polynomial-time complexity of the list scheduling algorithm allows it to be called during runtime of the application.

Heterogeneous Earliest Finish Time (HEFT) [23] is a list scheduling algorithm that uses a recursive approach in the bottom-up direction to determine the order of the tasks, which is based on the computation costs. The tasks are then processed following their order. HEFT is built on the notion of preferring the critical path tasks, which leads it to depth-first search based ordering of tasks and subsequent execution. Authors in [36] propose an algorithm, where tasks on the critical path are scheduled first, and non-critical path tasks are scheduled according to their calculated priority. Another algorithm, the Critical Path/Most Immediate Successors First (CP/MISF) algorithm by authors in [28] is also based on the bottom level task ordering with ties being broken by giving precedence to the task with the higher number of successors. Authors in [111], propose a scheduling heuristic for heterogeneous systems named Constrained Earliest Finish Time (CEFT). Their heuristic is based on the notion of a constrained critical path (CCP), which is a small task window representing ready tasks in one instance. CEFT finds critical paths in the DAG, and subsequently, the tasks in the CCPs are scheduled using the finish time of the entire CCP.

Authors in [112] analyse various priority schemes that are based on task orders according to the bottom level augmented with metrics based on the communication costs between two tasks and critical path based orders as proposed by authors in [113, 36]. Authors in [114] proposed a locality-aware list scheduling algorithm for homogeneous distributed systems where tasks in the same path are grouped together based on their path length and their level in the DAG. The groups are then scheduled onto free processors. Their algorithm is a work-conserving algorithm that takes into account both locality and load balancing in order to reduce the execution time of the DAG. The Dynamic Critical Path (DCP) scheduling algorithm presented in [115] is based on a critical pattern traversal approach. It attempts to minimise the schedule length at each step by using the remaining critical path. A final schedule is not produced until all the tasks have been processed. The DCP uses the absolute earliest start time and the absolute latest start time that represent the possible execution of a task at the earliest or the latest time, respectively. These values are computed through the breadth-first traversal of the task graph. Another algorithm, called the Modified Critical Path (MCP) is proposed in [34], where the tasks are ordered by their bottom level and for tasks with equal bottom levels, the bottom level of the successor tasks are considered and so on.

All of the mentioned algorithms do not consider periodic tasks and assume that there is no synchronisation between tasks and communication, i.e., they do not consider the communication contention between tasks and only focus on mapping and allocating tasks onto the processors. This assumption is unrealistic for hard real-time, distributed embedded systems where timing constraints need to be guaranteed. There are very few list scheduling algorithms that consider both task and message scheduling. One such example is [116] where the authors proposed a contention-aware list scheduling algorithm but they made the assumption that a message starts transmission at the same time on all links in a path even if the links have different bandwidths. This assumption is again unrealistic for real-time systems where a message cannot start transmitting until it is completely received from the previous link. Moreover, they also do not consider different periods of the tasks and messages while scheduling them. Another list scheduling algorithm that considers joint scheduling of tasks and time-triggered communication activities is proposed in [117] but their main goal is to reduce the overall execution time of the time-triggered communications rather than the whole system application.

## 3.3 Incremental Scheduling

Authors in [50] have presented a scheduling technique for the hard real-time embedded systems while minimising the modification cost of the system. The authors proposed their algorithm, keeping in mind two specific goals. The first goal is that the already running applications are disturbed as little as possible while scheduling the new functionality, i.e., the reconfiguration cost is minimised, and the second goal is that any future functionality can be added in the system with minimum effort. The proposed algorithm adds the new functionality into the vacant spaces of the old schedule such that the mentioned goals are achieved. The algorithm has one drawback that it should be aware of any future functionality being added to the system. Authors in [118] have proposed (i) a high-level scheduling algorithm for System-of-Systems (SoS) and (ii) a low-level scheduling problem for individual constituent systems. The SoS is comprised of constituent systems. Each constituent system is an embedded system, which consists of end systems connected through a real-time network. The incremental scheduling problem is expressed using MILP and implemented in IBM Cplex. Authors in [119] propose an ILP-based algorithm to incrementally add time-triggered flows in the domain of time-sensitive software-defined networks (TSSDN). The TSSDN is a network architecture which provides deterministic real-time guarantees for time-triggered traffic by isolating it either temporally or spatially. The evaluations show that the proposed algorithm can compute incremental schedules for time-triggered flows in a few seconds with an average scheduling ratio of 68%. The first multi-variant scheduling algorithm in the domain of automotive systems is proposed in [120] wherein a first fit based heuristic algorithm is used to create schedules for several vehicle variants at once, where a given signal in all the schedules is transmitted at the same time.

Sagstetter et al. [121] propose an iterative multi-variant schedule in the same domain, where the signals common to all the vehicle variants are scheduled in the first iteration, the signals shared among the variants are scheduled in the intermediate iteration, and the signals specific to just one variant are scheduled in the final iteration. Authors in [83] propose an efficient and robust heuristic algorithm to create schedules for time-triggered internal communication of new vehicle variants. The proposed algorithm provides variant management by ensuring compatibility among the new vehicle variants and preserving backward compatibility with the preceding vehicle variants. The algorithm also uses an extensibility optimisation heuristic that predicts the communication signals of future design iterations and enhances the current schedule, such that it allocates less bandwidth. All of the mentioned incremental scheduling algorithms either do not consider the already scheduled tasks and messages while making the changes or just consider the schedule of the time-triggered communication flows or require knowledge about any future changes in the system to create an optimal schedule. Moreover, the last three mentioned algorithms are applicable only to the domain of automotive systems where management of multiple variants of a vehicle is taken as a test case. Moreover, the mentioned algorithms are proposed for offline scheduling and are not efficient enough to be invoked during runtime.

## 3.4 Fault Detection and Diagnosis in Embedded Systems

Proactive maintenance is ensured by detecting faults through online monitoring schemes that help in assessing the credibility of the operations, performed by the system. The monitored output is then compared with the standard behaviour of the system to ensure that the system is working as expected [122]. As mentioned by [123], recent technological advancements have made fault detection and diagnosis challenging as it requires a deeper understanding of the system. Early fault detection is crucial in avoiding degradation of the embedded system, degradation of the product, and overall damage to the system. Correct fault detection and diagnosis also facilitate optimal and proper corrective decisions, associated with repairs and required remedial actions. Traditional fault detection and diagnosis approaches include checking variables or physical redundancy, whereas more complex methods also include behavioural and spectral analysis of signals [124]. Fault detection and diagnosis in embedded systems involve different tasks such as fault identification, fault isolation, and fault detection. Fault detection, according to [124] is an indication of fault possibility. Fault isolation, on the other hand, discovers the fault location while fault identification determines the fault magnitude. The simplest approaches to detecting and diagnosing faults are associated with making comparisons between pre-set limits and the system's output [125]. The output of the system is compared to the output obtained from the model using mathematical models, and any discrepancy means the existence of a fault. [126] asserted that fault detection and diagnosis is a data processing system which is based on information redundancy, in which data as well as the understanding of humans about the data, serve as the two basic elements. The study further claimed that fault detection and diagnosis is done via intelligent computation, signal processing, and mathematical modelling. The arrival of efficient information techniques, communication networks, and computerised control have made fault detection and diagnosis simple, efficient, and effective.

To ensure the safety and reliability of embedded systems and minimise breakdown risks, fault detection and diagnosis have been gaining significant attention in the automation and control community. It focuses on detecting a failure while identifying the fault's location and magnitude as precisely as possible [126]. Different analytical models, such as state-space and signal-based models, have significantly improved the techniques of diagnosing and detecting the faults. Authors in [127] state that diagnosis and detection of faults is often used to continually monitor a system during the execution of a task, which is also known as online monitoring to assure that the system is giving optimal performance. [127] further contends that fault detection and diagnosis can be categorised into model-free and model-based methods. The former can be categorised into univariate – signal-based, and multivariate – data-driven methods.

Model-based methods make use of mathematical models to understand the system's behaviour [127]. In this way, the system's faults are detected by observing the consistency between the predicted and observed behaviour via mathematical models. However, since it requires an accurate model to predict the behaviour, practical applications are limited. Data-driven methods, on the other hand, uses data history to mine implicit knowledge via machine learning or intelligent training methods [126]. The model, obtained from this method is then used to approximate actual values of the new measurements; thus, the evaluation of approximate residuals leads to fault detection. Signal-based fault detection and diagnosis methods make decisions through the comparison of feature spectrums [127]. These features spectrums are established for a signal with appropriate values as a baseline. Fault detection and diagnosis in embedded systems can contribute to mitigating faulty events and result in ensuring safety while eliminating the risk of any potential damage to the system. The mentioned fault detection and diagnosis techniques do not provide runtime scheduling mechanisms that are essential requirements for time-triggered ODRE systems. Moreover, the mentioned techniques do not consider the stringent timing constraints and the dynamic nature of the ODRE systems together. Therefore, there is a need for a scheduling algorithm that computes a feasible schedule that follows all the stringent timing constraints of the diagnostic application, and that also considers the dynamic nature of the ODRE systems.

# CHAPTER 4
# SYSTEM MODEL

THIS CHAPTER presents the system model used for the submitted work. The first part of this chapter illustrates the architecture of the system. The second part presents the abstract representation used to model the system application, and the last part of this chapter explains the diagnostic services provided by the system.

## 4.1 System Architecture

Architectures consisting of distributed nodes with a dedicated communication network are considered. Such a distributed system denoted with *S* contains a set of processors *P* and switches *SW* connected through bi-directional links *L*.

$$S =< P, SW, L > \tag{4.1}$$

An example of such a distributed system is given in Fig. 4.1.

### 4.1.1 Hardware Components of Processors

At the hardware level, each processor in the has the following components,

- A *communication controller* that controls the transmission and reception of time-triggered messages.
- An *input/output interface* to sensors and actuators.
- *ROM* to store the code of the kernel and *RAM* to store the local data and code of the task, e.g. diagnostic query, assigned to the processor.

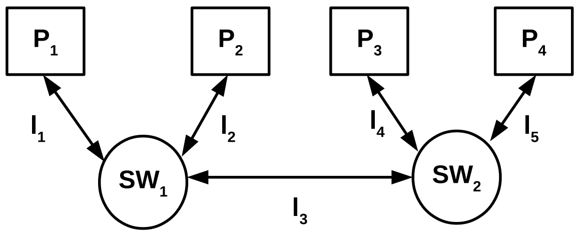


Figure 4.1: Example of a distributed system

### 4.1.2 Software Components of Processors

At the software level, each processor $P_i \in P$ in $S$ has the following components,

- A *local real-time database* that stores the data required for the execution of the query assigned to the processor. Once the query has completed its execution, its corresponding data is deleted from the *local database* provided that no other future query assigned to the processor requires it. This deletion is done to ensure that the data replication does not cause a memory overflow of the processor.
- A *query execution engine* that executes the queries assigned to the processor.
- A *real-time (RT) kernel* that stores the local *schedule table* containing all the information needed to make decisions on the activation of tasks such as diagnostic queries and the transmission of messages at predetermined moments in time.

The processors and switches in the system are connected through a real-time communication network. Although there are different types of communication models, this work only considers static time-triggered communication because it provides temporal prediction, guaranteed data transference, easier certification and fault isolation through guardians that prevent incorrect transmission of messages. Furthermore, time-triggered communication provides implicit synchronization of the database since each application task accesses the local database during its allocated time, so there is no need for locking the database or rollbacks. The communication network can use any topology that supports time-triggered communication.

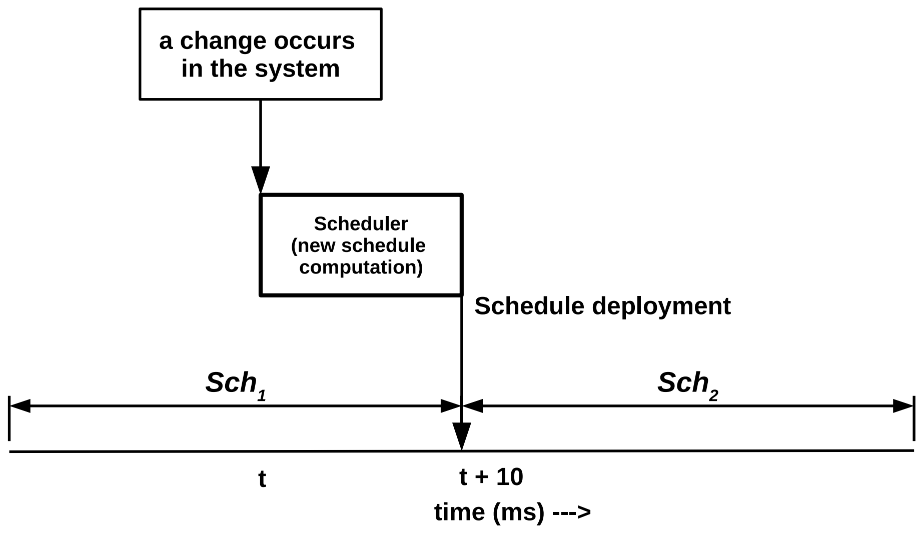

a change occurs in the system
Scheduler (new schedule computation)
Schedule deployment
Sch1
Sch2
t
t + 10
time (ms) --->

Figure 4.2: Transition between schedules during runtime whenever changes occur in the system

## 4.1.3 Scheduler

A node in the overall system is reserved solely to compute the schedule. It is termed as *scheduler* and its *RT-kernel* stores the global schedule of the whole system. Whenever there are changes in the system architecture or the system application, the scheduler is invoked to recompute the new schedule at run time. Until the new schedule is computed, the system keeps running the old schedule, for example, in Fig. 4.2, a change occurs in the system at *t* ms and the scheduler is invoked to recompute the schedule corresponding this change. At *t + 10* ms, the scheduler successfully computes the new schedule $Sch_2$ and deploys it to the other nodes in the system. Before *t + 10* ms, the system keeps running the old schedule $Sch_1$. The time between the occurrence of the change and the computation of the new schedule needs to be bounded because the schedule is computed at runtime, and it needs to be deployed before the system becomes unstable. Therefore, an efficient and fast scheduling heuristic is required to compute the new schedule within the bounded time. If the scheduler is unable to compute a schedule, then the designer is notified that the changes are not integrable with the running system and must be redefined. In this case, the system discards the changes and keeps running the old schedule $Sch_1$.

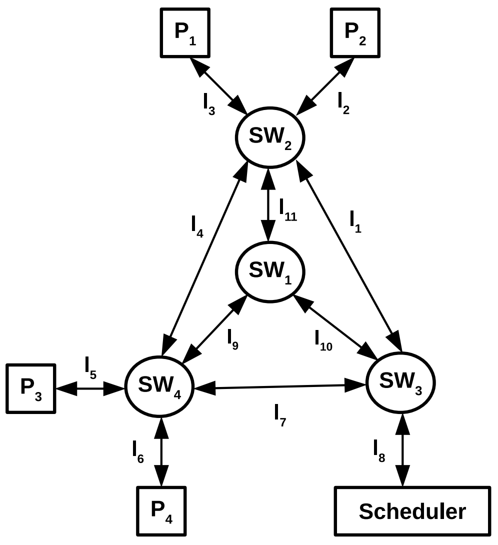


Figure 4.3: System Architecture (Mesh Network)

### 4.1.4 Constraints of the Distributed System

An example of the overall system architecture connected through a mesh network is given in Fig. 4.3. The overall system has the following constraints:

- The processors are only used for the execution of the tasks or queries, whereas the bi-directional links are only used for time-triggered communication.
- The schedule is non-preemptive, i.e. once a task (query) is assigned to a processor, the task frees the processor only when it is completed. Similarly, the messages are transmitted over a link during their predetermined time intervals and cannot be preempted.
- If the tasks (queries) are periodic, then all iterations are executed on the same processor. Similarly, all iterations of a periodic message are transmitted through the same links.
- The scheduler computes the schedule for both the application and the diagnostic services.

- For simplicity sake, it is assumed that the system runs both the application and the diagnostic services in parallel to each other without any conflicts.

- Each link in the system has a fixed rate of transmission, and the communication cost is calculated using,

$$c(m_{ij}, l_k) = \frac{d(m_{ij})}{w_k} \tag{4.2}$$

  where $c(m_{ij}, l_k)$ is the communication cost of message $m_{ij}$ when it transmits data $d(m_{ij})$ over a link $l_k$ of bandwidth $w_k$.

- The processors in the distributed system can either be homogeneous or heterogeneous e.g., it is not necessary that all the processors in the system have the same clock frequency.

- Before the query starts executing its required data is replicated from the other processors to the local database of the executing processor through peer-to-peer communication. Once a query has completed its execution, the data required for its execution is deleted from the local database.

- Any modification can occur in the system architecture e.g. removal of a switch as shown in Fig. 4.4.

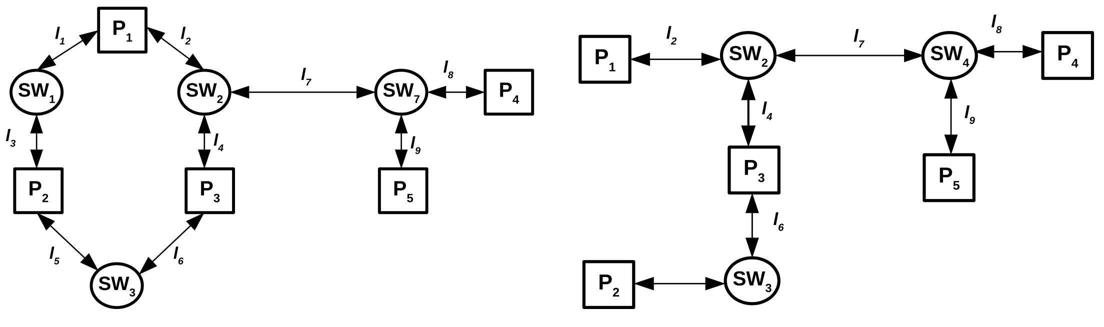


Figure 4.4: (a). System Architecture before Modification (b). System Architecture after Modification

## 4.2 Application Model

There are two types of services provided by the system: the application that implements system functionality and the application that provides the service of diagnosing and detecting faults

### 4.2.1 System Application (G$_S$)

The system application is represented through a Directed Acyclic Graph (DAG). In a DAG, each vertex represents a sequential code segment or a task, and each edge represents the relationship between the tasks. Such a task graph $G_A$ can be represented as,

$$G_S = < T, E > \tag{4.3}$$

where $T$ is a finite set of $|T|$ tasks and $E$ is a finite set of $|E|$ directed edges. Each task $t_i \in T$ represents a non-divisible sequential task and each edge $e_{ij} \in E$ represents the precedence constraint between tasks $t_i$ and $t_j$ where $t_i$ is the parent task of $t_j$. The weight $w_i$ assigned to the task $t_i$ represents its computation cost or execution time whereas the weight $d_{ij}$ assigned to the edge $e_{ij}$ is the amount of data transferred between tasks $t_i$ and $t_j$. Each task $t_i \in T$ has a hard deadline $D_i$ that must be fulfilled to keep the system from failing. The overall application is cyclic with a constant period $P$.

A few examples of the application graphs are shown in Fig. 4.5. There is no restriction on the input, and the task graphs can have an arbitrary shape with an arbitrary number of tasks and edges. Any modification can occur in the system. If a new application is added to the system, then two dummy tasks called *source*, and *sink* are added such that tasks in both the original and the new application graph are successors of *source* and predecessors of *sink*. These dummy tasks are added to make a connection between the two graphs and have no impact on the schedule of the application. Fig. 4.6 shows the modified version of the application graphs given in Fig. 4.5 if they are executed on the same system.

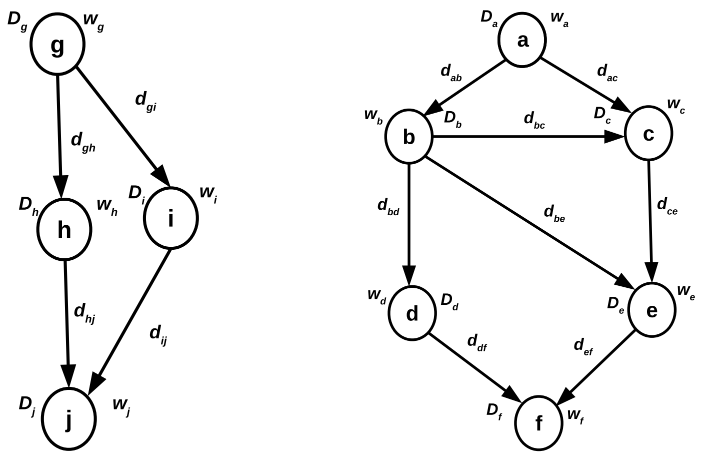


Figure 4.5: Examples of Directed Acyclic Graph (DAG)

### 4.2.2 Diagnostic Multi-Query Graph (DMG - $G_D$)

The Fault Detection and Diagnosis (FDD) platform used in this work is based on diagnostic queries that use rule-based inference and semantic web technology to identify faults in the system. In such an approach, a Diagnostic Knowledge Base (DKB) describes the structure of the ODRE systems using semantic web technology, i.e. constituting components and their interfaces, defines faulty or abnormal behaviour, rules for the identification of faults and the respective recovery actions for mitigating failures. A directed graph of diagnostic rules herein called Diagnostic Multi-query Graph (DMG) is the central element of this technique. Diagnostic features derived through local error detection mechanisms, e.g., message classification, self-testing, serve as the starting point for the inference on faults and recovery actions. The diagnostic inference process is temporally and spatially decomposed by introducing intermediate inference steps called symptoms. These symptoms and diagnostic features are stored in a real-time database, parts of which are timely and consistently replicated to enable the distributed execution of rules. Each rule is realized as a query on the diagnostic facts within the real-time database. The vertices in the DMG represent queries, and the edges depict data relationships through the real-time database between these queries [6]. The management unit of DKB maintains the diagnostic knowl-

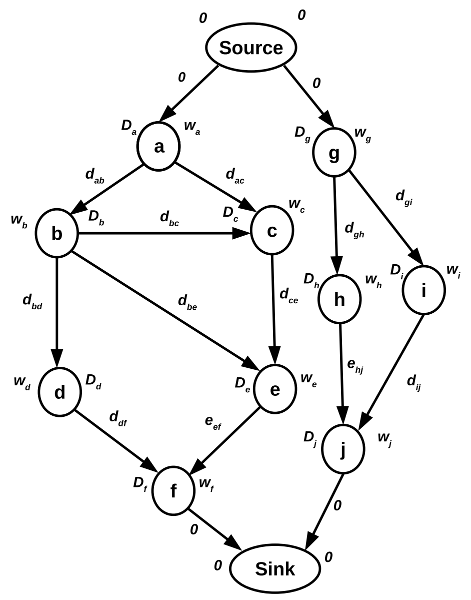
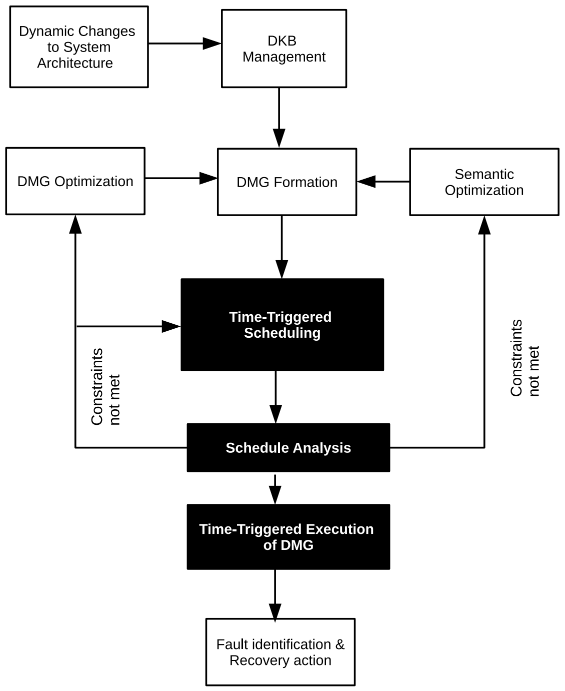


Figure 4.6: Modified version of the graphs presented in Fig. 4.5

edge, integrates new knowledge and ensures consistency, completeness and integrity. The faults considered in this system can either be permanent hardware faults, software faults or interaction faults. A time-triggered schedule for the execution of the diagnostic queries is computed to ensure that the system meets the stringent time constraints. It does not only specify the time and location for query execution but also which data is replicated at which component to enable the distributed implementation of the queries under given resource constraints. The execution of the DMG, according to the computed schedule, identifies faults. Once the fault is detected, the system proposes appropriate recovery actions. For example, possible reactions to permanent faults include introducing degraded service modes as a first-level recovery. Whereas less time-critical second-level recovery may include extended reconfiguration measures with knowledge base support. Fig. 4.7 shows the flow diagram of Fault detection and diagnosis using Diagnostic Queries (FDQ). The marked blocks are covered in this thesis.

In a DMG, each query is unique and is either linked with sensors or a set of queries. The sensors periodically provide data, which is replicated to the databases of other processors through peer-to-peer communication based on the schedule computed by the scheduler.



Dynamic Changes to System Architecture
DKB Management
DMG Optimization
DMG Formation
Semantic Optimization
Time-Triggered Scheduling
Constraints not met
Constraints not met
Schedule Analysis
Time-Triggered Execution of DMG
Fault identification & Recovery action

Figure 4.7: Flow diagram of Fault detection and diagnosis using Diagnostic Queries (FDQ)

Each vertex in the graph represents a query, and each edge specifies the input/output relationship between two queries through the real-time database. The queries are categorized as features, symptoms and faults. A feature is an output from a sensor, a symptom is an abnormality in said output, and a fault is the failure of the sensor. For example, in a car, an oxygen sensor is used to monitor the percentage of oxygen in the exhaust. The output of this sensor is a feature. The increasing or decreasing of the output beyond the threshold level of the sensor is a symptom. If the output of the oxygen sensor does not fall within the required threshold range, then the sensor is not functioning correctly and is faulty. The vertices without incoming edges are features and the ones without outgoing edges are faults. The vertices that have both incoming and outgoing edges are termed symptoms. Since the sensors are sending data periodically, so the queries also need to be regularly executed respecting predefined phase-shift from the sensor input to consider the computational and communication delays. Therefore, each query is linked with a strict time period and a strict phase respecting a global time base. The periodic executions of queries create the possibility that the child query requires output data from one or a group of the previous executions rather than the present one. This feature is incorporated in the DMG by labelling the edges with timing information called a history-interval. The history-interval defines an interval from which query results are required for the execution of the target query. The start of this interval determines what output of the parent query is discarded from the local database of the source processor, and the end interval determines the permissible communication delay between the executions of the parent and child queries. The history-interval is explained in detail later in this section.

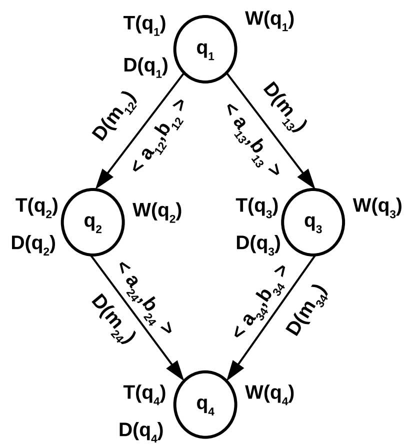
A DMG can be represented as,

$$G_D =< Q,M > \tag{4.4}$$

where $Q$ is a finite set of $|Q|$ queries and $M$ is a finite set of $|M|$ directed edges. Each vertex $q_i \in Q$ represents a non-divisible periodic query task and each edge $m_{ij} \in M$ represents the data dependency between query tasks $q_i$ and $q_j$ where the transference of data is from $q_i$ to $q_j$. Each $q_i \in Q$ is represented by the tuple $< W(q_i), D(q_i), T(q_i) >$ where $W(q_i)$ is the worst-case execution time (WCET) of $q_i$, $D(q_i)$ is the relative deadline of $q_i$ and $T(q_i)$ is the time period of $q_i$. Each periodic instance of $q_i$ has the same $W(q_i)$ and $D(q_i)$. Each directed edge $m_{ij} \in M$ is represented by the tuple $< D(m_{ij}), < a_{ij}, b_{ij} >>$ where $D(m_{ij})$ is the amount of output tuples transferred from $q_i$ to $q_j$ and $< a_{ij}, b_{ij} >$ is the history-interval of the edge $m_{ij}$. Any change can occur in the structure of the DMG e.g. generation of new queries when new sensors are added to the system. Two example DMGs are given in Fig. 4.8.

## 4.2.3 Characteristics of DMG

The DMG has the following characteristics that aid in the scheduling process.



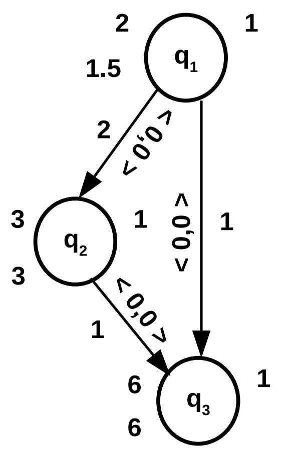


Figure 4.8: Examples of DMG

### *4.2.3 Time Period and Absolute Deadline*

The sensors in the system are sending data periodically. Therefore each query in the DMG also executes periodically to make sure that the query is executed on an up-to-date input data from the sensors. In a DMG represented by Eq. 4.4, each query $q_i$ repeats execution after a strict time interval $T(q_i)$. This time interval $T(q_i) \in \mathbb{R}_{>0}$, herein is termed as time period and is the exact time difference between release times of two consecutive instances of $q_i$ [21]. The release time of $q_i$ is the earliest time at which $q_i$ is available for execution. Let $rt_{k+1}(q_i) \in \mathbb{R}_{\geq 0}$ and $rt_k(q_i) \in \mathbb{R}_{\geq 0}$ be release times of two consecutive instances of query $q_i$ then the time period of $q_i$ can be represented as,

$$T(q_i) = rt_{k+1}(q_i) - rt_k(q_i) \tag{4.5}$$

Since it is possible that the query does not start execution right after being released and its actual start time may differ from its release time, therefore it is necessary that each instance of $q_i$ completes its execution before the start of its next instance to hold the strict time period. If $st_k(q_i) \in \mathbb{R}_{\geq 0}$ is the actual start time of the *kth* instance of $q_i$ and $W(q_i) \in \mathbb{R}_{>0}$ is its WCET then this condition can be represented as,

$$st_k(q_i) + W(q_i) < rt_k(q_i) + T(q_i) \tag{4.6}$$

For this condition to hold it is necessary that the WCET of $q_i$ is always less than its $T(q_i)$, i.e. $\frac{W(q_i)}{T(q_i)} < 1$ [21]. Time-period of a query is sometimes also considered as its deadline, in that case a previous iteration of the query should complete its execution before its calculated time period is reached. For example, two instances of a query $q_1$ with a time period *10* ms and a WCET *3* ms are shown in Fig. 4.9. The release time of the first instance of $q_1$ is *t* ms whereas the actual start time of the instance is *t + 1* ms. For $q_1$ to hold its strict time period (deadline) of *10* ms, it is essential that *t + 4 < t + 10.*

Each query $q_i$ in the DMG is annotated with a relative deadline $D(q_i)$. This deadline is relative to the release time of previous instance of $q_i$ and is always less than or equal to the time period of $q_i$. Absolute deadline of an instance of query $q_i$ is the time at which the execution of the instance must be completed. Each instance of $q_i$ has a different absolute deadline and is the sum of the release time of the instance and the relative deadline of $q_i$. Let $d_k(q_i) \in \mathbb{R}_{>0}$ be the absolute deadline of the *kth* instance of $q_i$ then it can be represented as,

$$d_k(q_i) = rt_k(q_i) + D(q_i) \tag{4.7}$$

If the *kth* instance of the query $q_i$ does not finish its execution before its absolute deadline $d_k(q_i)$ then the system becomes unstable. Therefore it is necessary that the sum of the start time of an instance of $q_i$ and its WCET $W(q_i)$ is always less than the absolute deadline of the instance. This can be represented in mathematical form as follows,

$$st_k(q_i) + W(q_i) < d_k(q_i) \tag{4.8}$$

In the example given in Fig. 4.9, $q_1$ has a relative deadline of *5* ms. The first instance of $q_1$

has an absolute deadline of *t + 5* ms whereas the second instance has an absolute deadline of *t + 15* ms. In this thesis, it is assumed that the difference between the start-times of two consecutive iterations of a query is equal to the time-period of the query to ensure that all the iterations meet their respective deadlines.

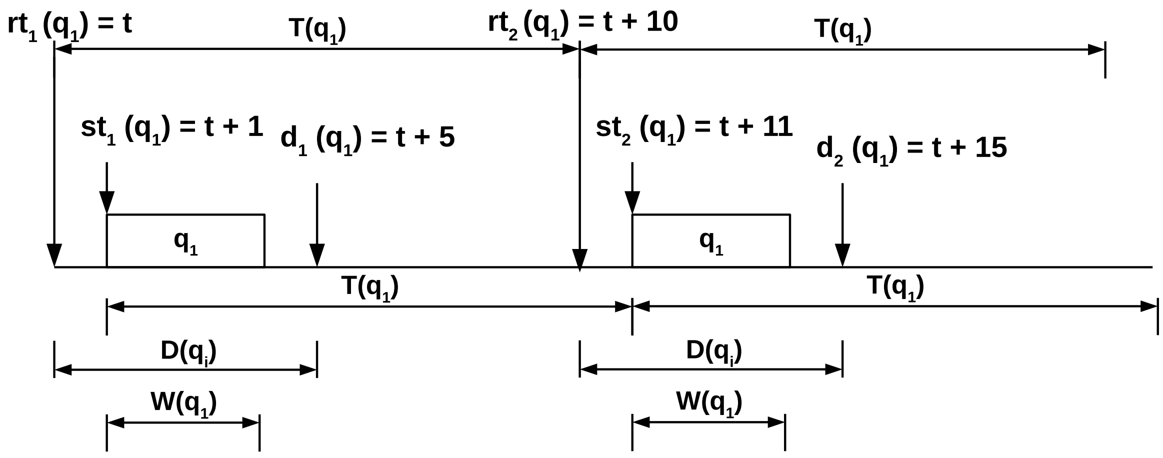


Figure 4.9: Time period and absolute deadline of two consecutive instances of $q_1$

### 4.2.3 Hyper-Period

For a DMG represented by Eq. 4.4, the hyper-period $H_G \in \mathbb{R}_{>0}$ is the minimum time interval after which $G_D$ starts its next cycle of execution [21]. It is calculated by taking the least common multiple (L.C.M.) of all the time periods in $G_D$.

$$H_G = L.C.M.\{T(q_1), T(q_2), ..., T(q_q)\} \tag{4.9}$$

The hyper-period is an important characteristic of $G_D$ and is used to calculate the minimum number of times each query is repeated within one complete execution [128]. Let *times($q_i$)* $\in \mathbb{Z}_{>0}$ represent the number of times a query $q_i$ repeats in one hyper-period $H_G$ of the DMG. It can be calculated as,

$$times(q_i) = f(H_G, T(q_i)) = \frac{H_G}{T(q_i)} \tag{4.10}$$

### *4.2.3 History-Interval*

Each edge $m_{ij} \in M$ is labelled with a history-interval $< a_{ij}, b_{ij} >$. As described before, the start of history interval determines what output data of a parent query is discarded from the local database of the source processor, and the end of the interval determines the permissible communication delay between the parent and child query. The history-interval determines what data must be discarded from the database that bounds the total data consumption. It also depicts the real-time requirements by providing the maximum delay between executions of parent and child queries. History-interval is essential to determine the amount of data required by $q_j$ and the earliest starting time of $q_j$. In simple words, the history interval determines the set of iterations of the parent query required by the child query. Here $a_{ij}$ represents the iteration of parent query from which the data transmission starts and $b_{ij}$ represents the last iteration of parent query that needs to send data to the child query. It has to be noted that $a_{ij}$ does not essentially need to be from the same hyper-period.

For better understanding of history interval, consider a two vertex DMG where $q_1$ is the parent of $q_2$ i.e. $q_1$ is transmitting data, $D(m_{12})$ = 1 Mb, to $q_2$. The time period of $q_1$ is $T(q_1)$ = 2 ms, and time period of $q_2$ is $T(q_2)$ = 5 ms. The queries are scheduled on two homogeneous processors, $P_1$ and $P_2$. The WCET of both queries is 1 ms. There is only one link $l$ ($B_l$ = 1 Mb/ms) between $P_1$ and $P_2$. Query $q_1$ is scheduled on processor $P_1$ while $q_2$ is scheduled on processor $P_2$. According to Eq. 4.9, the hyper-period of the DMG is 10 ms where $q_1$ and $q_2$ repeat five-fold and twice respectively (Eq. 4.10). To ensure that the child query $q_2$ receives all the data sent by the parent $q_1$ without it being lost or duplicated, it is essential to calculate the number of times $q_1$ should be repeated for the complete transference of data to $q_2$. Let $times_{ij} \in \mathbb{Z}_{>0}$ represent the iteration number of a parent query $q_i$ that transmits data to a child query $q_j$. It can be represented as a piece-wise function of the time-periods of the parent and child queries, as shown in Eq. 4.11. In the illustrated example, the parent query has a time period of 2 ms, and the child query has a time period of 5 ms. Therefore according to Eq. 4.11, $times_{12} = f(2,5) = \lfloor \frac{5}{2} \rfloor = \lfloor 2.5 \rfloor = 2$.

The resultant number implies that $q_1$ must repeat twice before $q_2$ can start its execution for the complete transference of data.

$$times_{ij} = f(T(q_i), T(q_j)) = \begin{cases} \left\lfloor \dfrac{T(q_j)}{T(q_i)} \right\rfloor, & T(q_j) \geq T(q_i) \\ 1, & otherwise \end{cases} \tag{4.11}$$

Since there are multiple iterations of parent query $q_1$, therefore it is a possibility that instead of every second iteration, the child query $q_2$ requires data from every first iteration or both iterations of $q_1$. In order to determine the iterations of $q_1$ that must send data to $q_2$, the concept of history-interval is introduced. The history-interval serves as a sliding window to determine what iterations of the parent query must forward their data to the child query. In the history-interval $< a_{ij}, b_{ij} >$ assigned to an edge $m_{ij}$ from a query $q_i$ to a query $q_j$, $a_{ij}$ serves as the starting point of the sliding window whereas $b_{ij}$ serves as the ending point. In order to determine what iterations of $q_i$ must transfer data to $q_j$, the history-interval needs to be translated to iteration numbers i.e., first iteration or second iteration, of parent query $q_i$. Let $I_1 \in \mathbb{Z}$ represent the iteration number of $q_i$ that starts data transmission to $q_j$. It can be represented as a function of $times_{ij}$ (cf. Eq. 4.11) and the start of the history-interval $a_{ij}$ as shown in Eq. 4.12. Let $I_2 \in \mathbb{Z}$ represent the last iteration number of $q_i$ that sends the data to $q_j$. It can be represented as a piece-wise function of $times_{ij}$ (cf. Eq. 4.11) and end of the history-interval $b_{ij}$ as shown in Eq. 4.13. $I_1$ and $I_2$ collectively form a range of iterations that must send data to the child query. This range is represented by the closed interval range $[I_1, I_2] \in \mathbb{Z}$. Consider the example given before where $times_{12} = 2$, now if history-interval is $< a_{12}, b_{12} >=< 1, 0 >$ then according to Eq. 4.12 and Eq. 4.13, the closed interval range is $[1,2]$. It means that the output data of the first and second iterations of $q_1$ must be transmitted to $q_2$. Since the complete data that must be transferred to $q_2$ is available only after the second execution of $q_1$ therefore the transmission starts after said execution. It cannot start before that because then $q_2$ will not receive the complete data required to start its execution. This has been depicted in Fig. 4.10b where

$q_2$ starts its execution after the data from both first and second iterations of $q_1$ has been transmitted. In comparison, in Fig. 4.10a where the history-interval is $< 0,0 >$, the data is transmitted to $q_2$ only from the second iteration of $q_1$. For the correct interpretation of history-interval, it is essential that $b_{ij}$ is never greater than $a_{ij}$. Both $a_{ij}$ and $b_{ij}$ should be non-negative integers i.e. $a_{ij} \in \mathbb{Z}_{\geq 0}$ and $b_{ij} \in \mathbb{Z}_{\geq 0}$.

$$I_1 = f(times_{ij}, a_{ij}) = times_{ij} - a_{ij}, \quad a_{ij} \in \mathbb{Z}_{\geq 0} \tag{4.12}$$

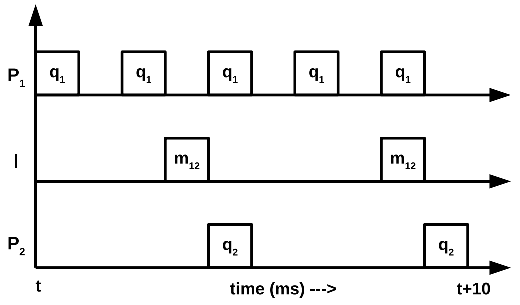
$$I_2 = f(times_{ij}, b_{ij}) = \begin{cases} times_{ij} - b_{ij}, & b_{ij} \in \mathbb{Z}_{b_{ij} \geq 0 \cup b_{ij} \leq a_{ij}} \\ times_{ij}, & otherwise \end{cases} \tag{4.13}$$

If $I_1$ is negative, it means that the required iteration is from a previous hyper-period. Consider the example given before with a history-interval $< a_{12}, b_{12} >=< 3, 1 >$. Now according to Eq. 4.12 and Eq. 4.13, the closed interval range for the required iterations of $q_1$ is $[-1, 1]$. The range values -1 and 0 imply that the data is also transferred from the last two iterations of the previous hyper-period of the DMG. In this case, it is assumed that the required data is available to $q_2$ when the DMG starts execution of its current hyper-period. It has been shown in Fig. 4.10c where the data transference starts after the first iteration of $q_1$ ($I_2 = 1$) but also includes the data from the last and second to last iteration of $q_1$ in the previous hyper-period.

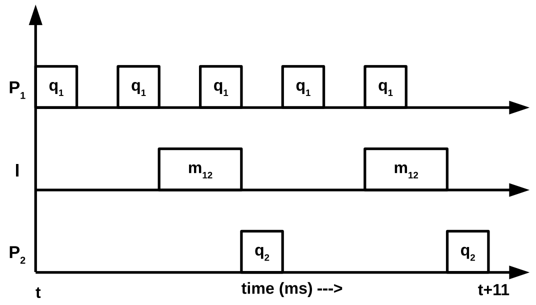
It is important to note that each child iteration gets data from a different parent iteration in the cyclic execution of the graph. The history-interval acts as a sliding window whose width can be given as a function of $times_{ij}$ and time-period of parent query $q_i$. Let $width \in \mathbb{R}_{>0}$ represent the duration of the sliding window of the history-interval, then it can be represented as,

$$width = f(times_{ij}, T(q_i)) = times_{ij} \cdot T(q_i) \tag{4.14}$$

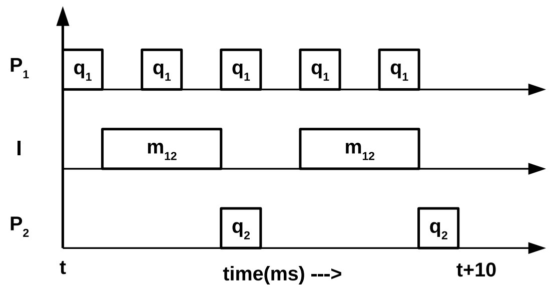
For example, consider Fig. 4.11 where the time-periods for $q_1$ and $q_2$ are 2 and 5 and the



(a) $< a_{12}, b_{12} >=< 0, 0 >$



(b) $< a_{12}, b_{12} >=< 1, 0 >$

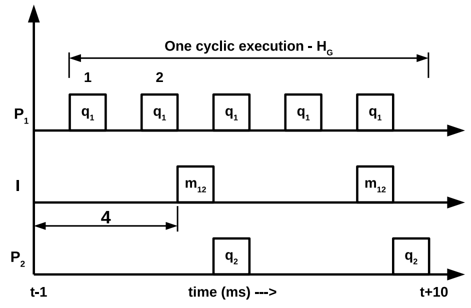


(c) $< a_{12}, b_{12} >=< 3, 1 >$

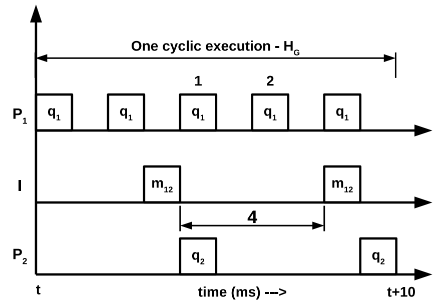
Figure 4.10: Impact of history-interval on the relationship between iterations of $q_1$ and $q_2$ (t = 0 ms)

history-interval is $< 0, 0 >$. In this case, the width of the history-interval according to Eq. 4.14 is $width = \left\lfloor \frac{5}{2} \right\rfloor \cdot 2 = 4$. According to Eq. 4.12 and Eq. 4.13, the data is transmitted after every second iteration of $q_1$. In Fig. 4.11a, the second iteration of $q_i$ with respect to the first iteration of $q_2$ and the calculated width is the second iteration of $q_1$ in one cyclic execution of the DMG. Whereas, the second iteration of $q_1$ with respect to the second iteration of $q_2$ is actually the fourth iteration in one cyclic execution of the DMG (cf. Fig. 4.11b).

The ready time of a child query has an equal impact on the choice of the parent iteration that transmits the data to it, specifically when a query has multiple parents. The query ready time is defined as the earliest time when all the required iterations of a parent query are completed. For example, consider the DMG given in Fig. 4.8b. It is scheduled on a two-processor homogeneous distributed system where the processors are connected by a single link. In the first case, the bandwidth of the link is taken as 1 Mb/ms, and in the second case, the bandwidth is 2 Mb/ms. The corresponding schedules for both cases are given in Fig.

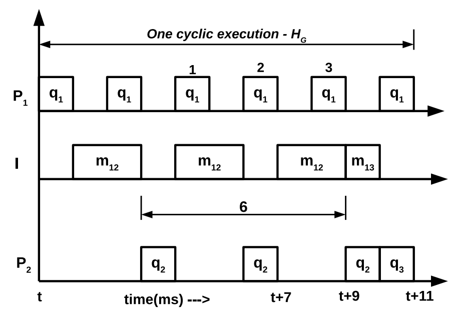


(a)

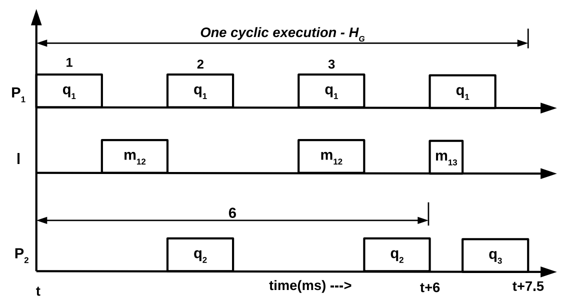


(b)

Figure 4.11: Sliding window function of history-interval ($< a_{ij}, b_{ij} > = < 0, 0 >$ and t = 0 ms)



(a) $w_l$ = 1 Mb/ms



(b) $w_l$ = 2 Mb/ms

Figure 4.12: Impact of ready time of a query on the data transmission (t = 0 ms)

4.12. In both cases, $q_3$ is ready for execution after $q_1$ and $q_2$ have completed three and two iterations, respectively (cf. Eq. 4.12 and Eq. 4.13). In Fig. 4.12a, $q_3$ is ready at *7 ms*, but the data from $q_1$ to $q_3$ cannot be transmitted yet since the link is not free. Therefore, $m_{13}$ is scheduled at the next free time slot i.e. *9 ms*. At *9 ms*, the third iteration of $q_1$ according to the sliding window of width 6 ms is the fifth iteration of $q_1$ in one cyclic execution of $G_D$. In Fig. 4.12b, $q_3$ is ready after *6 ms* therefore, the third iteration of $q_1$ in one cyclic execution of $G_D$ transmits its data to $q_3$.

### 4.2.4 Translation of History-Interval to Directed Edges

In literature, the DAG described in Section 4.2.1 is the standard form of representing an input to a scheduler. In a DAG, each edge represents a single relationship between a parent and child task. The weight of the edge represents the amount of data transferred between two tasks and also the maximum communication delay between their executions. On the other hand, in a DMG, an edge is represented through a history-interval that raises the possibility of multiple relationships between two dependent queries. For example, if the history-interval between two queries $q_i$ and $q_j$, where the edge is from $q_i$ to $q_j$ and the time periods are 4 and 8 ms, is $< 1,0 >$, then according to Eq. 4.12 and Eq. 4.13 the first iteration of $q_j$ must get its data from the first two iterations of $q_i$. Here, there should be two separate directed edges from $q_i$ to $q_j$ for the scheduler to determine the precedence constraint between the queries and also to compute the correct amount of data transferred from $q_i$ to $q_j$. Since the scheduler is not able to directly convert the history-interval to the corresponding directed edges, therefore, the history-interval between $q_i$ and $q_j$ needs to be translated before giving the DMG as an input to the scheduler.

By using the characteristics mentioned above, a DMG can be simplified to make the graph easier to schedule. The goal is to map the history-interval into appropriate edges so the scheduler can determine the correct precedence constraints between the queries and also the amount of data transferred. To accomplish this, firstly, the hyper-period of the DMG is calculated using Eq. 4.9. Then the minimum number of iterations for each query task in the DMG in one hyper-period is calculated using Eq. 4.10. Each query in the DMG is replicated the number of times it needs to be repeated in one hyper-period. Each replication represents an instance of the query. All the instances of a query are stored in a set that is enclosed with a specific tag. Each instance in the set is scheduled on the same processor and the time difference between two consecutive instances is equivalent to the time period of the set. All the instances in a set have the same relative deadline and WCET. The history-intervals given in the DMG are used to calculate data dependencies between different iterations of the query tasks using Eq. 4.12 and Eq. 4.13. These data dependencies are stored in the form of a $m\ x\ n$ query relationship matrix $RM$. This matrix represents the relationship between a parent query $q_i$ and the first iteration of the child query $q_j$ across multiple executions of the $q_i$ and $q_j$ DMG. The $m$ rows in this matrix represent the total number of cyclic executions of the DMG (comprising only of $q_i$ and $q_j$) from which the first iteration of $q_j$ requires data in the current cyclic execution. Since one cyclic execution of the DMG is equal to one hyper-period therefore *m* just represents multiple hyper-periods of the DMG from which the data is required by $q_j$. The *n* columns in the matrix represent the total number of iterations of $q_i$ in one hyper-period of the DMG. Each $rm_{ab}$, where *a* = *1,2,...,m* and *b* = *1,2,...,n*, gives the relationship between the first iteration of $q_j$ in the current hyper-period and the *bth* iteration of $q_i$ from *ath* hyper-period of the $< q_i, q_j >$ DMG. If $rm_{ab} = X$, then it means that *bth* iteration of $q_i$ from the *ath* hyper-period of the the DMG is transmitting data to the first iteration of $q_j$. The total number of rows *m* and the total number of columns *n* are calculated as follows,

$$m = \begin{cases} \left\lceil \dfrac{I_2 - I_1 + 1}{T(q_j)} \right\rceil, & [I_1, I_2] \in \mathbb{Z} \\ \left\lceil \dfrac{I_2 - I_1 + 1}{T(q_j)} \right\rceil + 1, & [I_1, I_2] \in \mathbb{Z}_{\leq 0} \end{cases} \tag{4.15}$$

$$n = \frac{L.C.M.(T(q_i), T(q_j))}{T(q_i)} \tag{4.16}$$

where, $I_1$ and $I_2$ are calculated using Eq. 4.12 and Eq. 4.13, respectively. Since it is necessary to know in which hyper-period $q_j$ is being scheduled, the last row in the matrix always represents the current hyper-period even if there are zero iterations of $q_i$ from said hyper-period that send data to $q_j$. Therefore, *m* is incremented if all the integers in the closed interval $[I_1, I_2]$ are non-positive integers so that the *mth* row in the matrix always represents the current hyper-period.

Consider the time-line representation given in Fig. 4.10c for the two vertex DMG example given in Section 4.2.3.3. The closed interval range for this example is $[I_1, I_2] = [-1, 1]$. Here, the first iteration of $q_2$ requires data from the first iteration of $q_1$ from the same hyper-period, and the last two iterations of $q_1$ from the previous hyper-period. To determine which elements in the matrix should contain 'X', it is necessary to first convert all the non-positive integers in the range to positive integers so that they correspond to the indices of the columns in the matrix. For this purpose, the range is first stored in an array of integers. Then the value of $n$ calculated using Eq. 4.16 is added to all non-positive integers in the array until all the values are within [1,n]. In the given example, the closed interval range for the iterations of $q_1$ is $[-1,1]$ and $n$ = *5*. So the integer array contains $\{-1,0,1\}$ and the resultant integer array after adding 5 to all non-positive integers is $\{4,5,1\}$ where all the values are ranging between 1 and 5. After converting the non-positive integers in the array to positive column indices, the next step is to determine the row indices that refer to the calculated column indices. The minimum column index of an element in a row is always 1, so whenever an integer in the array is 1, provided it is not the first integer in the array, it means that the next hyper-period has started. In this case, the row index is incremented. In the illustrated example, the resultant integer array from the previous step is $4,5,1$. Here integers 4 and 5 are the column indexes of the first row whereas integer 1 is the column index in the second row. The resultant relationship matrix for the example is given in Table 4.1. The pseudo-code representation of the algorithm to compute the relationship matrix between a parent query $q_i$ and the first iteration of child query $q_j$ across multiple hyper-periods of $< q_i, q_j >$ DMG is given in Algorithm 1.

The relationship matrix computed in the previous step is used to create directed edges between queries in the simplified DMG. The algorithm traverses through the relationship matrix and creates a directed edge between the *bth* iteration of $q_i$ from the *ath* hyper-period of $< q_i, q_j >$ DMG whenever $rm_{ab} = X$. For the iterations of $q_i$ that are from the previous cyclic execution of the DMG, a dummy vertex is created. This dummy vertex is added as a subset to the set of instances of $q_i$ so that it can be differentiated from the other instances. To determine which subset belongs to which cyclic execution of $< q_i, q_j >$, each subset is assigned the number corresponding its row in the relationship matrix. The WCET of each dummy vertex in the subset is zero since it has already completed its execution in the previous hyper-period, but the time period is equal to the time period of $q_i$. For example, a dummy vertex is created in Fig. 4.13b to depict the first iteration of $q_2$ from the previous hyper-period of $< q_2, q_4 >$ that is sending data to $q_4$. This dummy vertex is added in a separate subset of the overall set of $q_2$ and annotated with its corresponding row number (i.e., 1) in the relationship matrix of $q_2$ and $q_4$. The WCET of the vertex is zero, but it has the same time period and relative deadline as the other subset of $q_2$. A directed edge is then created from this dummy vertex to the first iteration of $q_j$. The weight of this edge is equal to the amount of data transmitted between $q_i$ and $q_j$. The pseudo-code representation of the algorithm used to simplify the DMG is given in Algorithm 2. An example of the transformation is given in Fig. 4.13. A directed edge here describes two characteristics: the minimum number of iterations of parent query required by the child query for its execution, and the iterations of $q_i$ that transfer the data to the first iteration of $q_j$.

| $q_i$ / $H_G$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | - | - | - | X | X |
| 2 | X | - | - | - | - |

Table 4.1: Relationship matrix of parent query $q_1$ and child query $q_2$ for the example given in Fig. 4.10c

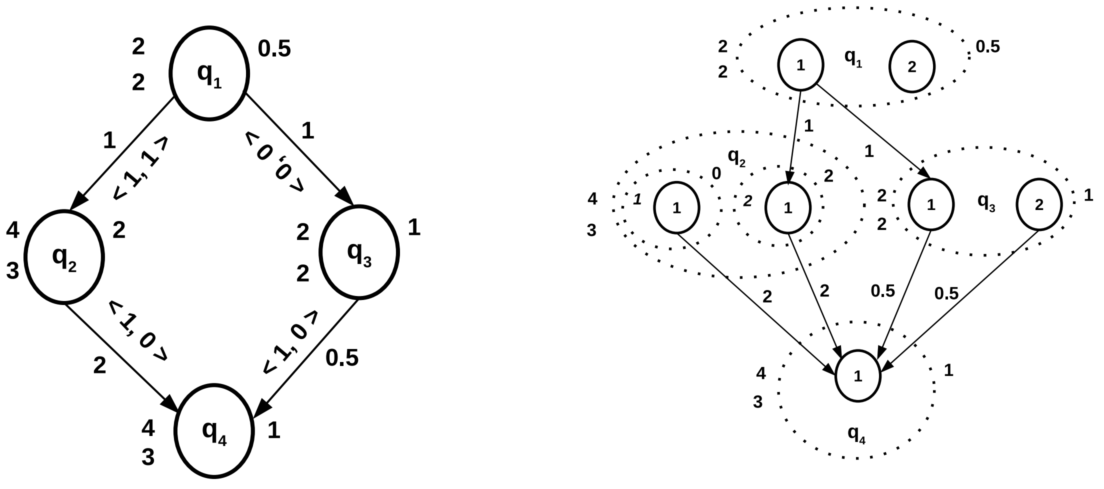


Figure 4.13: (a). Initial form of the DMG (b). DMG after translating the history-interval to directed edges

**Algorithm 1** Algorithm to compute relationship matrix *RM* between the first iteration of a child query $q_j$ and $n$ iterations of parent query $q_i$ across $m$ hyper-periods

```
$[I_1, I_2] \leftarrow$ range of iterations of parent query $q_i$ from which data is required by child query $q_j$
indexes[0,($I_2$-$I_1$+1)] $\leftarrow \{I_1, I_1+1, ..., I_2\}$, array of integers containing the values from the closed interval range $[I_1, I_2]$
$RM((q_i, q_j)) \leftarrow$ Relationship matrix of parent query $q_i$ and child query $q_j$ : $\exists$ a path $p(q_i \rightarrow q_j) \in G_D$
$m \leftarrow$ total number of rows in $RM((q_i, q_j))$ calculated using Eq. 4.15
$n \leftarrow$ total number of columns in $RM((q_i, q_j))$ calculated using Eq. 4.16
for k = 0 to k = $I_1$-$I_2$+1 do
    while indexes[k] < 1 do
        indexes[k] $\leftarrow$ indexes[k] + n
    end while
end for
$temp \leftarrow 0$
$b \leftarrow$, column index of $RM((q_i, q_j))$
$a \leftarrow$ row index of $RM((q_i, q_j))$
for a = 1 to a = m do
    for k = temp to k = $I_1$-$I_2$+1 do
        if indexes[k+1] == 1 then
            $temp \leftarrow k+1$
            break
        end if
        $b \leftarrow indexes[k]$
        $rm_{ab} = X$, element in $RM((q_i, q_j))$ at $ath$ row and $bth$ column
    end for
end for
```

## 4.3 Characteristics of the Application Model used for List Scheduling

The following characteristics that are common to both $G_S$ and $G_D$ are used to schedule the graphs via a list scheduling heuristic.

- *Path Length (pl):* For a task graph $G_S$ represented by Eq. 4.3 or $G_D$ represented by Eq. 4.4, a path represented as $p$ is a sequence of directed edges that connect a finite sequence of tasks [19]. For example, in Fig. 4.5a, $< t_g, t_i, t_j >$ and $< t_g, t_h, t_j >$ are two paths between tasks $t_g$ and $t_j$. Similarly, in Fig. 4.8a, $< q_1, q_2, q_4 >$ and

**Algorithm 2** Algorithm to Translate History-Intervals to Directed Edges

```
$Parent(q_i) \leftarrow$ Set of parent queries of $q_i \in Q : \exists$ a path $p(p_j \rightarrow q_i) \in G_D$
$times(q_i) \leftarrow$ Minimum number of times $q_i$ repeats in one $H_G$
$Iden(q_i) \leftarrow$ Tag for the set containing all iterations of $q_i$
**for** each $q_i \in Q$ **do**
    Create $times(q_i)$ identical query tasks in the graph.
    $Iden(q_i) \leftarrow i$
    **for** each $p_j \in Parent(q_i)$ **do**
        $D(m_{ji}) \leftarrow$ the amount of data transferred between $p_j$ and $q_i$
        Calculate range of iterations of $p_j$ required by $q_i$, $[I_1, I_2]$, using Eq. 4.12 and Eq. 4.13
        $RM((p_j, q_i)) \leftarrow$ Relationship matrix of parent query $p_j$ and child query $q_i : \exists$ a path $p(p_j \rightarrow q_i) \in G_D$
        $m \leftarrow$ total number of rows in $RM((q_i, p_j))$
        $n \leftarrow$ total number of columns in $RM((q_i, p_j))$
        Create $RM((q_i, p_j))$ using Algorithm 1
        **for** $a = 1$ to $a = m$ **do**
            **for** $b = 1$ to $b = n$ **do**
                $rm_{ab} \leftarrow$ element in $RM((q_i, p_j))$ at $ath$ row and $bth$ column
                **if** $rm_{ab}$ == X **then**
                    If $bth$ iteration of $p_j$ does not exist in $ath$ hyper-period of $< q_i, p_j >$ then create a dummy vertex and add it to its corresponding hyper-period set.
                    Create a directed edge between $bth$ iteration of $p_j$ from the $ath$ hyper-period of $< q_i, p_j >$ and the first iteration of $q_i$.
                    Assign the weight $D(m_{ji})$ to the created directed edge.
                **end if**
            **end for**
        **end for**
    **end for**
**end for**
```

$< q_1, q_3, q_4 >$ are two paths between query tasks $q_1$ and $q_4$. The length of a path $p$ in $G_S$ or $G_D$ is the sum of the weights of the vertices and edges in $p$ [19].

$$pl_p = \begin{cases} \sum\limits_{t_i \in T, p} w_i + \sum\limits_{e_{ij} \in E, p} d_{ij}, & p \in G_S \\ \sum\limits_{q_i \in Q, p} W(q_i) + \sum\limits_{m_{ij} \in M, p} D(m_{ij}), & p \in G_D \end{cases} \tag{4.17}$$

where, $w_i$ is the execution time of task $t_i$, $d_{ij}$ is the amount of data transferred on the edge $e_{ij}$, $W(q_i)$ is the WCET of query task $q_i$, $D(m_{ij})$ is the amount of data transferred on the edge $m_{ij}$, and $pl_p$ is a positive real number. A critical path $cp$ is the path that has the greatest length in the graph i.e. $pl_{cp} = \max_{p \in G_S \cup G_D} pl_p$. If the list scheduling heuristic is considered with zero communication cost between tasks, then $pl_{cp}$ is the lower bound of the schedule computed by the heuristic [19].

- *Task Level:* In a task graph $G_S$ represented by Eq. 4.3 or $G_D$ represented by Eq. 4.4, there are numerous paths that start or end at a task $t_i$ or a query task $q_i$. Similar to $cp$, the longest paths among both sets can be distinguished through task levels. These levels are generally used to calculate priorities for the tasks in the graph.

  – *Bottom-Level:* The bottom-level of a task $t \in T$ or a query task $q \in Q$ is the length of the longest path starting from $t$ or $q$. It can be presented as,

$$\begin{aligned} bl(t) &= max_{t_i \in [succ(t) \cap t_{sink}]} \{pl_{t->t_i}\} : \exists \text{ a path } p(t- > t_i) \in G_S \\ bl(q) &= max_{q_i \in [succ(q) \cap q_{sink}]} \{pl_{q->q_i}\} : \exists \text{ a path } p(q- > q_i) \in G_D \end{aligned} \tag{4.18}$$

  where $succ(t)$ and $succ(q)$ are sets of descendants of $t$ and $q$, $t_{sink}$ and $q_{sink}$ are the exit vertices of $G_S$ and $G_D$, $pl_{t->t_i}$ is the length of the path from task $t$ to the task $t_i$, and $pl_{q->q_i}$ is the length of the path from $q$ to $q_i$. Both $bl(t)$ and $bl(q)$ are positive real numbers. If $succ(t) = succ(q) = \emptyset$, then $bl(t) = w_t$ and

$bl(q) = W(q)$.

– *Top-Level:* The top-level of a task $t \in T$ or a query task $q \in Q$ is the length of the longest path ending at $t$ or $q$ excluding the weight of the task or the query. It can be presented as,

$$
\begin{aligned}
tl(t) &= max_{t_i \in [pred(t) \cap t_{source}]}\{pl_{t_i->t}\} - w_t : \exists \text{ a path } p(t_i - > t) \in G_S \\
tl(q) &= max_{q_i \in [pred(q) \cap q_{source}]}\{pl_{q_i->q}\} - W(q) : \exists \text{ a path } p(q_i - > q) \in G_D
\end{aligned}
\tag{4.19}
$$

where $pred(t)$ and $pred(q)$ are set of ancestors of $t$ and $q$, $w_t$ is the execution time of task $t$, $W(q)$ is the WCET of query $q$, $t_{sink}$ and $q_{sink}$ are the entry vertices of $G_S$ and $G_D$, $pl_{t_i->t}$ is the length of the path from task $t_i$ to the task $t$, and $pl_{q_i->q}$ is the length of the path from $q_i$ to $q$. Both $tl(t)$ and $tl(q)$ are positive real numbers. If $pred(t) = pred(q) = \emptyset$, then $tl(t) = tl(q) = 0$.

In this thesis, bottom-level is used when the intended system architecture consists of homogeneous processors whereas top-level combined with bottom-level is used when the processors are heterogeneous.

- *Schedule Length (sl):* Let *Sch* represent the execution plan of tasks/queries and messages present in $G_S$ or $G_D$ on the processors and the communication network, then the schedule length, $sl(Sch) \in \mathbb{R}_{>0}$ of *Sch* can be represented as [19],

$$
sl(Sch) = \begin{cases} max_{t_i \in T}\{FT(t_i)\} - min_{t_i \in T}\{ST(t_i)\} \\ max_{q_i \in Q}\{FT(q_i)\} - min_{q_i \in T}\{ST(q_i)\} \end{cases}
\tag{4.20}
$$

where $FT(t_i)$ and $ST(t_i)$ are start and finish times of task $t_i \in T$, and $FT(q_i)$ and $ST(q_i)$ are the finish and start times of query task $q_i \in Q$. In this thesis, it is assumed that the start time of all the tasks without precedence constraints in $T$ and the start time of all the diagnostic queries without any parent queries in $Q$ is *0*.

- *Communication to Computation Cost Ratio (CCR):* Let $G_S$ be a DAG represented by Eq. 4.3 and $G_D$ be a DMG represented by Eq. 4.4, then $CCR(G_S) \in \mathbb{R}_{>0}$ and $CCR(G_D) \in \mathbb{R}_{>0}$ can be given as [19],

$$\begin{aligned} CCR(G_S) &= \frac{\sum_{e \in E} d_e}{\sum_{t \in T} w_t} \\ CCR(G_D) &= \frac{\sum_{m \in M} D(m)}{\sum_{q \in Q} W(q)} \end{aligned} \tag{4.21}$$

In literature, $CCR$ is sometimes defined as the average edge weight to the average task weight ratio [113].

$$\begin{aligned} CCR(G_S) &= \frac{\overline{d_e}}{\overline{w_t}} \\ CCR(G_D) &= \frac{\overline{D(m)}}{\overline{W(q)}} \end{aligned} \tag{4.22}$$

Where, $\overline{d_e}$ and $\overline{D(m)}$ are the average edge weight of the graphs and $\overline{w_t}$ and $\overline{W(q)}$ are the average task weights of the graphs. Although this definition does not reflect the total communication volume yet it is a much simpler way to calculate the weight of the directed edges in the graph. If task $t_i$ is a parent of task $t_j$ in $G_S$ and query $q_i$ is a parent of query $q_j$ in $G_D$ then the approximate amount of data transmission can be calculated as follows,

$$\begin{aligned} \overline{d_{ij}} &= CCR(G_S) * w_i \quad : \exists \text{ a path } p(t_i \to t_j) \in G_S \\ \overline{D(m_{ij})} &= CCR(G_D) * W(q_i) \quad : \exists \text{ a path } p(q_i \to q_j) \in G_D \end{aligned} \tag{4.23}$$

Where, $w_i$ and $W(q_i)$ are the WCETs of $t_i$ and $q_i$, $\overline{d_{ij}}$ and $\overline{D(m_{ij})}$ is the average amount of data transferred from $t_i$ to $t_j$ and $q_i$ to $q_j$, respectively. In this thesis, $CCR$ values range between medium and low communication loads, i.e., the ratio is between 0.1 and 1.0, and are used to calculate the size of the data transmitted between tasks in $G_S$ and queries in $G_D$.

# CHAPTER 5

# LIST SCHEDULING FOR ACTIVE DIAGNOSIS IN HOMOGENEOUS ODRE SYSTEMS

THIS CHAPTER presents a list scheduling heuristic to schedule diagnostic queries and communication messages in a homogeneous open distributed real-time embedded (ODRE) system. The ODRE systems have requirements for reliable operations with stringent real-time constraints and an open-world assumption. In such systems, an embedded computer system has to provide its services with a dependability that is better than the dependability of its constituent components. If the failure rate of available electrical components in the system is considered, then this level of dependability can only be achieved if the system supports fault-tolerance. In this thesis, fault-tolerance in ODRE systems is supported through a Diagnostic Multi-query Graph (DMG). A DMG is a directed graph of diagnostic queries that is used to detect and diagnose faults in ODRE systems. Scheduling the DMG before executing it on the system depicts the temporal behaviour of the diagnostic application and thus bounds the time to detect and diagnose faults. The scheduling heuristic must also take into account the dynamic nature of ODRE systems, i.e., the integration of new components at runtime to realize global services. Therefore, the scheduling heuristic should be fast in recomputing a feasible schedule during runtime so that the faults can be diagnosed before the system becomes unstable. If no feasible schedule is found then the designer is notified that the changes cannot be integrated to the system. In this scenario, the system keeps running the old schedule. In this chapter, a list scheduling heuristic is proposed to schedule the diagnostic queries and communication messages in a distributed environment where all the processors have the same clock frequency. List scheduling is used because it is faster and more efficient than other static task scheduling algorithms in computing a schedule [19]. This algorithm is used to schedule both system application and the diagnostic graph.

However, in this chapter only the scheduling of diagnostic application is considered with the assumption that the system application is already scheduled using the same algorithm.

## 5.1 Problem Formulation

Given a homogeneous system architecture presented in Section 4.1 and a diagnostic application $G_D$ presented in Section 4.2.2 the following problem has to be solved: construct a feasible static cyclic schedule for the time-triggered diagnostic queries in $G_D$ and the time-triggered message executions such that all the timing constraints of the system are fulfilled.

The system architecture considered in this scenario consists of homogeneous distributed processors, i.e. all the processors in the system have the same clock frequency. Therefore the WCET for each query in $G_D$ is the same throughout the system. The problem is simplified by restricting the time periods of the parent and child queries in $G_D$ to be multiples of each other. An example of the homogeneous distributed system model is given in Fig. 5.1.

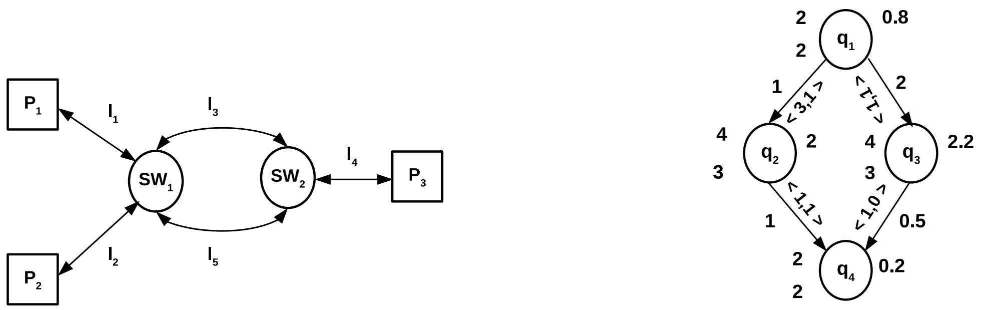


Figure 5.1: Homogeneous Distributed System Model

## 5.2 List Scheduling (LS) for Homogeneous Distributed Systems

The pseudo-code representation of the technique proposed to compute the schedule of the DMG in a homogeneous distributed system is given in Algorithm 3. The purpose of the algorithm is to find a feasible schedule for the given DMG in a bounded time so that whenever there are changes in the system a schedule can be computed before the system becomes

**Algorithm 3** Pseudo-code representation of LS for scheduling DMGs on a homogeneous distributed systems

```
$P \leftarrow$ Set of processors in the system
Calculate the hyper-period $H_G$ of the graph.
Calculate the minimum number of times each query $q_i$ is repeated within one cycle of the
graph, $times(q_i)$.
Translate history-intervals to directed edges.
Compute $bl(q_i)$ for each query $q_i \in Q$.
Assign $bl(q_i)$ to all the incoming edges of each query $q_i \in Q$
while there are unscheduled queries do
    Compute the ready-list $R$.
    for each ready query $r_j \in R$ do
        Order the queries in decreasing order of their bottom-level. The ties are broken by
        prioritising the query that has a child query with a greater bottom-level.
        $Parent(r_j) \leftarrow$ Set of parent queries of ready query $r_j$ : $\exists$ a path $p(p_i \rightarrow r_j) \in G_D$
        for each processor $P_j \in P$ do
            for each parent query $p_i \in Parent(r_j)$ do
                $P_i \leftarrow$ Allocated processor of parent query $p_i \in Parent(r_j)$
                if $P_j \neq P_i$ then
                    Create a communicating message $m_{ij}$ from $P_i$ to $P_j$.
                    for each path $path_k$ between $P_j$ and $P_i$ do
                        for each link $l_m$ in $path_k$ do
                            if $m_{ij}$ can be scheduled on $l_m$ then
                                Calculate the finish time of $m_{ij}$ on $l_m$
                            end if
                        end for
                        The finish time of $m_{ij}$ on $path_k$ is equal to the time the last link in the path
                        took to transmit the information.
                    end for
                    Select the path that gives the earliest finish time to $m_{ij}$.
                end if
            end for
            if $r_j$ can be scheduled on $P_j$ then
                Calculate the start time of $r_j$ on $P_j$
            end if
        end for
        Assign the processor $P_j$ that gives earliest start time to $r_j$ and also fulfills its relative
        deadline $D(r_j)$
    end for
end while
```

unstable and the faults can be identified within the system-defined time. The algorithm follows the concept of list scheduling, i.e. assign priorities to the queries, order the ready queries according to the calculated priorities and then assign them to free processors while satisfying the precedence constraints of the DMG. Along with scheduling the diagnostic queries, the algorithm also schedules the data transmission between the queries. The edges in the graph are assigned the priority of their destination query and are allocated to the communication path that computes least communication time for them. There are four main steps of the algorithm: (i). Priority assignment, (ii). Adding queries to ready list, (iii). Ordering the ready list and (iii). Processor selection for query and path selection for messages. Before starting the scheduling phase, the algorithm first calculates the hyper-period $H_G$ of the given DMG using Eq. 4.9. Then the minimum number of times each query $q_i$ repeats in one $H_G$ is calculated using Eq. 4.10. Lastly, Algorithm 2 is used to convert the history-intervals to directed edges.

### 5.2.1 Proposed Algorithm

1. ***Priority assignment:*** Before adding the queries to the ready list, the algorithm assigns a static priority to each query in the DMG. For this purpose, bottom-level of each query $q_i \in Q$ is computed using Eq. 4.18. Static priority here means that once a priority is assigned to a query it remains unchanged throughout the computation of the schedule. Pseudo-code representation of the algorithm used to calculate the bottom-level of a query $q_i$ is given in Algorithm 4. The priority calculated for each query $q_i$ is also assigned to the incoming edges of said query.

2. ***Query Selection:*** The next step is to add all the ready queries to a ready list that is represented by $R$. A query $q_i$ is said to be ready if all of its predecessors have completed their execution [19]. The algorithm traverses through the graph and adds all the ready queries to $R$. Here, it is essential to determine the number of iterations of the parent query that send the data to the child query. Only when all the required iter-

**Algorithm 4** Pseudo-code representation for calculating bottom-level of a query $q_i$

```
$Children(q_i) \leftarrow$ Set of successor queries of $q_i : \exists$ a path $p(q_i \rightarrow c_j) \in G_D$
$D(m_{ij}) \leftarrow$ Data transmitted from $q_i$ and $c_j : \exists$ a path $p(q_i \rightarrow c_j) \in G_D$
$W(q_i) \leftarrow$ WCET of $q_i$
$bl(c_j) \leftarrow$ Bottom-level of $c_j$
$bl(q_i) \leftarrow W(q_i)$, Bottom-level of $q_i$
for each child query $c_j \in Children(q_i)$ do
    $bl(q) \leftarrow bl(c_j) + W(q_i)$
    if $bl(q) > bl(q_i)$ then
        $bl(q_i) \leftarrow bl(q)$
    end if
end for
```

ations of the parent queries are completed then the child query can start its execution. The graph obtained after the translation of history-intervals to directed edges is used to determine the required iterations. For example in Fig. 4.13b, query $q_2$ can only start its execution after the completion of the first iteration of query $q_1$ whereas query $q_4$ can only start its execution after the first iterations of $q_2$ and $q_3$ are completed. The simplified DMG created using Algorithm 2 is used to determine the starting point of each query in the graph. If all the parent queries have completed their required iterations then the child query is added to $R$. The ready-time of a query $q_i$ is given as,

$$r_t(q_i) = \max_{q_j \in pred(q_i)} f_t^k(q_j) : \exists \text{ a path } p(q_j \rightarrow q_i) \in G_D \tag{5.1}$$

where, $f_t^k(q_j)$ is the finish time of the *kth* required iteration of $q_j$ and $pred(q_i)$ is the set of predecessor queries of $q_i$. The ready-time of a query is also called its release time. If there are no predecessors of $q_i$ i.e., $pred(q_i) = \emptyset$ then $r_t(q_i) = t = 0$. The ready-list $R$ is a set of queries and can be defined as,

$$R = \{q_i \in Q : \text{global time is equal to } r_t(q_i)\} \tag{5.2}$$

3. ***Order the ready list R*****:** In this step, the ready list $R$ obtained in step 2 (cf. Eq . 5.2) is ordered in descending order of the bottom-level of the queries in $G_D$. In case of ties, the query that lies in the critical path or has a child query with max bottom-level among children of other queries is ranked higher in the ready-list.

4. ***Path and processor selection*****:** After the generation and order of the ready list $R$, the next step is to select a processor for the execution of the query. The algorithm traverses through the list of processors and assigns the query to the processor that gives it the earliest start time and on which the query fulfils its relative deadline. For a processor to execute a query, the query must not overlap or hinder the execution of the queries already scheduled on the processor. To ensure a query $q_i$ does not overlap other queries scheduled on a processor $P_j$, Algorithm 5 is used. The algorithm traverses through the list of queries already scheduled on $P_j$ and determines whether the execution of $q_i$ overlaps the already scheduled queries on $P_j$. If an overlap occurs between any iteration of $q_i$ and $q_j$ already scheduled on $P_j$ then $q_i$ cannot be scheduled on $P_j$. Moreover, a query cannot start its execution until all of its required data is transmitted to the selected processor. This has been explained later in the chapter. The earliest start time of a query $q_i$ on a processor $P_j$ can be given as,

$$EST(q_i,P_j) = DRT(q_i,P_j) \tag{5.3}$$

where, $DRT(q_i,P_j)$ is the time at which all the data required by $q_i$ is available at $P_j$. For computing the start time of the query on the processor, it is first necessary to schedule the incoming edges of the query. It is assumed that intra-processor communication cost, i.e. communication between queries assigned to the same processor is zero. This assumption is following the fact that the communication on the same processor is cheap than communication between different processors and is therefore negligible [19]. If the parent and child queries are allocated to different processors, then a message is created and transmitted using the path that gives the least communication cost to the message. Since the links in the distributed system have different bandwidths, therefore the communication time might be different on each link in the same path. For complete transference of data, a link must not start transmitting a message before its previous link has completed its transmission. Consider a message $m_{ij}$ transmitting from processor $P_i$ to $P_j$ through a path $path_{ij}$ that has $k$ number of links or hops then the earliest start time of the message $m_{ij}$ on a link $l_n \in path_{ij}$ where $n$ is a natural number ranging from 1 to k can be given as,

$$EST(m_{ij}, l_n) = \max_{l_n \cap l_{n-1} \in path_{ij}} \{FT(m_{ij}, l_{n-1}), EAT(l_n)\} \tag{5.4}$$

where, $FT(m_{ij}, l_{n-1})$ is the finish time of $m_{ij}$ on the previous link $l_{n-1}$ in the path $path_{ij}$ and $EAT(l_n)$ is the earliest available time of $l_n$. The finish time of $m_{ij}$ on $l_n$ is,

$$FT(m_{ij}, l_n) = ST(m_{ij}, l_n) + c(m_{ij}, l_n) \tag{5.5}$$

where, $ST(m_{ij}, l_n)$ is the start time of $m_{ij}$ on $l_n$ and $c(m_{ij}, l_n)$ is the communication cost of $m_{ij}$ on $l_n$ calculated using Eq. 4.2. The finish time of $m_{ij}$ on $path_{ij}$ is the maximum finish time of $m_{ij}$ on the last link in the path i.e. the kth link. The finish time of $m_{ij}$ can then be represented as,

$$FT(m_{ij}, path_{ij}) = FT(m_{ij}, l_k) \tag{5.6}$$

The path between processors $P_i$ and $P_j$ that satisfies $min\{FT(m_{ij}, path_{ij})\}$ is selected for transmission of the message $m_{ij}$. The queries transmitting the messages are periodic, which means that the transmission is also periodic. Therefore, each message is assigned the greater time period among the transmitting queries and is repeated the number of times the query with a greater time period is repeated. Moreover, all the iterations of the message are transmitted using the same path. Similar to the processors, a message cannot be scheduled on a link if it overlaps the execution of other messages scheduled on the link. A message $m_{ij}$ can be transmitted on a path $path_{ij}$ if it does not overlap the messages already scheduled on any link in the path. The pseudo-code representation of the algorithm used to ensure that a message $m_{ij}$ does not overlap any other messages on a link $l_n$ is given in Algorithm 6. If all the incoming messages of $q_i$ are scheduled successfully and Algorithm 5 does not return any query that is scheduled in the same duration as $q_i$ then $q_i$ is allocated to $P_j$ and removed from the ready-list $R$. All the iterations of $q_i$ are scheduled on the same processor.

**Algorithm 5** Pseudo-code representation for ensuring a query $q_i$ can be scheduled on the processor $P_j$

```
$W(q_i,P_j) \leftarrow$ WCET of query $q_i$ on processor $P_j$
$T(q_i) \leftarrow$ time period of query $q_i$
$a \leftarrow$ Iteration number of query $q_i$
$b \leftarrow$ Iteration number of query $q_j$ already scheduled on processor $P_j$
$H_G \leftarrow$ Hyper-period of the graph $G_D$
$W(q_j,P_j) \leftarrow$ WCET of query $q_j$ already scheduled on processor $P_j$
$T(q_j) \leftarrow$ time period of query $q_j$
for $a = 0$ to $a = \frac{H_G}{T(q_i)}$ do
  $ST(q_i,P_j) \leftarrow$ Start time of the $ath$ iteration of query $q_i$ on processor $P_j$
  duration = $ST(q_i,P_j) + W(q_i,P_j)$
  for $b = 0$ to $b = \frac{H_G}{T(q_j)}$ do
    $ST(q_j,P_j) \leftarrow$ Start time of the $bth$ iteration of query $q_j$ on processor $P_j$
    d = $ST(q_j,P_j) + W(q_j,P_j)$
    if ($ST(q_i,P_j) \geq ST(q_j,P_j) <$ d) OR ($ST(q_i,P_j) \geq$ duration $<$ d) then
      $q_i$ can not be scheduled on $P_j$
      Exit
    else if ($ST(q_j,P_j) \geq ST(q_i,P_j) <$ d) OR ($ST(q_j,P_j) \geq$ d $<$ duration) then
      $q_i$ can not be scheduled on $P_j$
      Exit
    end if
  end for
end for
```

**Algorithm 6** Pseudo-code representation for ensuring a message $m_{ij}$ can be scheduled on the link $l_n$

```
$c(m_{ij}, l_n) \leftarrow$ Communication cost of message $m_{ij}$ on link $l_n$
$T(m_{ij}) \leftarrow$ time period of message $m_{ij}$
$a \leftarrow$ Iteration number of message $m_{ij}$
$b \leftarrow$ Iteration number of message $M_{ij}$ already scheduled on link $l_n$
$H_G \leftarrow$ Hyper-period of the graph $G_D$
$c(M_{ij}, l_n) \leftarrow$ Communication cost of message $M_{ij}$ already scheduled on link $l_n$
$T(M_{ij}) \leftarrow$ time period of message $M_{ij}$
for $a$ = 0 to $a = \frac{H_G}{T(m_{ij})}$ do
    $ST(m_{ij}, l_n) \leftarrow$ Start time of the $ath$ iteration of message $m_{ij}$ on link $l_n$
    duration = $ST(m_{ij}, l_n) + c(m_{ij}, l_n)$
    for $b$ = 0 to $b = \frac{H_G}{T(M_{ij})}$ do
        $ST(M_{ij}, l_n) \leftarrow$ Start time of the $bth$ iteration of message $M_{ij}$ on link $l_n$
        d = $ST(M_{ij}, l_n) + c(M_{ij}, l_n)$
        if ($ST(M_{ij}, l_n) \geq ST(m_{ij}, l_n) <$ d) OR ($ST(M_{ij}, l_n) \geq$ duration $<$ d) then
            $m_{ij}$ can not be scheduled on $l_n$
            Exit
        else if $ST(m_{ij}, l_n) \geq ST(M_{ij}, l_n) <$ duration) OR ($ST(m_{ij}, l_n) \geq$ d $<$ duration)
        then
            $m_{ij}$ can not be scheduled on $l_n$
            Exit
        end if
    end for
end for
```

From the mentioned steps, steps [2-4] are repeated until all the queries in the DMG are scheduled.

### 5.2.2 Example

Consider the DMG given in Fig. 5.2a. The target 3-processor homogeneous distributed system architecture is given in Fig. 5.3. The system has single bi-directional communication paths from $P_1$ to $P_2$ ($< l_1, l_2 >$), $P_1$ to $P_3$ ($< l_1, l_3, l_4 >$) and $P_2$ to $P_3$ ($< l_2, l_3, l_4 >$). Using Eq. 4.9, the hyper-period of the DMG is calculated as 8 ms. After that Eq. 4.10 is used to calculate the minimum number of times each query is repeated in the graph. Queries $q_1$, $q_2$ and $q_3$ are repeated at least twice, $q_5$ and $q_6$ are executed only once whereas $q_4$ is repeated at least four times in one execution cycle. The algorithm then translates the history-intervals to directed edges as shown in Fig. 5.2b. The bottom-level calculated for the queries using Eq. 4.18 are: $bl(q_1) = 13$, $bl(q_2) = 11$, $bl(q_3) = 10$, $bl(q_4) = 5$, $bl(q_5) = 8$, and $bl(q_6) = 3$. At time *0 ms*, only query $q_1$ is ready for execution so it is assigned to processor $P_1$. At *1 ms*, both $q_2$ and $q_3$ are ready but $q_2$ is given priority because it has a greater bottom-level. Processor $P_1$ gives earliest start time to $q_2$ while fulfilling its absolute deadline i.e., *5 ms*. For $q_3$ with an absolute deadline *5 ms*, processor $P_2$ is selected and the message from $P_1$ to $P_2$ is transmitted on path $< l_1, l_2 >$. Since $q_4$ requires only one iteration of $q_3$, therefore it is ready for execution at *3.5 ms*. Earliest start time of $q_4$ at $P_1$ is *5 ms* and at $P_2$, it is *3.5 ms*. Therefore, it is assigned to $P_2$. Query $q_5$ is ready at *7.5 ms* after the second iteration of $q_3$. Processors $P_1$ and $P_2$ cannot execute $q_5$ so it is allocated to $P_3$. Messages $m_{25}$ and $m_{35}$ are transmitted on paths $< l_1, l_3, l_4 >$ and $< l_2, l_3, l_4 >$, respectively. The calculation for the communication time of the messages is slightly tricky here because both paths share links $l_3$ and $l_4$. Both messages start transmitting from their respective processors at *7.5 ms*. Since link $l_2$ is faster than link $l_1$ so $m_{35}$ reaches link $l_3$ first at *8 ms*. When $m_{25}$ reaches $l_3$, it has to wait for $m_{35}$ to finish its transmission on $l_3$ that ends at *9 ms*, at which point $m_{25}$ starts transmitting again. In Fig. 5.4, it is visible that $m_{25}$ gets blocked by $m_{35}$ for *1.5 ms* and starts transmitting again as soon as link $l_3$ is free. After that there are no collisions on the path and $m_{35}$ reaches $P_3$ at *9.5 ms* and $m_{25}$ reaches at *10.5 ms*. Query $q_5$ completes its execution at *14.5 ms*. Lastly, again it is not possible to execute $q_6$ on either $P_1$ or $P_2$ so it is allocated to $P_3$. Message $m_{46}$ is transmitted on path $< l_2, l_3, l_4 >$. Thus, $q_6$ starts its execution at *16.5 ms* and completes at *19.5 ms* which is the schedule length for one cyclic execution of given $G_D$. The timeline representation of the schedule is given in Fig. 5.4. Here, $path_{12}$ is $< l_1, l_2 >$, $path_{23}$ is $< l_2, l_3, l_4 >$ and lastly $path_{13}$ is $< l_1, l_3, l_4 >$.

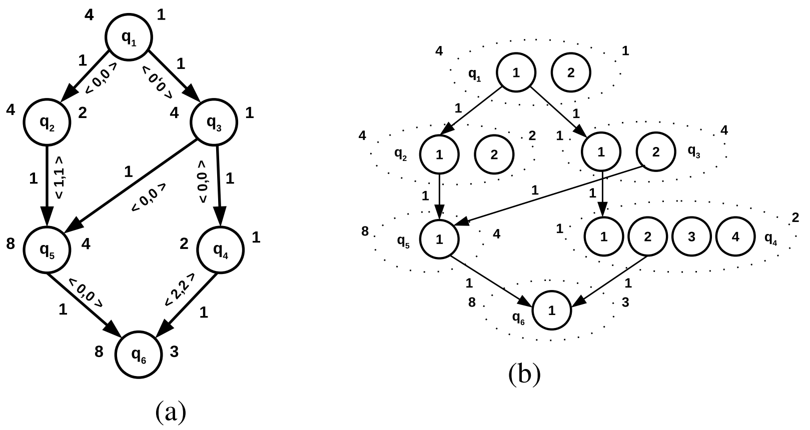


Figure 5.2: (a). DMG - $G_D$ (here D($q_i$) = T($q_i$)) (b). $G_D$ after translating the history-interval to directed edges (here D($q_i$) = T($q_i$))

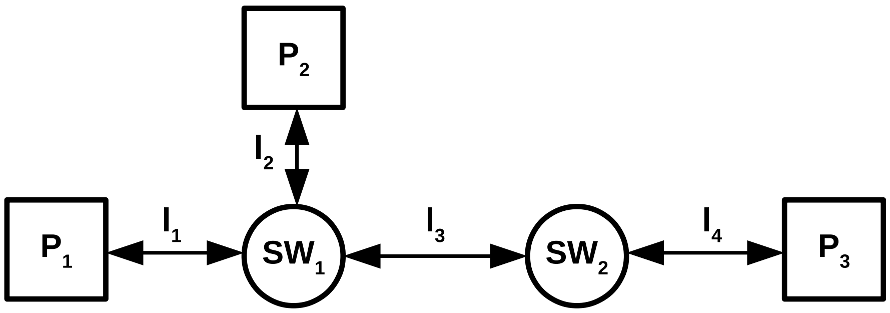


Figure 5.3: Homogeneous distributed system ($l_1 = l_3 = 1$ Mb/ms, $l_2 = l_4 = 2$ Mb/ms)

## 5.2.3 Complexity of LS algorithm for homogeneous systems

In the LS algorithm for homogeneous distributed systems, the time complexity to compute the bottom-level of queries using a depth-first search (DFS) is $O(|Q| + |M|)$ [129]. Sort-

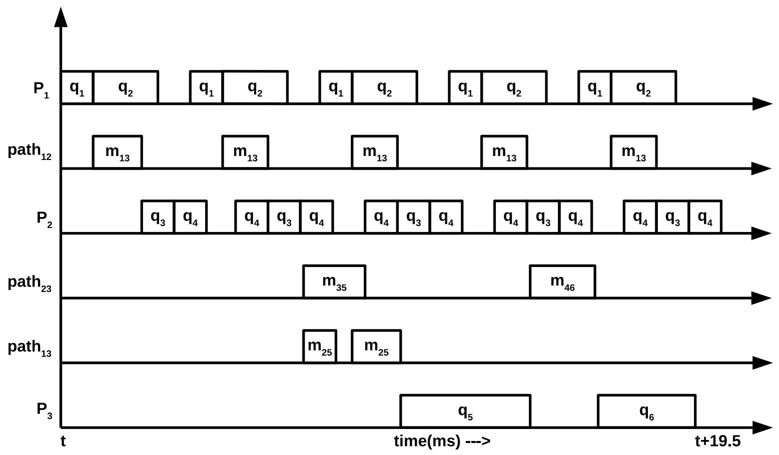


Figure 5.4: Schedule representation for $G_D$ given in Fig. 5.2a implemented on architecture given in Fig. 5.3 (t = 0 ms)

ing the queries has a complexity of $O(|Q|log|Q|)$. Computing the time required for each message on each path has a worst-case time complexity of $O(|Q|+|M|)|P||L|)$ and finding the shortest path for the message communication has a complexity of $O(|L|log|L|)$. Lastly, allocating the query to the processor that gives it the earliest start time has a worst-case complexity of $O(|Q|+|M|)|P|$. The complexity of the algorithm majorly depends upon the number of nodes and edges in the graph and the communication network.

## 5.3 Experimental Setup

In this section, various experiments are performed to check the validity of the proposed algorithm. The experiments are performed on a platform with four Intel Core i5-6200U cores containing 23 GB RAM and operating at a fixed clock frequency of 2.30 GHz. The operating system used is fedora 27 with kernel Linux 5-generic. The algorithm is designed in C and implemented on an online-server.

A set of randomly generated graphs are used to test the LS algorithm for different parameters. The following characteristics, that are widely used to generate random graphs [19], are used to generate test DMGs.

1. Total number of queries in the DMG, i.e., $|Q|$.

2. Probability of a directed edge between two queries, i.e. *prob*.

3. Time period $T(q)$ of a query $q \in Q$. They are generated from the geometric sequence $\{T, T*r, T*r^2, ..., T*r^k\}$ where $r \neq 0$ and $k$ is a non-negative integer. A geometric sequence is used because the algorithm is applied to DMGs that have either the same or multiple periods. If an arithmetic sequence is used, then it cannot be guaranteed that the time periods of two queries are multiple of each other.

4. Relative deadline $D(q)$ of a query $q \in Q$. It is equal to the time period of the query, i.e., $T(q)$.

5. A utilization factor, $UT(q) \in \mathbb{R}_{>0}$ is used to calculate the WCET of each query. $UT(q)$ is the ratio of WCET of a query $q$ to its time period i.e. $\frac{W(q)}{T(q)}$. The value for the utilization factor lies in the range $(0,1)$ with both 0 and 1 excluded.

6. History interval $< a_{ij}, b_{ij} >$ between queries $q_i$ and $q_j$, where $q_i$ is the parent of $q_j$, is generated randomly such that is generated randomly from an integer arithmetic sequence such that $a_{ij} \in \mathbb{Z}_{\geq 0}$ and $b_{ij} \in \mathbb{Z}_{b_{ij} \geq 0 \cup b_{ij} < a_{ij}}$.

7. Eq. 4.23 is used to calculate the average amount of data transferred between the queries. The unit for this is megabits.

The mentioned characteristics are used to generate 100 DMGs each of 100, 150 and 200 queries per graph. The probability of an edge between two queries in the graph is 0.5. The time period of each query is taken from the geometric range $\{4, 12, 36, 108\}$ where $T = 4$, $r = 3$ and $k = 3$ measured in milliseconds and $UT$ is 0.5. Thus the range of $W(q)$ is from 2 to 54 ms. The history-interval $< a_{ij}, b_{ij} >$ is chosen from the arithmetic range $\{0, 1, 2, ..., 10\}$ such that $b_{ij} \leq a_{ij}$. The *CCR* value is either 0.1, 0.5 (low communication loads) or 1 (medium communication load). The amount of data transferred between the queries is given in megabits.

The system contains distributed clusters connected through a defined network topology. Following characteristics are used to generate the system architectures.

1. Total number of processors in the distributed system, P.
2. Total number of processors, $P_{SN}$ and switches $SW$ in each cluster.
3. Network topology used in each cluster and the overall system.
4. Total number of links per cluster represented by $L$.
5. Rate of transmission $w_k$ of a link $l_k$ in a cluster. It is randomly generated from the arithmetic sequence $\{w_{min}, w_{min} + \beta, ..., w_{max}\}$ such that the overall bandwidth of the cluster is always equal to $\frac{w_{max}+w_{min}}{2}$. $\beta$ is the common difference between two consecutive numbers of the sequence.

For the system architecture, the total number of processors is varied between 8, 16, 32 and 64. There are four processors in each cluster, so the total number of distributed clusters are 2, 4, 8 or 16. Each cluster uses a star topology with one switch and one bi-directional link between a processor and the switch. The bandwidths of the links are generated from the arithmetic sequence $\{60, 100, 140, 180, 220\}$. The rates are given in Mbps. For the overall system, a ring topology is used.

The metrics used to explain the results are scheduling length (cf. Eq. 4.20) and scheduling rate of the DMGs, i.e., the total number of DMGs in a set that are successfully scheduled (cf. Eq. 5.7). These metrics were measured against the total number of queries per DMG, communication to computation cost ratio, and the total number of processors in the distributed system.

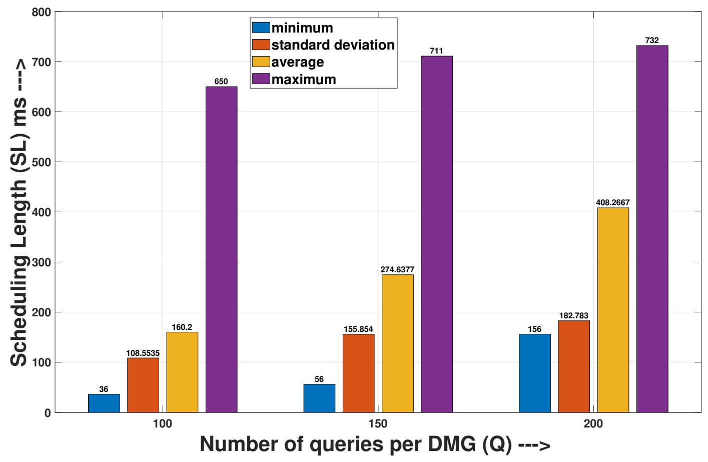
$$\%Rate = \frac{\text{total number of DMGs scheduled in the set}}{\text{total number of DMGs in the set}} * 100 \tag{5.7}$$

## 5.4 Results

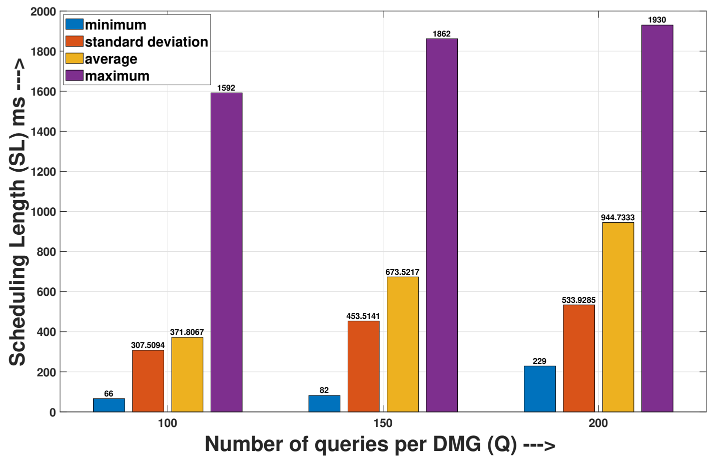
This section presents an analysis of the results obtained from the experiments detailed above. In the first set of experiments, the trend in the scheduling length is observed by a varying number of queries in the DMG. Consider Fig. 5.5 where 100 instances of 100, 150 and 200 query task graphs are scheduled on a 64-processor ring based homogeneous distributed system. The experimental setup is considered with low and medium communication loads. The graphs show that the average scheduling length generally increases with an increase in the number of queries. For example, in Fig. 5.5b, the average scheduling length for 100 instances of 200 query task graphs is approximately 154% greater than the average scheduling length for the same amount of 100 query task graphs. The parallelization factor of the DMGs plays an important role in the increase in the scheduling length. For example, consider a 100 query graph scheduled on a 4-processor distributed system. If the graph has a maximum parallelization factor of three (i.e. at maximum three queries can execute parallel to each other), it means that at maximum it uses only three processors at a time. Now if the graph size is increased to 150 queries with the same parallelization factor, then the DMG still at maximum uses only three processors which in turn increases the schedule length. If the parallelization factor is increased to four in the latter case, there would still be an increase in *SL* but not as high as before. The parallelization factor also plays a role in determining the schedulability of the DMG on the target distributed system. If there are enough resources to cater to the increase in DMG size, then the schedulability ratio is high.

In the next set of experiments, the scalability of the algorithm was measured against the computation to communication cost ratio (CCR). For these experiments, the trend of scheduling length and schedulability rate is observed when the communication load on the network is increased. Consider Fig. 5.6a where results of the scheduling length obtained by scheduling 100 instances of 100 query task graphs on a 16-processor ring distributed system are presented. Here, the communication load is varied from low, i.e. 0.1 and 0.5 to medium, i.e. 1.0. The results show that as the communication load increases, the overall schedule length is increased, for example, in Fig. 5.6a the scheduling length is approximately four times more when CCR is 1.0 than when CCR is 0.1. A similar pattern is observed for the other results, as shown in Fig. 5.6b and Fig. 5.6c. The increase in the schedule length can be designated to the fact that an increase in the amount of data transferred between queries increases the time required for transmission if the bandwidth remains unchanged. The results also showed that an increase in the CCR value lowers the schedulability of the DMGs. It can be attained to the fact that a message that was previously schedulable on a link may now overlap other messages scheduled on the link with its increased duration making it unschedulable.

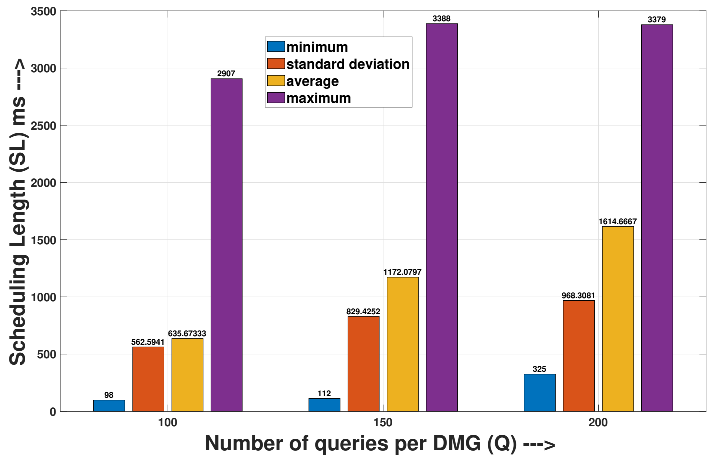
Lastly, to study the effect of increasing the number of clusters (processors and links) on the scheduling length, 100 different instances of 100 query task graphs are implemented on 8, 16, 32, and 64-processor ring based distributed systems. The number of links for the mentioned system architectures is 9, 20, 40, and 96, respectively. The communication load is varied from low to medium. The results (Fig. 5.7) show that the scheduling length decreases with an increase in the number of resources. Although the scheduling length tends to converge to the same result even if the resources are further increased. The parallelization factor plays a role in this scenario, i.e. if the system can execute only three queries in parallel than the scheduling length does not change even if the number of processors is increased from 4 to 100. The results also showed that the schedulability ratio increases with an increase in the number of resources. This increase is understandable since a greater number of resources increases the chances of obtaining a schedule.



(a) CCR = 0.1

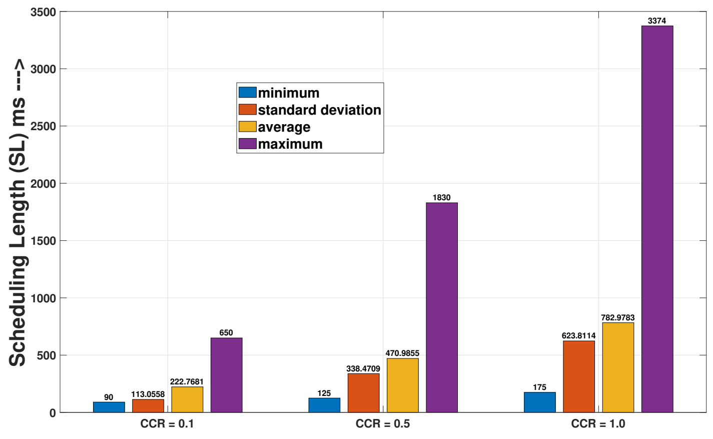


(b) CCR = 0.5

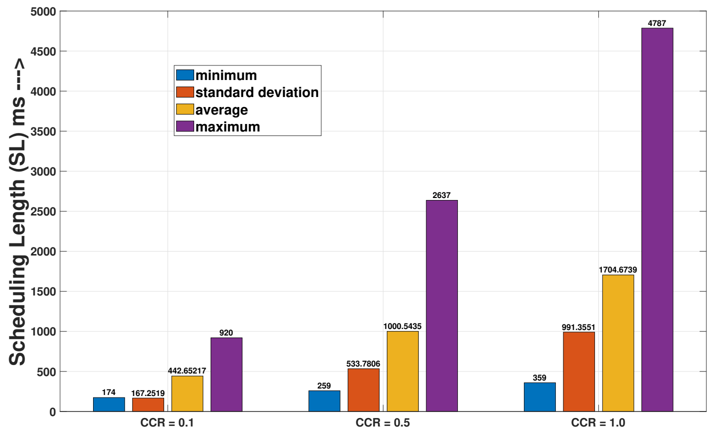


(c) CCR = 1.0

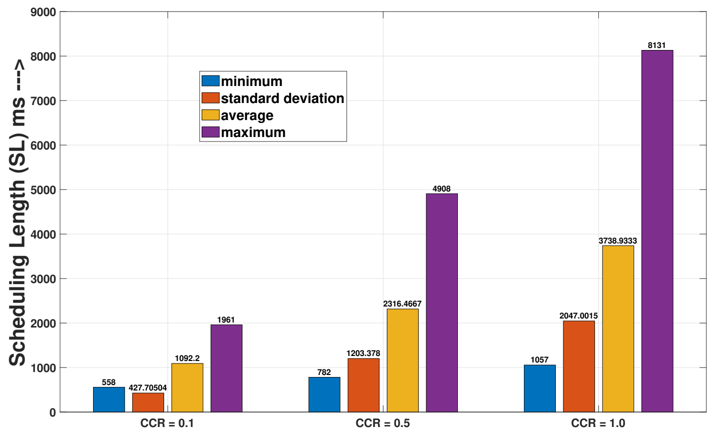
Figure 5.5: Variation in SL with increase in size of DMG (64-ring distributed system)



(a) Q = 100

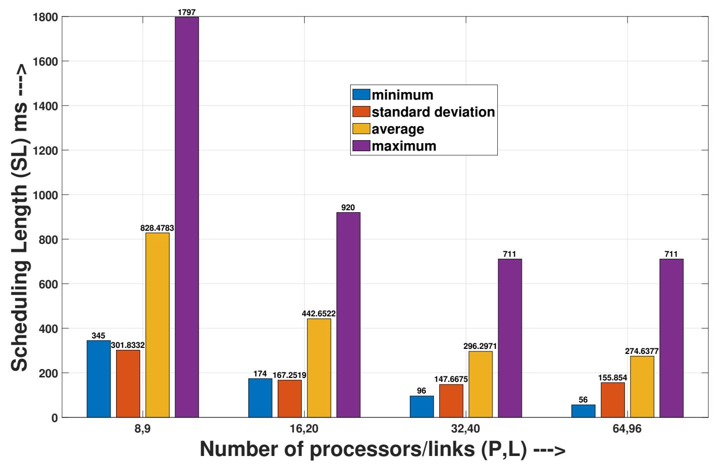


(b) Q = 150

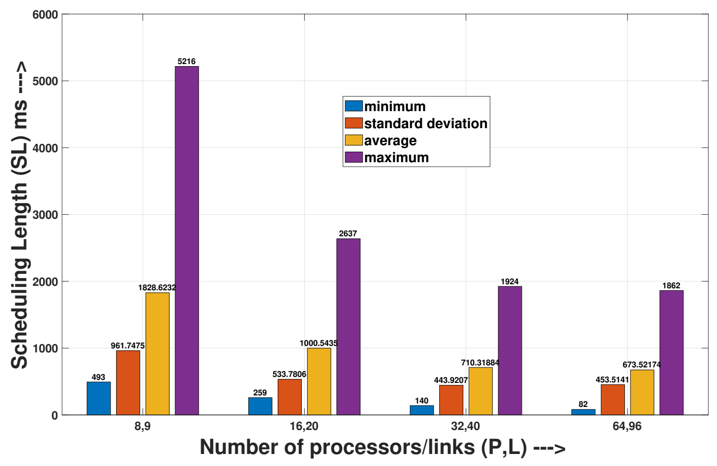


(c) Q = 200

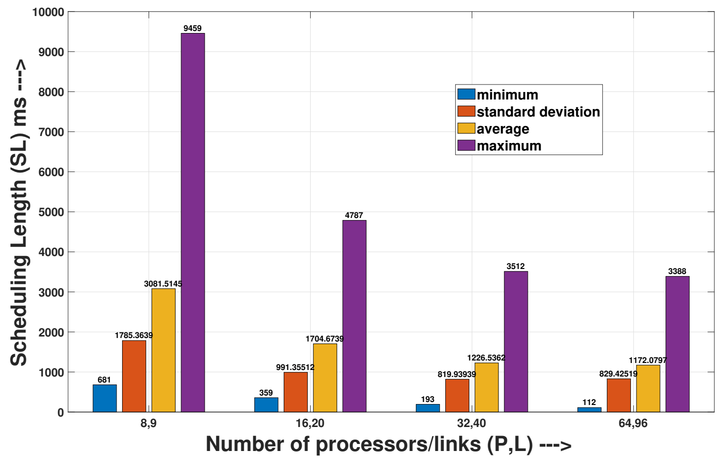
Figure 5.6: Variation in SL with increase in CCR (16-ring distributed system)



(a) CCR = 0.1



(b) CCR = 0.5



(c) CCR = 1.0

Figure 5.7: Variation in SL with increase in number of processors and links (Q = 100)

# CHAPTER 6
# LIST SCHEDULING FOR ACTIVE DIAGNOSIS IN HETEROGENEOUS ODRE SYSTEMS

THIS CHAPTER presents a list scheduling heuristic to schedule diagnostic queries and communication messages in a heterogeneous open distributed environment. As explained in previous sections, Open Distributed Real-Time Embedded (ODRE) systems have requirements for reliable operations with strict timing constraints and an open-world assumption. In such systems, an embedded computer system has to provide its services with a dependability that is better than the dependability of its constituent components. Considering the failure rate of electrical components, one way to achieve this level of dependability is to make the ODRE system fault-tolerant. This thesis uses diagnostic queries to detect and diagnose faults in ODRE systems. The diagnostic queries are represented in the form of a Diagnostic Multi-query Graph (DMG). Scheduling the diagnostic queries before executing them on the system depicts their temporal behaviour and also bounds the time to detect and diagnose faults. In ODRE systems, the term openness means that the electrical components can leave and enter the system at runtime and there is a time-bound in which the schedule must be recomputed to continue the normal and accurate functioning of the system. Therefore, the scheduling heuristic must be fast and efficient so that the schedule can be recalculated before the system becomes unstable. The time complexity of the list scheduling heuristic is pseudo-polynomial that makes it an efficient heuristic to solve the real-time scheduling problem. If a feasible schedule is not found then the designer is notified that the changes cannot be integrated to the system. In this scenario, the system keeps running the old schedule. In this chapter, a modified version of the traditional list scheduling heuristic is proposed to schedule the DMG to diagnose faults in heterogeneous open distributed embedded systems. The proposed algorithm is used to schedule both system application and diagnostic graphs. This chapter only focuses on the scheduling of the diagnostic application and assumes that the system application is already scheduled.

## 6.1 Problem Formulation

Given a heterogeneous system architecture presented in Section 4.1 and a diagnostic application $G_D$ presented in Section 4.2.2 the following problem has to be solved: construct a feasible static cyclic schedule for the time-triggered diagnostic queries in $G_D$ and the time-triggered message executions such that all the timing constraints of the system are fulfilled.

The system architecture considered in this scenario consists of heterogeneous processors. It means that the processors have different clock frequencies and compute the queries with different execution times. The worst-case execution time (WCET) of each diagnostic query $q_i \in Q$ is represented in a $m$ $x$ $n$ computation cost matrix $W$. The $m$ rows in this matrix represent the queries, whereas the columns $n$ represent the processors in the overall system. It means that there are total $m$ queries in the DMG and total $n$ processors in the system. Each $w_{ij}$, where $i = 1,2,...,m$ and $j = 1,2,...,n$, gives the worst-case execution time of query task $q_i$ on processor $P_j$. An example of the computation matrix is given in Table 6.1 for the heterogeneous distributed system model of Fig. 6.1.

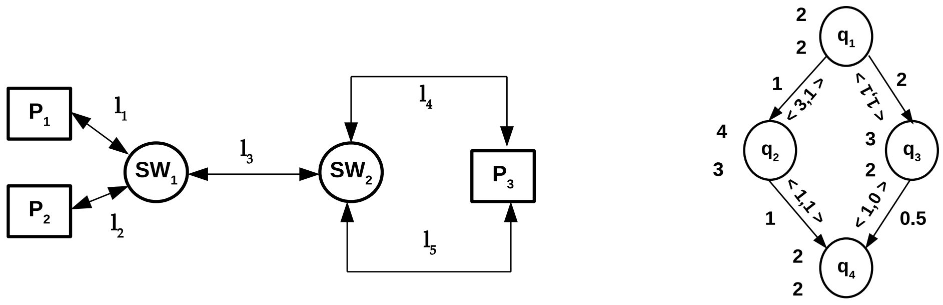


Figure 6.1: Heterogeneous Distributed System Model

**Algorithm 7** Pseudo-code representation of LS for scheduling DMGs on a heterogeneous distributed systems

```
$P \leftarrow$ Set of processors in the system
Calculate the hyper-period $H_G$ of the graph.
Calculate the minimum number of times each query $q_i$ is repeated within one cycle of the graph, $times(q_i)$.
Translate history-intervals to directed edges.
Compute the $\overline{W(q_i)}$ for each query $q_i \in Q$.
Compute $bl(q_i)$ for each query $q_i \in Q$.
Assign $bl(q_i)$ to all the incoming edges of each query $q_i \in Q$
while there are unscheduled queries do
  Compute the ready-list $R$.
  for each ready query $r_j \in R$ do
    $D(r_j) \leftarrow$ Deadline of $r_j$.
    Compute $tl(r_j)$ for $r_j$.
    Assign $tl(r_j)$ to all incoming edges of $r_j$
    Order the queries in increasing order of their top-level. The ties are broken by giving priority to the query with greater bottom-level.
    $Parent(r_j) \leftarrow$ Set of parent queries of ready query $r_j$ : $\exists$ a path $p(p_i \rightarrow r_j) \in G_D$
    for each processor $P_j \in P$ do
      for each parent query $p_i \in Parent(r_j)$ do
        $P_i \leftarrow$ Allocated processor of parent query $p_i \in Parent(r_j)$
        if $P_j \neq P_i$ then
          Create a communicating message $m_{ij}$ from $P_i$ to $P_j$.
          for each path $path_k$ between $P_i$ and $P_j$ do
            for each link $l_m$ in $path_k$ do
              if $m_{ij}$ can be scheduled on $l_m$ then
                Calculate the finish time of $m_{ij}$ on $l_m$
              end if
            end for
            The finish time of $m_{ij}$ on $path_k$ is equal to the time the last link in the path took to transmit the information.
          end for
          Select the path that gives the earliest finish time to $m_{ij}$.
        end if
      end for
      if $r_j$ can be scheduled on $P_j$ then
        Calculate the finish time of $r_j$ on $P_j$
      end if
    end for
    Select the processor that gives the earliest finish time to $r_j$ and on which $r_j$ fulfills its relative deadline $D(r_j)$.
  end for
  Update the weights of the edges and the already scheduled queries.
end while
```

| **Queries** \ **Processors** | $\mathbf{P_1}$ | $\mathbf{P_2}$ | $\mathbf{P_3}$ |
|---|---|---|---|
| $\mathbf{q_1}$ | 0.5 | 1 | 1.5 |
| $\mathbf{q_2}$ | 1.75 | 2.25 | 2.75 |
| $\mathbf{q_3}$ | 0.65 | 1.15 | 1.65 |
| $\mathbf{q_4}$ | 1 | 1.5 | 2 |

Table 6.1: Computation Cost Matrix $W$ for Fig. 6.1

## 6.2 List Scheduling (LS) for Heterogeneous Systems

The pseudo-code representation of the technique proposed to schedule diagnostic queries in heterogeneous distributed systems is given in Algorithm 7. It follows the concept of list scheduling, i.e. assign priorities to the queries and then allocate the *ready* queries to processors that give them the earliest finish time. While scheduling the queries, message scheduling is also considered. Each edge in the DMG is assigned the same priority as its destination query and is allocated to the path that gives it the least communication cost. There are four steps of the proposed scheduling heuristic: i). Adding queries to ready list, ii). Priority assignment and ordering the ready list, iii). Path and processor selection for messages and queries and, iv). Status update. Additionally, the following steps are performed before going to the query selection phase.

1. The algorithm first calculates the hyper-period $H_G$ of $G_D$ using Eq. 4.9.

2. The minimum number of times each query $q_i$ is repeated in one $H_G$ is calculated using Eq. (4.10).

3. Algorithm 2 is used to translate the history-intervals to directed edges in the graph.

4. Since each query has a different computation cost on each processor, therefore, calculating the bottom-level without simplification is not possible because it leads to multiple priorities. Therefore, average computation cost $\overline{W(q_i)}$ of each query $q_i \in Q$ is computed using Eq. (6.1). Here, $P$ represents the total number of processors in the

system.

$$\overline{W(q_i)} = \frac{\sum_{k=1}^{P} W(q_i, P_k)}{P} \tag{6.1}$$

5. Bottom-level $bl(q_i)$ of each query $q_i \in Q$ is computed using Eq. 4.18 and $\overline{W(q_i)}$, calculated in the previous step. Pseudo-code representation of this step is given in Algorithm 4. The calculated $bl(q_i)$ of each query $q_i$ is also allocated to all of its incoming edges.

### 6.2.1 Proposed Algorithm

1. ***Query Selection***: After completing the initial steps of the algorithm, the scheduler computes the ready list $R$. A query $q_i \in Q$ is said to be ready when all its parent queries have completed their execution [19]. Here, the graph obtained after translating the history-intervals to directed edges is used to determine the starting point of the child query. If the queries related to the incoming edges of the child query have completed their execution, then the child query is added to $R$. The ready time of a query $q_i \in Q$ is given in Eq. 5.1 and the ready-list $R$ is computed using Eq. 5.2. The ready-time of a query is also its release time.

2. ***Priority assignment and Ordering the Ready list***: The top-level of each query $r_i \in R$ is calculated using Eq. 4.19. The ready list $R$ is then ordered in increasing order of the top-level of each query. Also, all the incoming edges are assigned the same priorities as their destination queries because naturally, the query with higher priority should have its incoming messages scheduled first. If priorities are the same for two queries, then the tie is broken using the average bottom level of the queries computed in step 6 of the algorithm. In this case, the query that has a greater bottom-level is placed higher in the ready list. Algorithm 8 shows the pseudo-code representation for calculating top-level of a ready query $r_i$.

3. ***Path and processor selection***: After the generation of a ready list, the scheduler

**Algorithm 8** Pseudo-code representation for calculating top-level of ready query $r_i$

```
$Parent(q_i) \leftarrow$ Set of predecessor queries of $q_i$ : $\exists$ a path $p(p_j \leftarrow q_i) \in G_D$
$W(p_j) \leftarrow$ WCET of $p_j \in Parent(q_i)$ computed in Algorithm 7
$D(m_{ji}) \leftarrow$ Data transmitted between $p_j$ and $q_i$ : $\exists$ a path $p(p_j \leftarrow q_i) \in G_D$
$tl(p_j) \leftarrow$ Top-level of $p_j$
$tl(q_i) \leftarrow$ Initialize to zero, Top-level of $q_i$
for each parent query $p_j \in Parent(q_i)$ do
   $tl(q) \leftarrow tl(p_j) + D(m_{ji}) + W(p_j)$
   if $tl(q) > tl(q_i)$ then
      $tl(q_i) \leftarrow tl(q)$
   end if
end for
```

traverses through the paths and processors to select the pair that gives the least communication time to the incoming messages and least computation cost to the query, respectively. The goal of this phase is to select the combination that gives the earliest finish time to the ready query. Here, it is assumed that intra-processor communication cost is zero, i.e. if two dependent queries are scheduled on the same processor, then their data transfer cost is zero. On the other hand, if two dependent queries are scheduled on two different processors, then a message is created whose communication cost depends upon the path selected for communication.

- ***Path selection***: If a message $m_{ij}$ is being transmitted from processor $P_i$ to $P_j$, then among all the paths between $P_i$ and $P_j$, the path $path_{ij}$ that gives the least communication cost is selected. Since the links have different bandwidths, so a link must not start transmitting a message before it is fully received, i.e. completely transmitted by the previous link in the path. Consider a message $m_{ij}$ transmitting from processor $P_i$ to $P_j$ through a path $path_{ij}$ that has k number of links or hops then the earliest start time of $m_{ij}$ on a link $l_n$ in the path $path_{ij}$ is given in Eq. 5.4 and its finish time is given in Eq. 5.5. The finish time of $m_{ij}$ on $path_{ij}$ is the finish time of the message on the last link i.e. *kth* link in the path (cf. Eq. 5.6). The path between processors $P_i$ and $P_j$ that satisfies $min\{FT(m_{ij}, path_{ij})\}$ is selected for transmission of the message $m_{ij}$. Since the queries exchanging the message are periodic, therefore $m_{ij}$ also repeats its transmission in regular time intervals. The time period for the message is equal to the time period of the sending query task if it is greater than the time period of the receiving query task and vice versa. All the iterations of a message are executed on the same path. Algorithm 6 is used to make sure that only one message is transmitted over a link during a time slot. If any iteration of message $m_{ij}$ overlaps the time-slot of an iteration of message $M_{ij}$ scheduled on link $l_n$ then $m_{ij}$ cannot be scheduled on the path containing the link.

- ***Processor selection***: The scheduler traverses through all the processors of the system and selects the processor that gives the earliest finish time to the scheduling query. Another condition for the selection of the processor is that the query fulfils its relative deadline $D(q_i)$. The finish time of $q_i$ on a processor $P_j$ is given as [112]:

$$FT(q_i, P_j) = ST(q_i, P_j) + W(q_i, P_j) \tag{6.2}$$

Here, $ST(q_i, P_j)$ is the start time of $q_i$ on $P_j$ and $W(q_i, P_j)$ is the WCET of $q_i$ on $P_j$. So the processor that gives the minimum finish time and on which the query $q_i$ fulfills its deadline, i.e., $FT(q_i, P_j) < ST(q_i, P_j) + D(q_i)$, is selected. A query $q_i$ should not start its execution until all of its required data is transmitted to the allocated processor. Therefore, the earliest start time of a query $q_i$ on processor $P_j$ can be given as,

$$EST(q_i, P_j) = max\{DRT(q_i, P_j), EAT(P_j)\} \tag{6.3}$$

where, $DRT(q_i, P_j)$ is the data ready time of $q_i$ on $P_j$ i.e. the time at which all the data required for $q_i$ is transmitted to $P_j$ and $EAT(P_j)$ is the earliest available time of $P_j$. Also Algorithm 5 is used to ensure that only one query is executed during a time slot in the schedule. If any iteration of a query $q_j$ scheduled on $P_j$ lies within the time-slot allocated to an iteration of $q_i$ or vice versa then $q_i$ cannot be scheduled on $P_j$.

4. ***Status Update***: Once the query is scheduled on a processor, the status of the system is updated. As $q_i$ is now scheduled on $P_j$ so its WCET is no longer unknown. Hence the weight of the vertex that identifies $q_i$ in the graph is set to its WCET on $P_j$. Moreover, the weight of the edges between $q_i$ and all of its parents that were scheduled on $P_j$ are set to zero. The change in the weight of edges changes the top-level of $q_i$ so it is recalculated. Since $tl(q_i)$ affects the top-level of all the child queries of $q_i$ therefore it is recalculated to keep the critical path accurate at each step of the schedule.

The above mentioned steps are repeated until a valid schedule is obtained.

### 6.2.2 Example

Consider the DMG given in Fig. 6.2a. It needs to be scheduled on the heterogeneous distributed system given in Fig. 6.3 that has two bi-directional communication paths from $P_1$ to $P_2$ i.e. $< l_1, l_2, l_3 >$ and $< l_1, l_4, l_3 >$. The corresponding computation cost matrix $W$ is given in Table 6.2. Each $w_{ij}$ in the computation matrix is in millisecond. Similarly the time periods of the queries are also in milliseconds and given in Fig. 6.2a. According to the algorithm, the first step is to calculate the hyper-period of the DMG using Eq. 4.9 i.e. 6 ms. The algorithm then calculates the minimum number of times each query is repeated in one cyclic execution of the DMG using Eq. 4.10 i.e. $times(q_1) = times(q_4) = 3$, $times(q_2) = times(q_3) = 2$, and $times(q_5) = 1$. After that the history-intervals in the DMG are translated into directed edges as shown in Fig. 6.2b. The average computation cost and the average bottom-level calculated for the example are given in Table 6.3. At *0 ms*, only $q_1$ and $q_2$ are ready since they have no precedence constraints. Both the queries have top-levels equal to zero so average bottom-level is used to break the tie. According to Table 6.3, $q_1$ has a higher priority than $q_2$ so it is scheduled first. Processor $P_1$ gives the earliest finish-time to $q_1$ *0.5 ms* therefore $q_1$ is scheduled on $P_1$. Query $q_2$ ends at *1.5 ms* for both $P_1$ and $P_2$ but it hinders the execution of $q_1$ on $P_1$ so it is assigned to $P_2$. Since the queries are allocated to the processors so their WCETs are updated in the DMG. According to Fig. 6.2b, query $q_3$ requires a single iteration of $q_1$ so it is ready for execution at *0.5 ms*. Its top-level is calculated as *0.5 + 1 = 1.5*. The finish time for $q_3$ on $P_1$ is *1 ms* but it hinders the execution of already scheduled $q_1$. Therefore, $q_3$ is allocated to $P_2$. The finish time for $m_{12}$ on $< l_1, l_2, l_3 >$ is *3 ms* but it is *2.5 ms* on $< l_1, l_4, l_3 >$ therefore the latter path is selected for the transmission of $m_{12}$. If $q_3$ starts execution at *2.5 ms* on $P_2$, it overlaps the execution of $q_2$. Therefore, $q_3$ is allocated the next available time slot on $P_2$ at *4.5 ms*. Query $q_4$ is ready at *5.5 ms* and is assigned to $P_1$ since it cannot be scheduled on $P_2$. Similar to $m_{12}$, path $< l_2, l_4, l_1 >$ is selected for the transmission of $m_{34}$ since it provides faster communication service than the other path. Lastly, query $q_5$ is ready at *10 ms* and is assigned to $P_2$ since it cannot be scheduled on $P_1$. Message $m_{45}$ cannot be assigned to path $< l_1, l_4, l_2 >$ because it hinders the transmission of $m_{13}$ therefore it is assigned to $< l_1, l_3, l_2 >$. Query $q_5$ does not start transmission on $P_2$ at *12.5 ms* because it overlaps both $q_2$ and $q_3$. Therefore it is assigned to the next possible time slot at *14.5 ms*. The schedule length for one cyclic execution of given $G_D$ is *15 ms*. The timeline representation of the schedule is given in Fig. 6.4. Here, *path$_{12}$* is $< l_1, l_4, l_2 >$, *path$_{21}$* is $< l_2, l_4, l_1 >$, and finally *path'$_{12}$* is $< l_1, l_3, l_2 >$.

| **Processors** / **Queries** | **P$_1$** | **P$_2$** |
|---|---|---|
| **q$_1$** | 0.5 | 1 |
| **q$_2$** | 1 | 1.5 |
| **q$_3$** | 0.5 | 1 |
| **q$_4$** | 0.5 | 1 |
| **q$_5$** | 0.25 | 0.5 |

Table 6.2: Computation Cost Matrix $W$ for $G_D$ given in Fig. 6.2a implemented on architecture given in Fig. 6.3

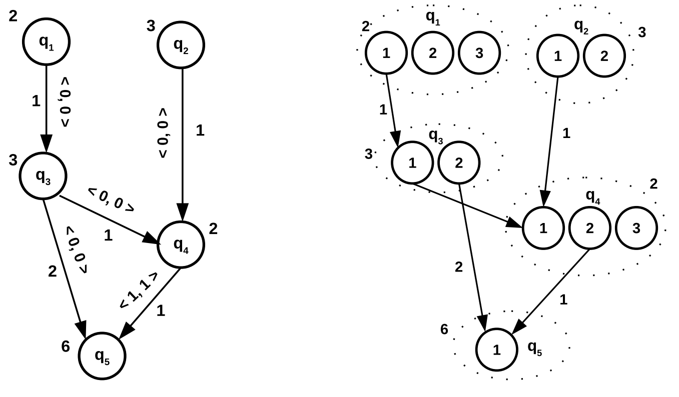

(a) (b)

Figure 6.2: (a). DMG - $G_D$ (here D($q_i$) = T($q_i$)) (b). $G_D$ after translating the history-interval to directed edges (here D($q_i$) = T($q_i$))

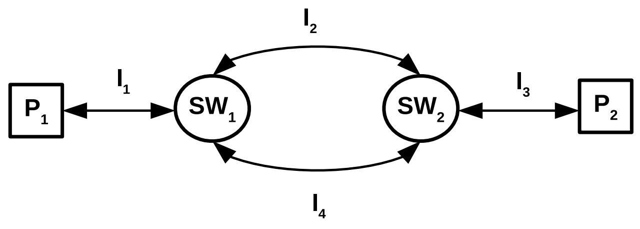


Figure 6.3: Heterogeneous distributed system ($l_1 = l_2 = 1$ Mb/ms, $l_3 = l_4 = 2$ Mb/ms)

| Features / Queries | $\overline{W(q_i)}$ | $bl(q_i)$ | $tl(q_i)$ | $W(q_i)$ |
|---|---|---|---|---|
| **$q_1$** | 0.75 | 6.5 | 0 | 0.5 |
| **$q_2$** | 1.25 | 5.25 | 0 | 1.5 |
| **$q_3$** | 0.75 | 4.75 | 1.5 | 1 |
| **$q_4$** | 0.75 | 3 | 2.5 | 0.5 |
| **$q_5$** | 1.25 | 1.25 | 3.5 | 0.5 |

Table 6.3: Features of the queries calculated for scheduling

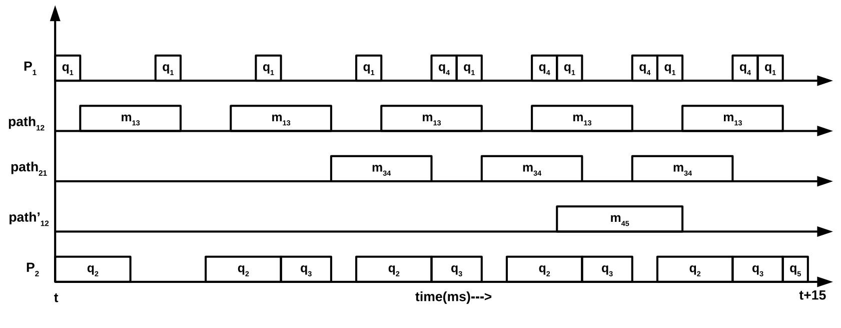


Figure 6.4: Schedule representation for $G_D$ given in Fig. 6.2a implemented on architecture given in Fig. 6.3 (t = 0 ms)

### 6.2.3 Complexity of LS algorithm for heterogeneous systems

In the LS algorithm for heterogeneous distributed systems, the time-complexity to calculate the bottom and top levels of queries using the depth-first search (DFS) is $O(|Q| + |D||E|)$ [129]. Sorting the queries according to their priorities has a complexity of $O(|Q|log|Q|)$. Calculating the communication time for each message has a complexity of $O((|Q| + |D||E|)|P||L|)$ and assigning the best possible link has $O(|L|log|L|)$ time complexity. Lastly assigning the query to the processor that gives its earliest finish time has complexities of $O(|Q| + |D||E|)|P|$ and $O(|P|log|P|)$. It shows that the complexity of the scheduler depends greatly upon the size of DMG and the structure of system architecture. The highest complexity is for the last two steps of the scheduler. It can be reduced by restricting the algorithm to consider the first free link or the first available processor instead of searching for the best possible solution. This restriction, however, might increase the schedule length of the graph.

## 6.3 Experimental Setup

This section presents the results for the various experiments performed to test the validity of the proposed algorithm. The experiments are performed on a platform with four Intel Core i5-6200U cores operating at a fixed clock frequency of 2.30 GHz and 23 GB RAM. The operating system used is fedora 30 with kernel Linux 5-generic. The algorithm is programmed in C language and implemented on an online-server. A set of randomly generated graphs are used to test the scheduler, and the results are compared through varying parameters. The following characteristics are used to generate these test graphs.

1. The total number of queries in the DMG represented by $|Q|$.

2. The number of directed edges per query in the DMG represented by $|M|$.

3. Time period $T(q)$ of a query $q \in Q$. It is randomly generated from the arithmetic

sequence $\{T_{min}(q), T_{min}(q)+1, ..., T_{max}(q)\}$ such that the average time period $\overline{T(q)}$ of the DMG is always equal to $\frac{T_{min}(q)+T_{max}(q)}{2}$.

4. Relative deadline $D(q)$ of a query $q \in Q$. It is equal to the time period of the query, i.e., $T(q)$.

5. History interval $< a_{ij}, b_{ij} >$ between queries $q_i$ and $q_j$, where $q_i$ is the parent of $q_j$, is generated randomly from an arithmetic sequence of integers such that $a_{ij} \in \mathbb{Z}_{\geq 0}$ and $b_{ij} \in \mathbb{Z}_{b_{ij} \geq 0 \cup b_{ij} < a_{ij}}$.

6. Average utilization factor $\overline{UT(q)}$, i.e. the ratio of average WCET of a query $q \in Q$ to its time period $T(q)$, is used to calculate the $\overline{W(q)}$ of the query. This factor is chosen such that the time period is always greater than the maximum WCET of a query.

7. Computation cost heterogeneity factor $\alpha$ is used in most scheduling algorithms [130, 131] to introduce heterogeneity to the system. If $\alpha = 0$ then the WCET of a query is the same for all the processors. If there are $P$ processors in a distributed system and the average WCET of a query is $\overline{W(q)}$ then the WCET of $q$ on a processor $P_i \in P$ lies between $[\overline{W(q)} * (1 - \frac{\alpha}{2}), \overline{W(q)} * (1 + \frac{\alpha}{2})]$ [130, 131]. This range is used to calculate $P$ different WCETs of a query $q$, and then the costs are arranged in ascending order, i.e. the lowest cost is assigned to $P_1$ and so on. The value of $\alpha$ is between $(0, \frac{2-2\overline{UT(q)}}{\overline{UT(q)}})$ so that the WCET of a query at any processor is not greater than its time period.

8. Eq. 4.23 is used to calculate the amount of data transmission between queries. Here, the average WCET of a query is used to keep the data transmission same between queries for all processors in the distributed system.

The mentioned characteristics are used to generate 300 DMG applications. The application size $|Q|$ ranges from 200 to 600 with an increment of 200 queries per DMG. The number of directed edges per query is four. The time period is taken from the arithmetic sequence $\{1, 2, 3, ..., 10\}$ measured in milliseconds and $\overline{UT}$ is 0.25. Therefore, the range for $\alpha$ is (0, 6) and is taken as one for the experiments. Thus the WCET of $q$ lies between $[0.5\overline{W(q)}, 1.5\overline{W(q)}]$ where the range of $\overline{W(q)}$ is from 0.25 to 2.5 ms. The history-interval $< a_{ij}, b_{ij} >$ is chosen from the arithmetic range $\{0, 1, 2, ..., 10\}$ such that $b_{ij} \leq a_{ij}$. The *CCR* value varies between low and medium communication loads and is either 0.1, 0.5 or 1. The amount of data transmitted is given in megabits.

Different system architectures consisting of distributed clusters are generated for the implementation of the scheduler. To keep things simpler, a distributed cluster in the system consists of only homogeneous processors; therefore, heterogeneity factor $\alpha$ is zero for each cluster. These nodes are combined through a defined network topology to form a heterogeneous distributed system. Following characteristics are used to generate these architectures.

1. Total number of processors used in the distributed system, $P$.

2. Network topology used in each cluster and the overall system.

3. Total number of processors $P_C$ and the total number of switches $SW$ in a cluster. The total number of switches here usually depend upon the topology used in the node.

4. Total number of links per cluster represented by $L$.

5. Rate of transmission $w_k$ of a link $l_k$ in the system. It is randomly generated from the arithmetic sequence $\{w_{min}, w_{min} + \beta, ..., w_{max}\}$ such that the overall bandwidth of the cluster is always equal to $\frac{w_{max}+w_{min}}{2}$. $\beta$ is the common difference between two consecutive numbers of the sequence.

For the system architecture, the total number of processors is varied between 16, 32 and 64. There are four processors in each cluster, so the total number of distributed clusters are 4, 8 or 16. Each cluster uses a star topology with one switch and one bi-directional link between processors and the switch. The bandwidths of the links are generated from the arithmetic sequence $\{40, 60, 80, 100\}$. The rates are given in Mbps. For the overall system, two topologies are used i). ring topology and ii). bus topology. The homogeneous clusters are combined using these topologies to form a heterogeneous distributed system.

The metrics used to evaluate the results are scheduling length (cf. Eq. 4.20) and scheduling rate of the DMGs, i.e., the percentage of DMGs successfully scheduled in a set (cf. Eq. 5.7). These metrics were measured against the total number of queries per DMG, communication to computation cost ratio, the total number of processors in the distributed system, and the network topology of the overall distributed system.

## 6.4 Results

This section presents an analysis of the results obtained from the experiments performed using the setup described before. In the first set of experiments, the trend in the scheduling length was observed with an increase in DMG's size. The scheduling length increases with an increase in the total number of queries. Consider the bar graph shown in Fig. 6.5. Here, 100 graphs with application sizes 200, 400 and 600 are scheduled on a 64-bus networked distributed system. Communication to computation cost ratio *CCR*, is 0.5. Each bar in the graph represents the minimum, standard deviation, average and the maximum values of the scheduling lengths. Here, the average scheduling length for $Q = 200$ is approximately 38.9 ms, whereas it increases to approximately 168.9 ms when the total number of queries is increased to 600. The size of the DMGs does not considerably affect the scheduling rate if there are enough resources in the distributed system.

In the second set of experiments, the results were evaluated against the increase in the communication load on the network, i.e., an increase in CCR values from low to medium. An increase in *CCR* increases the amount of data being transmitted from a parent query to its child query. Which in turn increases the congestion over the network as each message now occupies more bandwidth of the link. For example, an edge transmitting data of 4 units over a link of the rate of four units/ms takes 1 ms to complete its transmission. Now if the data is doubled to 8 units, the same edge occupies the link for the double amount of time.

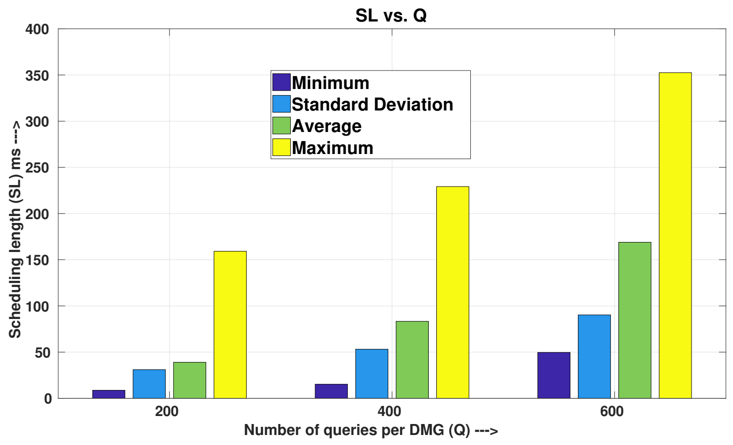


Figure 6.5: 64-bus, CCR = 0.5, $\alpha$ = 1

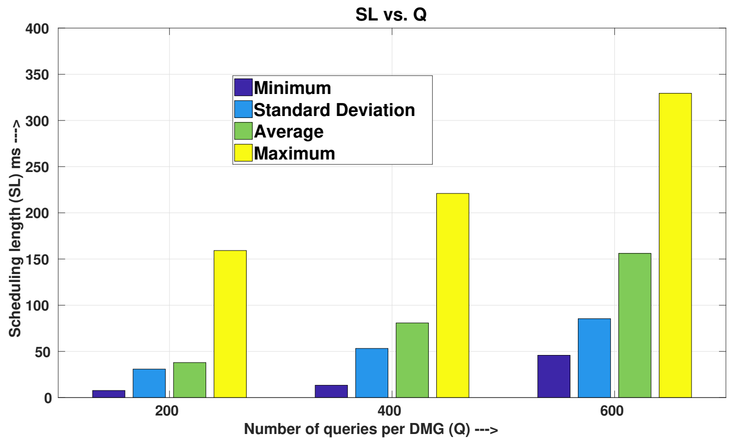


Figure 6.6: 64-ring, CCR = 0.5, $\alpha$ = 1

Consider the graph in Fig. 6.7. Here 100 separate instances of $Q = 200$ queries per DMG have been scheduled on a 32-ring distributed network. The *CCR* varies between 0.1, 0.5 and 1.0. Fig. 6.7 shows that when the communication load increases from low to medium, the scheduling length is increased. It was also noticed from the results that increasing this *CCR* value often results in a decrease in the scheduling rate over the same network. It is explainable in the sense that a message which was previously scheduled over a link might now hinder other messages on that link with its increased duration.

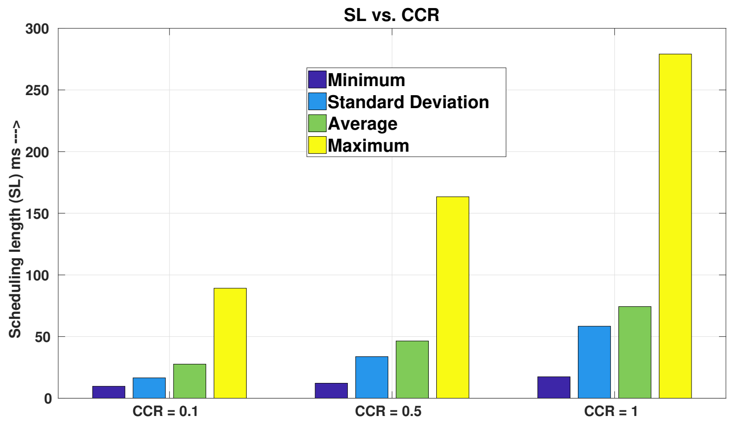


Figure 6.7: Variation in SL with an increase in CCR

The next set of results evaluate the results when the number of resources in the system is increased. The increase in the number of processors increases the parallel execution of queries which in turn reduces the overall schedule length of the DMG. Consider Fig. 6.8, where 100 instances of a DMG containing 400 queries is scheduled on 16-, 32- and 64-ring distributed systems. When a DMG is scheduled on a 64 processor distributed system, the average scheduling length is considerably lower than when it is scheduled on a 16 processor distributed system. An interesting point here is that after some time the scheduling length starts converging to the same result irrespective of the number of resources being used. The parallelization of the graph plays a role in this scenario. For instance, consider a graph consisting of three queries that are possible to execute in parallel with each other. Now if the number of processors is two, then two of the queries can run in parallel while the last query has to wait for the other queries to complete execution. But if the number of resources is increased to three, then the queries can easily execute in parallel. Now even if the processors are increased to five, the graph still utilizes three processors so it will give the same schedule length as in the previous case. The same principle applies in case of scheduling rate of the graph. If there are more resources, then there is more chance of obtaining a scheduling result.

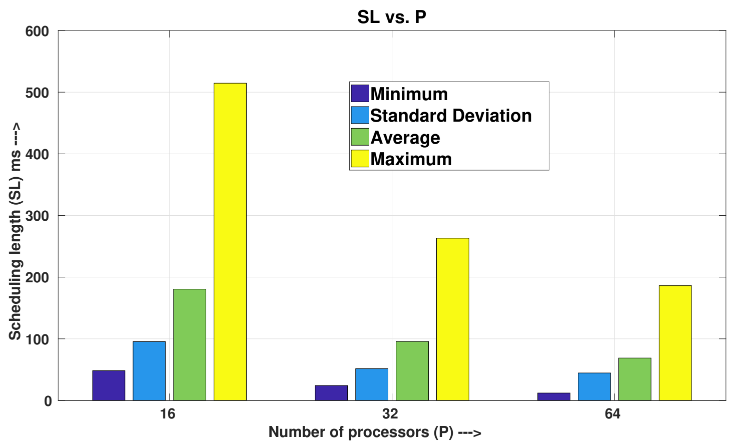


Figure 6.8: Variation in SL with an increase in *P*

In the last set of experiments, the results are evaluated for different network topologies in the system. Fig. 6.6 shows the ring based implementation of the experiment performed in Fig. 6.5. The results show that there is not much difference in the scheduling length of the graphs. However, the network topology has an impact on the scheduling rate. For example, for $Q = 600$, the scheduling rate for the 64-ring network is approximately 86%, but it reduces to 75% when the same scenario is applied on a 64-bus distributed network. It might be because there are more link resources in a ring network which increases the chances of successful message allocations. Increase in the link resources better exploits the parallel execution of queries. Therefore, an increase in the number of link resources increases the scheduling rate of the graph.

# CHAPTER 7
# INCREMENTAL LIST SCHEDULING

THIS CHAPTER presents an incremental list scheduling heuristic to schedule diagnostic services in a homogeneous open distributed real-time environment where changes are made to the system during runtime. An open distributed real-time embedded (ODRE) system has requirements for reliable operations with strict timing constraints and open world assumptions. If the failure rate of the electrical components is considered, then one way to obtain dependability in such a system is through fault tolerance. This thesis supports fault-tolerance in ODRE systems using a Diagnostic Multi-query Graph (DMG). A DMG is a directed graph of diagnostic queries, that is used to detect and diagnose faults in ODRE systems. It is scheduled before being implemented to depict its temporal behaviour and also to bound the time to detect and diagnose faults. An open-world assumption in ODRE system means that the system components can be changed, i.e., new electrical components can be added, or old ones can be removed from the system, during runtime to realise global services. Whenever there are changes in the system, the diagnostic graph is also changed. For example, consider a sensor whose output is being used by a set of diagnostic queries to diagnose faults. If this sensor is replaced with another sensor of the same type but different characteristics then the characteristics of the diagnostic queries getting data from this sensor are also changed. Whenever there are changes in the system the schedule is recomputed to integrate these changes. The system then switches to this new schedule and the execution of the diagnostic services proceed according to this schedule. Since ODRE systems have stringent timing constraints and the changes are made at runtime therefore the scheduling algorithm should be fast in recomputing a feasible schedule so that the changes can be integrated before the system becomes unstable. An efficient way to do is to only schedule the changes in the system while not touching the already scheduled diagnostic queries and communication messages. This reduces the complexity of the scheduling algorithm because it takes less time to find a schedule for only the changes rather than the whole diagnostic graph. In an ideal situation, the algorithm manages to find a feasible schedule for the changes without touching the already scheduled diagnostic queries and communication messages. However, in real-world applications, it might be necessary to modify resource allocations of some of the already scheduled diagnostic queries and communication messages to obtain a feasible schedule. In this case, the modifications must be minimised for the stability of prior applications and to maximize the continuity of services. If a feasible schedule for the changes is unattainable even after modifications in the old schedule then the designer is notified that it is not possible to make the requested changes in the system. In this scenario, the system discards the changes and keeps running the old schedule. This chapter proposes an incremental list scheduling heuristic that computes a schedule for the changes in a homogeneous distributed system while minimising the modifications to the already scheduled diagnostic queries and communication messages. List scheduling is used because its time complexity is pseudo-polynomial, and it can be implemented during runtime. It has to be noted that the proposed algorithm can be used to schedule any application on a open distributed real-time embedded system and the scheduling of diagnostic services to support fault-tolerance in ODRE systems is just taken as a case study here.

## 7.1 Problem Formulation

Given a homogeneously distributed system architecture presented in Section 4.1 and a diagnostic application presented in Section 4.2.2 $G_D$: construct a feasible static cyclic schedule for the time-triggered execution of modified queries and the time-triggered execution of modified messages in $G_D$. Whenever changes occur in the system architecture or the system application, the scheduler is activated, and it computes the schedule of the queries and messages effected by the change while minimising the modifications to the already computed schedule.

For supporting the open-world assumption and fault-tolerance in time-triggered ODRE systems, a schedule must be generated at runtime whenever new functionality is added to the system. After the computation of the schedule it is deployed and the message and diagnostic query dispatching continues. When the system is initialised, the scheduler makes an execution and communication plan for the queries/messages that serves as the primary schedule for the system. Whenever, there are changes in the system architecture or the system application, e.g., removal of a processor or addition of a new task, the scheduler is reactivated at runtime to recompute the schedule. The changes in the already computed schedule need to be kept minimum for the stability of prior applications. If a feasible schedule for the changes is not obtained, then the requested changes can not be added to the system. In this case, the scheduler discards the changes and keeps running the old schedule.

The scheduler starts scheduling the new queries and messages using the primary schedule as the starting point. Each scheduling decision for a new query, i.e. the processor selected for its execution and the paths chosen for the transmission of its incoming messages, is termed as a 'scheduling step'. After a 'schedule step' is made, it is analysed to determine whether the query fulfils its deadline with the selected combination of processor and paths. If the query cannot complete execution before its deadline, then the corresponding schedule step is discarded, and a new schedule step is generated. This process is repeated until the query is allocated to a processor that fulfils its timing constraints. The final schedule step for the query is integrated into the overall schedule that is a combination of the old schedule and the newly generated schedule steps. The process of making a scheduling decision and keeping or discarding the schedule step fits well into the list scheduling approach. There is no limit to the number of changes, or the type of changes occurring in the system and any change can occur in the system architecture and diagnostic application.

**Algorithm 9** Pseudo-code representation of ILS to schedule changes in the diagnostic multi-query graph in a homogeneous distributed embedded system

```
signal ← external signal that notifies the occurrence of a change in the system to the scheduler
if system initialised then
    Use Algorithm 3 to compute a schedule for the DMG.
    Store the schedule of tasks in schedule table.
    Store the schedule of messages in message table.
    Store the characteristics of the system architecture.
    Store the characteristics of the DMG.
end if
if signal then
    Compare the characteristics of the old system architecture and the new system architecture to identify changes.
    Compare the characteristics of the old DMG to the characteristics of the new DMG to identify changes.
    Store the new DMG and the new system architecture.
    Compute the set of queries $U$ affected by the change.
    Calculate $bl(u_i)$ of each query $u_i \in U$.
    Order $U$ according to the bottom-level of the queries. Ties are broken by prioritising the query whose children have a
    greater bottom-level.
    Remove all the queries in $U$ from the schedule table.
    Remove all the messages that are transmitting data to queries in $U$ from the message table.
    for each query $u_i \in U$ do
        Use Algorithm 3 to schedule the query $u_i$ on a processor $P_j$ at its ready-time $r_t(u_i)$ calculated using Eq. 7.2.
        if $u_i$ is not schedulable at calculated $r_t(u_i)$ then
            Calculate the earliest ready-time of $u_i$ using Eq. 5.1.
            Use Algorithm 3 to schedule the query $u_i$ on a processor $P_j$ that gives it the earliest start time and on which $u_i$ fulfils
            its deadline.
            Compute the set of queries $H$ that hinder the execution of $u_i$ at the selected processor $P_j$ using Algorithm 11 and
            Algorithm 12.
            if all the queries in $H$ have a greater bottom-level or an earlier deadline than $u_i$ then
                Query $u_i$ cannot be scheduled on $P_j$ during the selected time slot so the algorithm schedules it on the next
                processor that gives it the earliest start time and on which $u_i$ fulfils its deadline provided that $u_i$ is schedule on
                said processor.
            else if all the queries in $H$ have a lower bottom-level or a later deadline than $u_i$ then
                All the queries in $H$ along with their child queries are removed from the schedule table and added to $U$.
                All such messages that are transmitting data to queries in $H$ and their child queries are removed from message
                table.
                Query $u_i$ is scheduled on $P_j$.
                Update the schedule and message table.
            end if
        end if
    end for
    if a feasible schedule is not obtained then
        Discard the requested changes and continue running the old schedule.
    end if
end if
```

## 7.2 Incremental List Scheduling (ILS)

An iterative or incremental scheduling algorithm was first used by authors in [50] to incorporate incremental design in hard real-time distributed embedded systems. The authors give a more detailed version of the process in [132] where they describe the incremental scheduling algorithm as a continuous iterative process. When the system is first initialised, the algorithm generates a primary schedule based on the initial system requirements and constraints. When the system encounters a change in its architecture or system application, the algorithm analyses the change and schedules the new tasks and messages. The resultant schedule is analysed to ensure that all tasks fulfil their deadline. The process is repeated until a schedule is obtained that follows all the stringent timing constraints of the system.

This thesis proposes an Incremental List Scheduling (ILS) algorithm to schedule modifications in a DMG to support diagnostic services in Open Distributed Real-Time Embedded (ODRE) systems. When the system is first initialised, a schedule is generated using Algorithm 3. This schedule is termed as the primary schedule. Whenever a change occurs in the system, a signal is generated to notify the scheduler. The ILS algorithm then identifies the change in the system architecture or the DMG and computes the queries affected by said change. Then the algorithm schedules both the affected queries and the new queries onto the target system while minimising the changes to the already scheduled queries and messages. There are three main steps of ILS: (i). Identify the changes in the system architecture or the system application, (ii). Identify the queries affected by the change and (iii). Schedule the affected queries and new queries/messages on the system. The pseudo-code representation of ILS is given in Algorithm 9.

### 7.2.1 Identify the modifications in the system

To simplify the algorithm, ILS considers only one modification at a time, be it a change in the system architecture or the system application. The characteristics of the old system

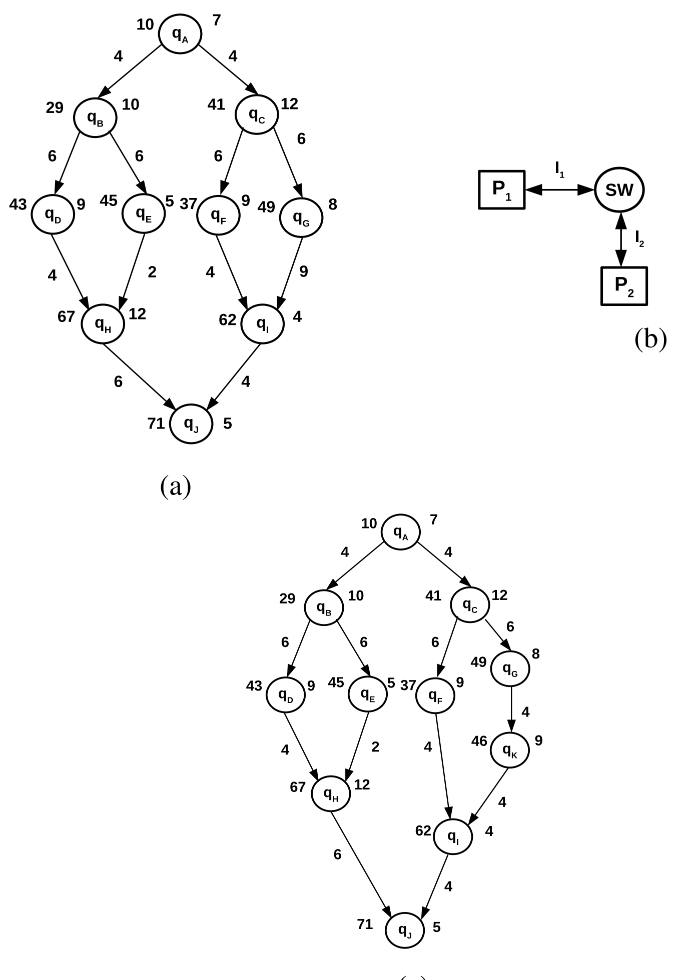


Figure 7.1: (a). DMG - $G_D$ ($T(q_i)$ = 100 ms, $< a_{ij}, b_{ij} >$ = $< 0, 0 >$) (b). Homogeneous distributed system ($l_1$ = 2 Mb/ms, $l_2$ = 1 Mb/ms) (c). Modified version of $G_D$ ($T(q_i)$ = 100 ms, $< a_{ij}, b_{ij} >$ = $< 0, 0 >$)

architecture and DMG are compared with the characteristics of the modified architecture and DMG. The algorithm assigns a unique ID to each component in the system architecture, and a unique one to each query in the DMG. These IDs once assigned, remain unchanged. The ILS uses these IDs to identify changes in the system. For example, if a particular ID present in the old architecture is not present in the new one, then it means that the designer has removed the corresponding component from the system architecture. Similarly, a query with a particular ID might have a different WCET or deadline in the new DMG and would require rescheduling. If a completely new DMG is added to the system, then the old and

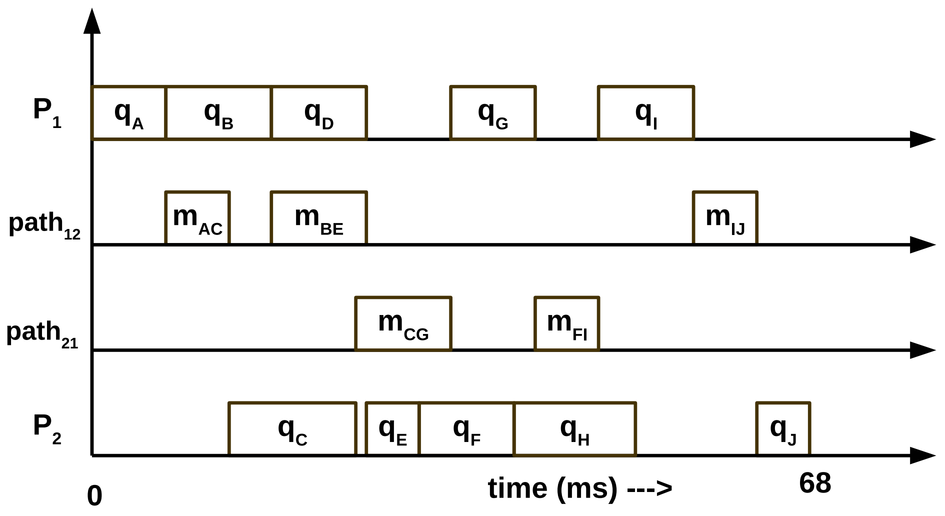


Figure 7.2: Schedule representation for $G_S$ given in Fig. 7.1a implemented on architecture given in Fig. 7.1b

new DMGs would have no common IDs. In this scenario, dummy source and sink vertices with zero WCETs are added to combine the two DMGs. The algorithm adds the dummy source vertex as a predecessor to all the queries in the DMGs that have no predecessors and the sink vertex as a successor to all the queries that have no successors. To understand this phase, consider the two DMGs given in Fig. 7.1a and Fig. 7.1c. When we compare the two DMGs, the result shows that the designer has added a new query $q_K$ between queries $q_G$ and $q_I$.

### 7.2.2 Compute the affected queries

After identifying the changes in the system, the next step is to identify queries in the DMG affected by the change. Let $U$, called rescheduling set, be a set of unscheduled queries affected by the change. It can be represented as

$$U = q_i \in Q : q_i \text{ is unscheduled in the old schedule or needs rescheduling} \tag{7.1}$$

The rescheduling set $U$ contains all such queries that were either affected by the change (their WCET or the WCET of one of their predecessors was changed) or the queries that

Table 7.1: *Schedule Table* for Fig. 7.2

| **Task** | **Start Time (ms)** | **End Time (ms)** | **Assigned Processor** |
|---|---|---|---|
| $q_A$ | 0 | 7 | $P_1$ |
| $q_B$ | 7 | 17 | $P_1$ |
| $q_C$ | 13 | 25 | $P_2$ |
| $q_D$ | 17 | 26 | $P_1$ |
| $q_E$ | 26 | 31 | $P_2$ |
| $q_F$ | 31 | 40 | $P_2$ |
| $q_G$ | 34 | 42 | $P_1$ |
| $q_H$ | 40 | 52 | $P_2$ |
| $q_I$ | 48 | 57 | $P_1$ |
| $q_J$ | 63 | 68 | $P_2$ |

Table 7.2: *Message Table* for Fig. 7.2

| **Message** | **Start Time (ms)** | **End Time (ms)** | **Sender** | **Receiver** |
|---|---|---|---|---|
| $m_{AC}$ | 7 | 13 | $P_1$ | $P_2$ |
| $m_{BE}$ | 17 | 26 | $P_1$ | $P_2$ |
| $m_{CG}$ | 25 | 34 | $P_1$ | $P_2$ |
| $m_{DH}$ | 31 | 37 | $P_1$ | $P_2$ |
| $m_{FI}$ | 42 | 48 | $P_2$ | $P_1$ |
| $m_{IJ}$ | 57 | 63 | $P_1$ | $P_2$ |

were added to the DMG corresponding a change in the system (the addition of a new sensor). For example, if the designer removes a processor from the system architecture, then all the queries that were previously allocated to the said processor cannot execute if they are not assigned to another processor. In this case, all the queries allocated to the removed processor are added to the rescheduling set $U$. Consider another example, where the designer adds a set of new queries in the DMG. All the queries in this set need scheduling and therefore, are added to $U$.

The queries in the DMG follow a precedence constraint so a successor query cannot execute until all of its predecessors have completed their execution. Therefore, the removal of a query from the schedule also affects the scheduling of its successor queries. For computing all the queries affected by the change, the algorithm needs to find the paths in the DMG that contain the queries from the rescheduling set $U$. The pseudo-code representation of the algorithm used to determine the paths is given in Algorithm 10 that was previously used by authors in [114]. The algorithm uses a recursive function to find all the paths in the graph that start with the queries present in $U$. The function is called for all such queries that are successors of the considered query $u \in U$. If a successor query has no outgoing messages, then it is the last query in the path $p$. The computed path $p$ is then added to a path set $P$. All the queries in this set are then added to $U$. As a final step, queries in $U$ are removed from the *schedule table* and the messages that are transmitting data to queries in $U$ are removed from the *message table*. For example, the scheduling set $U$ for the modified DMG given in Fig. 7.1c contains queries $< q_K, q_I, q_J >$. The updated *schedule* and *message tables* for this example are given in Table 7.3 and 7.4.

**Algorithm 10** Pseudo-code representation for computing the paths in $G_D$ that start with queries in $U$

```
$u \leftarrow$ unscheduled query $\in U$
$p \leftarrow \emptyset$ , current path
$P \leftarrow$ Set of all paths in $G_D$ with source query $u$
$Children(u) \leftarrow$ Set of all successor queries of u : $\exists$ a path $p(u- > c_j) \in G_D$
**function** ComputePaths (u, p, P)
$p \leftarrow p \cup u$
**for** each task $c_i \in Children(u)$ **do**
    **if** $c_i$ is the last successor **then**
        ComputePaths($c_i$, p, P)
    **else**
        $tempP \leftarrow p \setminus c_i$, create a new path
        $P \leftarrow P \cup p$
        ComputePaths($c_i$, tempP, P)
    **end if**
**end for**
**return**
```

### 7.2.3 Scheduling the queries/messages

After the computation of the scheduling set $U$, the next step is to schedule the queries in $U$ and their corresponding communication messages. The queries and messages are scheduled incrementally from the end of the primary schedule. There are two scenarios here: (i). A query scheduled at the end of the primary schedule fulfils its deadline and (ii). It does not fulfil its absolute deadline in which case changes need to be made in the primary

Table 7.3: Updated *Schedule Table* after modification of Fig. 7.1a to Fig. 7.1c

| **Task** | **Start Time (ms)** | **End Time (ms)** | **Assigned Processor** |
|---|---|---|---|
| $q_A$ | 0 | 7 | $P_1$ |
| $q_B$ | 7 | 17 | $P_1$ |
| $q_C$ | 13 | 25 | $P_2$ |
| $q_D$ | 17 | 26 | $P_1$ |
| $q_E$ | 26 | 31 | $P_2$ |
| $q_F$ | 31 | 40 | $P_2$ |
| $q_G$ | 34 | 42 | $P_1$ |
| $q_H$ | 40 | 52 | $P_2$ |

Table 7.4: Updated *Message Table* after modification of Fig. 7.1a to Fig. 7.1c

| **Message** | **Start Time (ms)** | **End Time (ms)** | **Sender** | **Receiver** |
|---|---|---|---|---|
| $m_{AC}$ | 7 | 13 | $P_1$ | $P_2$ |
| $m_{BE}$ | 17 | 26 | $P_1$ | $P_2$ |
| $m_{CG}$ | 25 | 34 | $P_1$ | $P_2$ |
| $m_{DH}$ | 31 | 37 | $P_1$ | $P_2$ |

schedule to accommodate the query.

ILS first uses the method given in Algorithm 3 to schedule the queries in $U$ at the end of the original schedule. It means that the ready time for the queries in $U$ is the schedule length of the original schedule. Although there might be a case where the parent queries of a query $u_i \in U$ are not yet scheduled in which case the maximum finish time among the parent queries of $u_i$ is its ready time. Thus, the ready time of a query $u_i$ in $U$ is given as,

$$r_t(u_i) = max\{max_{u \in pred(u_i)} f_t(u), max_{q_i \in schedule\ table} f_t(q_i)\} : \exists \text{ a path } p(u- > u_i) \in G_D \tag{7.2}$$

where $pred(u_i)$ is the set of predecessors of $u_i \in G_D$, $f_t(u)$ is the finish time of query $u \in pred(u_i)$ and $f_t(q_i)$ is the finish time of a query $q_i$ in the *schedule table*. If a query $u_i \in U$ does not fulfil its deadline on any processor in the system with its calculated ready time, then its earliest ready time is calculated using Eq. 5.1. ILS then again uses Algorithm 3 to schedule $u_i$ at this ready time. However, there is a possibility that the processor chosen by the algorithm might be assigned to another query $q_i$ in the schedule. In this scenario, if $u_i$ has a greater bottom-level or earlier deadline than $q_i$ then it is assigned to the chosen processor otherwise the algorithm assigns $u_i$ to the next processor that gives it the earliest start time and at which $u_i$ fulfils its deadline. There is also a possibility that although $u_i$ is schedulable on the processor in the time slot computed using its earliest ready time. Still, one of its incoming messages overlaps a message in the *message table*. If this is the case, then the bottom-level and deadline of $u_i$ are compared with the source query $q_i$ of the conflicting message and the query with a higher bottom-level or an earlier deadline is given the priority. In both possibilities, if query $q_i$ has a lower priority or later deadline, then it along with all of its successors and related messages are removed from the schedule. All the removed queries are also added to $U$.

On the other hand, if $u_i$ has a lower priority or later deadline, then ILS tries to schedule it in the next available time slot or another processor. Pseudo-code representation of the algorithms used to compute the queries hindering the execution of $q_i$ are given in Algorithms 11 and Algorithm 12. After each successful allocation of a query in $U$, the *schedule* and *message tables* are accordingly updated to keep the ready times for other queries concurrent. The process is repeated until the queries in $U$ are scheduled, and a schedule is obtained that satisfies the stringent timing constraints of the system. The queries in $U$ are scheduled following their bottom-level where a query with a greater bottom-level is scheduled first. Ties are broken by prioritising the query whose children have a greater bottom-level.

To understand this phase, consider the DMG is given in Fig. 7.1a that is scheduled using Algorithm 3 on the homogeneous distributed system given in Fig. 7.1b. The time-line representation of the primary schedule is given in Fig. 7.2. The *schedule* and *message table* of the primary schedule are given in Table 7.1 and Table 7.2, respectively. The designer modifies the original DMG by adding a query $q_k$ between queries $q_G$ and $q_I$ as shown in Fig. 7.1c. After applying the previous two steps, the resultant rescheduling set $U$ contains queries $< q_K, q_I, q_J >$. The ready time for $q_k$ using the updated *schedule table* (Table 7.3) is

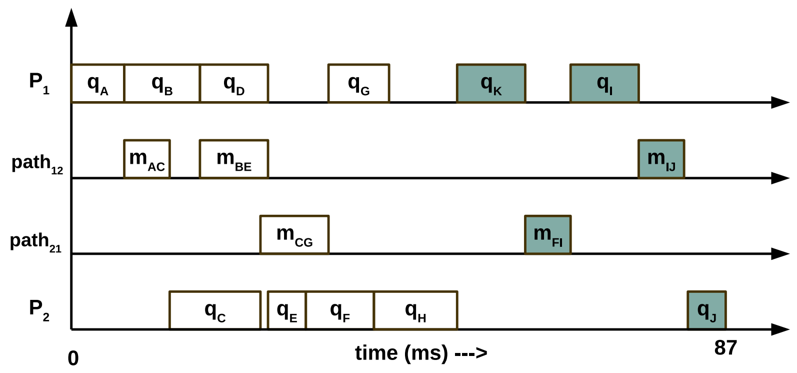
**Algorithm 11** Pseudo-code representation for checking whether any task in the *schedule table* overlaps $u_i$

```
$time \leftarrow$ Start time of query $u_i$
$endtime \leftarrow$ Finish time of query $u_i$
$processor(u_i) \leftarrow$ Processor assigned to a query $u_i \in U$
$Queries \leftarrow$ Configuration of the queries in the schedule table
function GetConflictingTask ($Queriess, time, endtime, processor(u_i)$)
for each query $q_i \in Queries$ do
    $ST \leftarrow$ Start time of query $q_i$
    $ET \leftarrow$ Finish time of query $q_i$
    $processor(q_i) \leftarrow$ Processor assigned to query $q_i$
    if $processor(u_i) == processor(q_i)$ then
        if $time == ST$ then
            return $q_i$
        else if $time \geq ST$ and $time < ET$ then
            return $q_i$
        else if $endtime > ST$ and $endtime \leq ET$ then
            return $q_i$
        else if $ST \geq time$ and $ST < endtime$ then
            return $q_i$
        else if $ET \geq time$ and $ET < endtime$ then
            return $q_i$
        end if
    end if
end for
return $\emptyset$
```

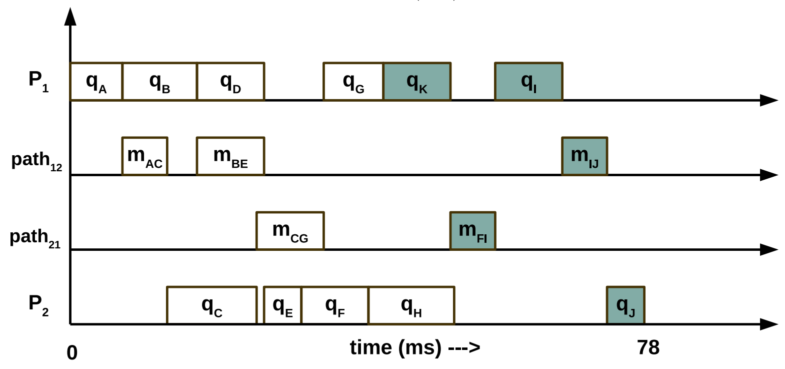
**Algorithm 12** Pseudo-code representation for checking whether any message in the *message table* overlaps message $m_{ji}$ from parent query $p_j$ to $u_i$

```
$m_{ji} \leftarrow$ a message from parent query $p_j \in G_D$ to child query $u_i \in U$ : $\exists$ a path $p(p_j -> u_i) \in G_D$
$time \leftarrow$ Start time of message $m_{ji}$
$endtime \leftarrow$ Finish time of message $m_{ji}$
$path(m_{ji}) \leftarrow$ Path assigned to transmit message $m_{ji}$
$Messages \leftarrow$ Configuration of the messages in the message table
$mess_{ji} \leftarrow$ a message in $Messages$ from parent query $p_j \in G_D$ to child query $q_i \in G_D$ : $\exists$ a path $p(p_j -> q_i) \in G_D$
function GetConflictingMessageQueries ($Messages, time, endtime, path(m_{ji})$)
for each message $mess_{ji} \in Messages$ do
    $path(mess_{ji}) \leftarrow$ Path assigned to transmit message $mess_{ji} \in Messages$
    $q_i \leftarrow$ Destination query of message $mess_{ji} \in Messages$
    $ST \leftarrow$ Start time of message $mess_{ji} \in Messages$
    $ET \leftarrow$ Finish time of message $mess_{ji} \in Messages$
    if $path(m_{ji}) == path(mess_{ji})$ then
        if $time == ST$ then
            return $q_i$
        else if $time \geq ST$ and $time < ET$ then
            return $q_i$
        else if $endtime > ST$ and $endtime \leq ET$ then
            return $q_i$
        else if $ST \geq time$ and $ST < endtime$ then
            return $q_i$
        else if $ET \geq time$ and $ET < endtime$ then
            return $q_i$
        end if
    end if
end for
return $\emptyset$
```

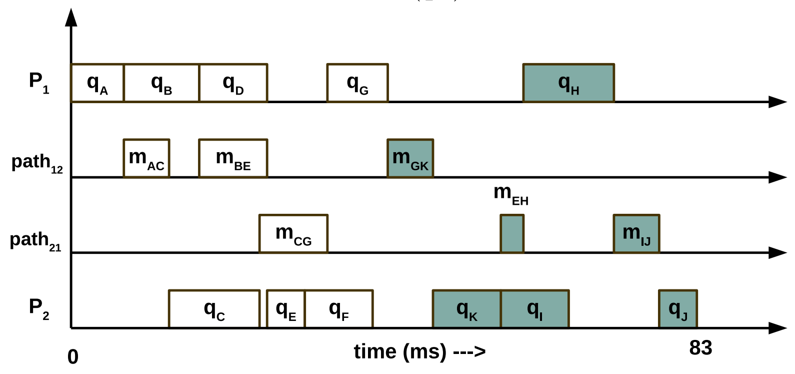
52 ms. It finishes at 61 ms on $P_1$ and at 67 ms on $P_2$ so it is assigned to $P_1$ and the *schedule table* is updated. Similarly, tasks $q_I$ (ready time = 61 ms) and $q_J$ (ready time = 76 ms) are assigned to processors $P_1$ and $P_2$ respectively. The resultant schedule given in Fig. 7.3a is a valid solution because all the queries are fulfilling their timing constraints. In this case $q_k$ has a relative deadline $D(q_K)$ of 46 ms which means that the first instance of $q_k$ has an absolute deadline of 88 ms since its release time is 42 ms (finish time of its parent query $q_G$). If the absolute deadline of $q_K$ is changed from 88 ms to 58 ms ($D(q_K)$ = 16 ms) then this solution becomes invalid since $q_k$ is finishing at 61 ms. In this scenario, the original ready time for $q_K$ is calculated using Eq. 5.1. Since $q_K$ has only one parent $q_G$ so its ready time is 42 ms. It has an earlier finish time on $P_1$ (51 ms) than $P_2$ so it is allocated on $P_1$ (57 ms). After the allocation of $q_K$, the *schedule table* is updated. Since $q_H$ is still the last task that finishes its execution in the *schedule table* so the ready time for $q_I$ is 52 ms. It is assigned to $P_1$ whereas $q_J$ is assigned to $P_2$. The resultant schedule (Fig. 7.3b) is valid because all the queries now fulfill their deadlines. To explain the displacement of queries in case of an overlap, assume that it is necessary to allocate $q_k$ on $P_2$. Query $q_k$ starts at 48 ms and finishes at 57 ms on $P_2$ which overlaps the execution of $q_H$ already assigned to $P_2$. Query $q_k$ is given priority in this scenario because it has a greater bottom-level (31) and an earlier deadline (58 ms) than query $q_H$ (24 and 98, respectively). Therefore, $q_H$ is removed from the schedule and added to $U$. After the allocation of $q_k$, the updated *schedule table* gives a maximum finish time of 57 ms. Queries $q_H$ and $q_I$ are both ready at this time but $q_H$ is given priority because it has a greater bottom-level. Query $q_H$ ends at 72 ms on $P_1$ and at 75 ms on $P_2$ so it is assigned to $P_1$. The process is repeated for queries $q_I$ and $q_J$. The resultant schedule given in Fig. 7.3c fulfills all the deadline constraints therefore is considered valid.



(a) $D(q_K) = 46$ ms



(b) $D(q_K) = 16$ ms



(c) Query $q_K$ is assigned to $P_2$

Figure 7.3: Schedule representation for $G_D$ given in Fig. 7.1c implemented on architecture given in Fig. 7.1b using ILS and different use case scenarios (Colored boxes show the modifications to the original schedule given in Fig. 7.2)

### 7.2.4 Complexity of the ILS algorithm

If there are $N$ queries in the diagnostic application that need rescheduling then the minimum time complexity to reschedule these queries on $|P|$ processors is $O(|N|+|M|)|P||L|$ where $|M|$ is the total number of incoming messages required for the execution of the query $n_i \in N$ scheduled on $|L|$ number of links. This means that the time complexity of the ILS algorithm depends upon the total number of queries that need rescheduling. In the worst-case scenario, if the whole DMG needs to be rescheduled then the time complexity is equal to the time complexity of the list scheduling algorithm proposed in Chapter 5. Therefore, even in worst case scenario, the algorithm still remains in pseudo-polynomial time and can be invoked during runtime.

## 7.3 Experimentation and Evaluation

In this section, a series of experiments demonstrate the effectiveness of ILS. The experiments are performed on a platform with four Intel Core i5-6200U cores operating at a fixed clock frequency of 2.30 GHz and 23 GB RAM. The operating system used is fedora 30 with the kernel Linux 5-generic. The algorithm was tested for three types of changes in the system: (i). Addition of a query/edge in the DMG, (ii). Addition of a new DMG, and (iii). Removal of a processor/switch from the system architecture. Each type of change was tested for 60 different structures of DMGs and 60 different structures of system architectures. Therefore, a total of 180 DMGs and 180 architecture models were generated for experimental purposes. The effectiveness of ILS was measured against the total number of modifications in the original schedule and total reconfiguring cost of the system.

For each test case, Algorithm 3 from Chapter 5 is used to compute a primary schedule for the original DMG on the original system architecture. After that whenever a change occurs in the system, Algorithm 9 is used to compute the schedule for the changes while minimising the modifications in the primary schedule and Algorithm 3 is used to compute the schedule for the DMG from scratch without consideration to the primary schedule. A comparative analysis was performed between the total number of modifications and the total reconfiguration cost obtained for the two schedules to demonstrate that ILS is more effective in terms of maximizing the continuity of services and preserving the existing validation and certification results by minimizing the changes in the system. Henceforth, the primary schedule produced by the list scheduler when the system is first initialised will be called the original schedule. The Incremental List Scheduling algorithm is called ILS, and the original List Scheduling algorithm (cf. Algorithm 3) is called LS. The results were measured against the following performance metrics,

1. ***% Similarity between schedules:*** The percentage similarity between the computed schedules is measured by calculating the total number of unchanged task configurations in the *schedule tables* generated by ILS and LS when compared with the original *schedule table*. Similarly, the percentage similarity between the network schedules is measured by the total number of unchanged message configurations in the *Message Tables* generated by ILS and LS. Let $\%Sim_{ST}$ represent the percentage similarity between the query schedules generated by ILS and LS and $\%Sim_{MT}$ represent the percentage similarity between the message schedules generated by ILS and LS, then Eq. 7.3 is used to calculate the percentage similarity between query schedules and Eq. 7.4 is used to calculate the percentage similarity between message schedules.

$$\%Sim_{ST} = \frac{\text{total number of unchanged query conf. in the } \textit{schedule table}}{\text{total number of queries in the original } \textit{schedule table}} * 100 \tag{7.3}$$

$$\%Sim_{MT} = \frac{\text{total number of unchanged message conf. in the } \textit{message table}}{\text{total number of messages in the original } \textit{message table}} * 100 \tag{7.4}$$

2. ***Reconfiguration cost:*** The total cost required for the reconfiguration of the system for both ILS and LS schedules is measured. This cost determines the amount of disturbance in the already running system. Therefore, lower-cost means that a smaller part of the system is reconfigured and revalidated. If a query in the *schedule table* of ILS or LS has a different execution time (i.e. start and end-time) or is executed on a different processor than in the original *schedule table*, then the total modification cost is incremented by one. The modification cost is incremented twice if both the mentioned cases are true at the same time. Similarly, if a message has a different duration or is transmitted on a different path than in the original *message table*, then the cost is incremented by one. The cost is also incremented twice if a message is executed during a different duration and using a different path. Lastly, the cost is also incremented for the new queries and messages created by the change in the system.

### 7.3.1 Experimental Setup

The algorithms were programmed in Java, and a set of test cases are generated to measure their validity. Three use case scenarios are used: (i). Addition of a query/edge to the existing DMG, (ii). Addition of a new DMG to the system and lastly iii). Removal of a processor/switch from the existing system architecture. For each case, the function *GenForestFire* is used from the library [133] to generate random forest fire directed DMGs with different sizes. The forward probability of an edge between two queries is randomly chosen using the uniform distribution within the range $[0.2, 0.4]$. The execution times of the queries and the amount of data on the edges are also assigned randomly using the uniform distribution within the interval of 10 to 100 ms., and 25 to 50 Mb, respectively. The deadlines for the queries were calculated by reversing the bottom-levels from source to sink queries and multiplying the resultant with 2. The bottom-level is used because it shows the scheduling order of the queries and thus determines which query is more critical than the others. The values of the bottom-levels are reversed from source to sink because the more critical query should have an earlier deadline than the other queries. For simplification of the results, the time period of all the queries is the same, i.e., 300 ms. The history interval $< a_{ij}, b_{ij} >$ between queries $q_i$ and $q_j$ is chosen from the arithmetic range $\{0, 1, 2, ..., 10\}$ such that $b_{ij} \leq a_{ij}$.

The system architecture contains different distributed clusters connected through a defined network topology. Each distributed cluster consists of two processors and a switch connected through a star topology. Ring, star, or bus topology are used to connect the distributed clusters. The bandwidth of the links was selected randomly using the uniform distribution within the 128 to 256 Mbps range. The performance of the algorithm was measured in all the cases against the performance metrics mentioned before. The details of the use cases are described below, and Table 7.5 gives a brief overview of the system specification for the three use cases. Table 7.6 shows the average execution time of ILS and LS for the three use cases. From the table, it is visible that on average there is not a high difference between the computation times of both algorithms. Understandably, the better choice is to compute the schedule from scratch using LS but it will majorly effect the reconfiguration cost of the system. However, if ILS is used the computation time is still feasible and it also reduces the reconfiguration cost of the system. This is a trade-off between computation time and reconfiguration cost so at the end it depends upon the user of what they would prefer. However, since the average execution time of ILS is in seconds, it can be used to recompute the schedule at run-time.

1. ***Case 1 - Addition of a query / edge:*** For the diagnostic service, 60 random forest fire directed DMGs were generated each for sizes 20, 40, 60, 80, 100, 120, 140, and 160. Sixty instances of system architectures were also generated each for the ring, bus, and star topology with varying bandwidths for links. Each instance had five distributed clusters, i.e. ten processors and five switches for ring and bus topology, whereas ten processors and six switches are used for the star topology. For each test, a single edge or a query was added to the DMG with a probability of 0.5. The starting

query of the edge was chosen randomly from the first half of the DMG, i.e. if there is a total of 100 queries in the graph then the parent query was selected randomly from the range $[0, 50]$. In contrast, the sink query was selected randomly from the second half, i.e. from the range $[51, 100]$. It is similar for an added query, but the number of sink queries was taken as either 2 or 4.

2. ***Case 2 - Addition of a DMG:*** For the diagnostic service, a single instance of a 50 query random forest fire directed DMG was generated. For the system architecture, 60 instances each for the ring, star, and bus topology with different bandwidths for links were generated. Each instance had five distributed clusters, i.e. ten processors for all three topologies whereas five switches for the ring and bus and six switches for the star. For the modifications, 60 random forest fire directed DMGs were generated for sizes 20, 40, 60, 80, and 100.

3. ***Case 3 - Removal of a processor/switches:*** For the diagnostic service, 60 instances of random forest fire directed DMGs with a size of 160 queries each were generated. For the system architecture, again 60 instances of 5, 7, 9, 11 and 13 clusters each with the ring, bus, and star topology were generated. It means that each instance has 10, 14, 18, 22, and 26 processors. The ring and bus topology have 5, 7, 9, 11 and 13 switches, whereas the star topology has 6, 8, 10, 12, and 14 switches for each instance. For the modifications, a single processor or a switch was randomly removed from the distributed system with a probability of 0.5.

### 7.3.2 Evaluation

The average percentage similarities between the query and messages schedules and the average reconfiguration costs for both ILS and LS were measured for the three experimental use cases mentioned above. A comparative analysis was performed between the results to show that ILS is more efficient in maximizing the continuity of services and preserving the

Table 7.5: System Specifications for the Experiments

| System Specifications | | **Case 1** *addition of query / edge* | **Case 2** *addition of a DMG* | **Case 3** *removal of switch / processor* |
|---|---|---|---|---|
| *DMG* | Size | 20, 40, 60, 80, 100, 120 140, 160 | 50 | 160 |
| | Instances | 60 | 1 | 60 |
| *System Architecture* | Topology | ring, bus star | ring, bus star | ring, bus star |
| | No. of processors | 10 | 10 | 10, 14, 18 22, 26 |
| | No. of switches | 5/6 | 5/6 | 5/6, 7/8, 9/10 11/12, 13/14 |
| | Instances | 60 | 60 | 60 |

| **Scheduler** | **Case 1** (s) | **Case 2** (s) | **Case 3** (s) |
|---|---|---|---|
| LS | 10.5 | 7.4 | 5.2 |
| ILS | 15.6 | 12.8 | 7.5 |

Table 7.6: Average execution time of ILS and LS for the three use cases

existing validation and certification results by minimizing the changes in the system.

1. ***Addition of a task or an edge to the existing application graph:*** The average similarities between the query and message schedules computed for case 1 using the ring network topology are shown in Fig. 7.4a and Fig. 7.4b, respectively. It is visible from the results that ILS is more successful in keeping the disturbances in the schedule to the minimum. The average similarity decreases for both algorithms with an increase in the size of the DMG. It is mainly because a larger data set means that there is a higher possibility of displacement in the scheduled queries and messages

by the addition of an edge or a query. However, the amount of displacement depends upon the location of the added query or an edge. To better understand this, take a DMG of 20 queries where all the queries are connected in a straight line, i.e. one after the other from 1 to 20. If a query is added between queries 10 and 11, then the scheduling of query 11 and all the queries after that is invalidated, i.e., ten tasks need rescheduling. But if the new query is added between queries 17 and 18, then only three queries need to be rescheduled. If the same scenario is considered with a DMG of 40 queries and a query is added between queries 20 and 21, then 20 queries would need rescheduling. Thus, the similarity between schedules depends upon the structure and size of the existing DMG and the location of the added query/edge. In the experiments, a new query/edge is placed between the first and second half of the DMGs irrespective of their size. Therefore the average similarity decreases with an increase in the size of the DMG.

Fig. 7.4c shows the average reconfiguration cost computed for case 1 using the ring network topology. There is a significant difference between the average reconfiguration costs of ILS and LS specifically in case of bigger DMGs. The cost increases with an increase in the size of the DMG, which follows the decrease in the average similarity between schedules. Even with the increasing reconfiguration cost and decreasing similarity between schedules, ILS performs much better than LS. The experiments were performed for all three network topology, i.e., ring, star and bus. The average, minimum and maximum values of the performance metrics for all the experiments in case 1 are given in Tables 7.7, 7.8 and 7.9, respectively.

2. ***Addition of a new DMG to the system***: The average similarities between the query and message schedules generated for the 60 instances using the star topology are shown in Fig. 7.5a and Fig. 7.5b, respectively. The results show that ILS is better in keeping changes in the running application to a minimum compared to LS. The average similarities drop significantly when the size of the DMG is more than the size

Table 7.7: Case 1 - Results for Ring Topology

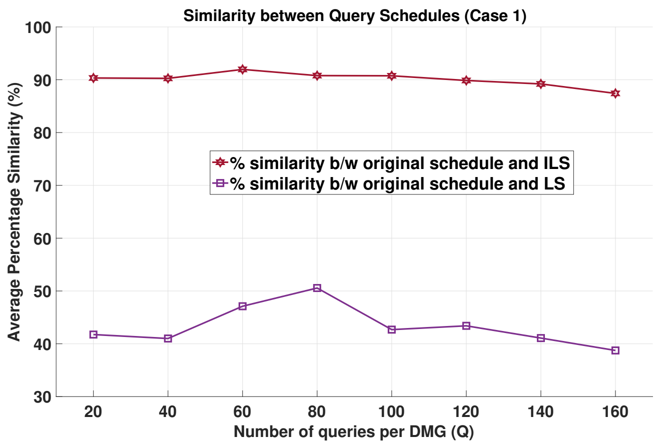
| **DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 61.8/90.3/100 | 48/85.2/100 | 0/3.9/12 |
| | LS | 5/41.7/100 | 0/26.9/100 | 0/17.5/35 |
| 40 | ILS | 67.8/90.2/99.3 | 51/84.9/99.6 | 0/6.7/22 |
| | LS | 2.5/41/95 | 0/29.4/96.6 | 0/35.7/69 |
| 60 | ILS | 65.8/91.95/100 | 45.4/87.2/100 | 0/8.9/39 |
| | LS | 1.67/47.1/98.3 | 0/30.7/97.2 | 1/46.4/111 |
| 80 | ILS | 60/90.78/100 | 40.5/85.8/100 | 0/10.6/47 |
| | LS | 3.7/50.6/100 | 1/36.2/100 | 0/54.6/128 |
| 100 | ILS | 60/90.75/100 | 43.8/85.7/100 | 0/13.2/74 |
| | LS | 8/42.68/100 | 1.45/28.2/100 | 0/81.8/173 |
| 120 | ILS | 60/89.8/100 | 37.9/83.1/100 | 0/13.7/69 |
| | LS | 5.8/43.4/96.6 | 1.36/27.8/97.28 | 4/93.8/164 |
| 140 | ILS | 60/89.2/100 | 48.2/83.6/100 | 1/19.3/66 |
| | LS | 2.14/41.08/95 | 0.37/26.3/93.13 | 7/116.9/238 |
| 160 | ILS | 60/87.4/100 | 24.7/77.5/100 | 0/20.9/86 |
| | LS | 5.62/38.7/100 | 0.99/23.8/100 | 0/139/290 |

Table 7.8: Case 1 - Results for Star Topology

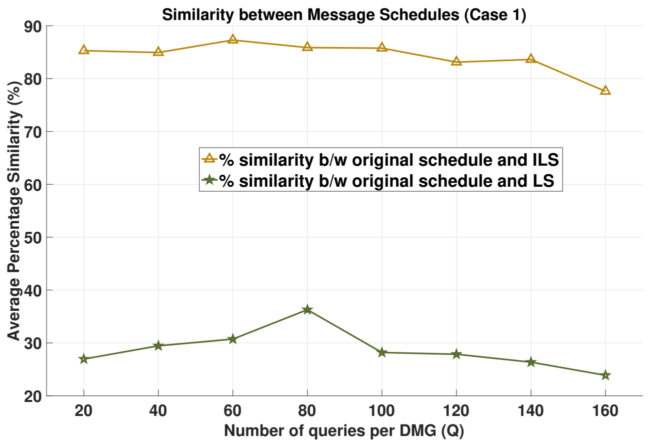
| **DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 68.7/93.3/100 | 55.69/89.2/100 | 0/4/15 |
| | LS | 5/53.9/100 | 0/38.6/100 | 0/13.3/32 |
| 40 | ILS | 64.3/91.2/99.2 | 45.4/86.3/99.6 | 0/6/27 |
| | LS | 2/45.7/95 | 0/32.3/84.9 | 2/29.6/61 |
| 60 | ILS | 59.2/89.9/99.1 | 40.6/84.2/99.5 | 0/9.6/41 |
| | LS | 1/43.6/97 | 0/28.7/93.6 | 2/46.9/101 |
| 80 | ILS | 62/89.8/100 | 44.7/84.1/100 | 0/14/44 |
| | LS | 3.7/45.5/96 | 0.4/30.9/96.4 | 3/61.2/144 |
| 100 | ILS | 61.3/87.8/100 | 39.5/80.5/100 | 0/14.5/57 |
| | LS | 1/47.7/99 | 0/32.4/96.9 | 1/70.4/143 |
| 120 | ILS | 55/89/100 | 26.9/82.6/100 | 1/17.2/61 |
| | LS | 5.8/49.1/95 | 1.4/32.1/93.7 | 6/82.3/163 |
| 140 | ILS | 57.5/90.1/100 | 41.9/84.3/100 | 1/17.6/81 |
| | LS | 4.2/44.6/93.5 | 0.7/29.2/97.6 | 14/104/229 |
| 160 | ILS | 55/85.8/100 | 11/75.3/100 | 0/15.4/70 |
| | LS | 9.3/47/100 | 2.1/33.9/100 | 0/114.4/239 |

Table 7.9: Case 1 - Results for Bus Topology

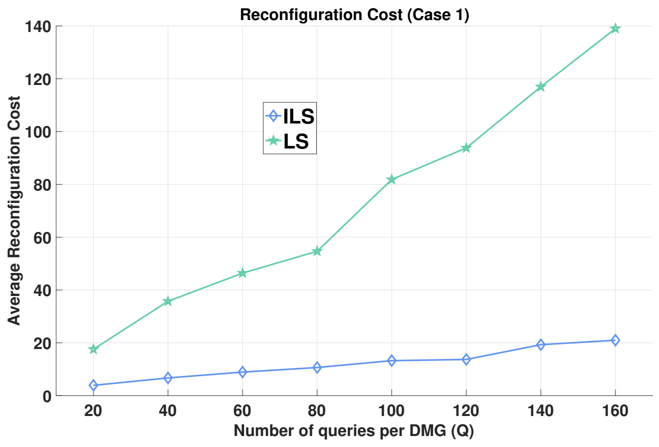
| **DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 63.1/92.6/100 | 43.9/87.6/100 | 0/4/13 |
| | LS | 5/32.7/100 | 0/25.4/100 | 0/20.9/32 |
| 40 | ILS | 70.7/91.7/100 | 51.9/86.4/100 | 1/6.3/32 |
| | LS | 2.5/45.9/100 | 0/32/100 | 0/29.6/62 |
| 60 | ILS | 60/89.1/100 | 40.7/82.2/100 | 0/7.5/26 |
| | LS | 1.7/43.9/93.3 | 0.5/29/88.6 | 6/47.3/85 |
| 80 | ILS | 60/91.3/100 | 38.6/86/100 | 0/10.5/40 |
| | LS | 2.5/47.3/100 | 0/32.2/100 | 0/57.4/139 |
| 100 | ILS | 63.7/90.9/100 | 45.4/84.4/99.3 | 0/12.3/61 |
| | LS | 5/44.3/99 | 0.8/28.5/97.2 | 1/75.7/143 |
| 120 | ILS | 73.3/91.5/100 | 48/86/100 | 0/18.5/68 |
| | LS | 0.8/45/99.2 | 0.17/29.3/99.4 | 1/89.4/185 |
| 140 | ILS | 52.5/89.7/97.5 | 28/82.7/100 | 0/16.9/59 |
| | LS | 0.7/43.8/99.2 | 0/28.7/97.8 | 1/109.5/241 |
| 160 | ILS | 60/87.5/100 | 28.5/84.6/100 | 0/17.5/87 |
| | LS | 1.3/42.7/100 | 0.2/27.8/100 | 0/130/233 |



(a)



(b)

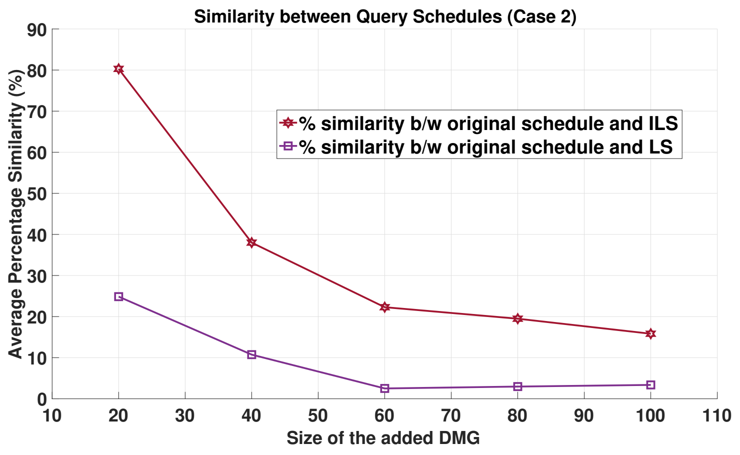


(c)

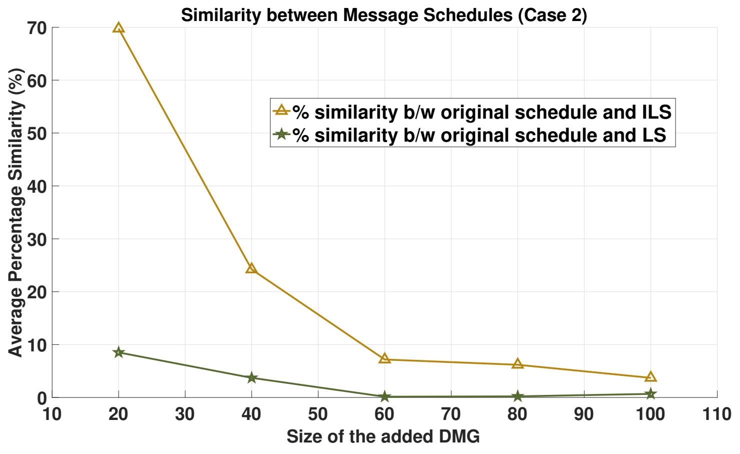
Figure 7.4: Case 1 - Addition of a query or an edge to an existing DMG in a Ring Network

of the already scheduled DMG, i.e. 60 and onward. The decrease in the similarity is because queries in DMGs with sizes 60 and onward have greater path lengths that increase the bottom-level of their queries. Consider a DMG of 3 queries $q_1$, $q_2$ and $q_3$ with WCETs 2, 3, and 4 ms. The DMG has a single path $< q_1, q_2, q_3 >$ from source to sink query and the amount of data transferred between the queries is zero. The bottom-level of $q_1$ is this case is 2 + 3 + 5 = 10, for $q_2$ it is 3 + 5 = 8 and for $q_3$ it is 5. Now consider another DMG with 2 queries $q_1$ and $q_2$ and a single path $< q_1, q_2 >$ from source to sink query. The queries have a WCET of 2 and 4, respectively, so the bottom-level of $q_1$, in this case, is 2 + 4 = 6 and for $q_2$ it is 4. Therefore, the position of the query and the total number of queries in the DMG affect its bottom-level. In the experiments, bottom-level is reversed from source to sink vertex (i.e., the bottom-level of sink vertex is assigned to the bottom-level of source vertex) and is used to calculate the deadline of the queries. For example, in the first DMG considered here, the deadline for $q_1$ is 5 * 2 = 10 ms whereas in the second DMG it is 4 * 2 = 8 ms. Therefore, as the size of the DMG increases the criticality level of its queries also increase which means that the queries in the DMGs with sizes 60 and onward have more critical queries than the queries in the DMG with 50 queries. Thus there is a greater chance that these queries do not keep their deadlines without displacing the already scheduled queries.

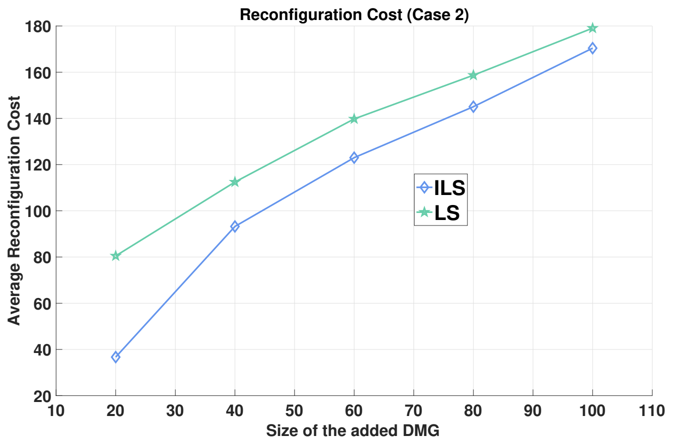
Fig. 7.5c shows the average reconfiguration cost computed for the 60 instances of case 2 using the star topology. The difference between the average reconfiguration cost between ILS and LS decreases as the size of the added DMG increases. The modification cost depends upon the structure and the size of the added DMG. If the queries in the added application have lower deadlines than the already scheduled queries, then it is natural that the scheduled queries are displaced. Even with the decreasing reconfiguration cost, ILS performs better than LS and is a better choice for scheduling the modified DMG. The rest of the results for this case are shown in



(a)



(b)



(c)

Figure 7.5: Case 2 - Adding new applications of varying sizes to an existing system running a DMG of 50 queries in a Star Network

Tables 7.10, 7.11 and 7.12, respectively.

Table 7.10: Case 2 - Results for Ring Topology

| **Added DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 34/76/100 | 10.1/64/100 | 20/38.8/72 |
| | LS | 2/30.6/60 | 0/14.1/40.8 | 46/73.8/104 |
| 40 | ILS | 6/36.9/88 | 1.1/22.2/75.9 | 52/90.8/124 |
| | LS | 2/10.8/60 | 0/4.1/36.3 | 71/111.6/128 |
| 60 | ILS | 6/20.8/58 | 0/5.6/38.5 | 96/123.9/140 |
| | LS | 2/2.45/16 | 0/0.15/4.6 | 122/137.6/149 |
| 80 | ILS | 6/18.9/80 | 0/5.9/69.9 | 96/145.4/164 |
| | LS | 2/3.6/56 | 0/0.78/27.2 | 119/158/168 |
| 100 | ILS | 8/14.8/66 | 0/3.3/49.4 | 126/168.5/180 |
| | LS | 2/3/38 | 0/0.3/12.2 | 150/178/187 |

Table 7.11: Case 2 - Results for Star Topology

| **Added DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 28/80.2/100 | 7.2/69.7/100 | 20/36.6/76 |
| | LS | 2/24.8/62 | 0/8.5/40.2 | 46/80.4/108 |
| 40 | ILS | 4/38/100 | 0.2/24.2/100 | 40/93.2/131 |
| | LS | 2/10.7/56 | 0/3.7/33 | 74/112.4/132 |
| 60 | ILS | 8/22.2/74 | 0.9/7.2/62.3 | 84/123/142 |
| | LS | 2/2.5/16 | 0/0.1/4.5 | 129/139.7/152 |
| 80 | ILS | 6/19.4/76 | 0.5/6.2/61.1 | 104/145/166 |
| | LS | 2/2.9/32 | 0/0.2/7.2 | 145/158.6/170 |
| 100 | ILS | 8/15.8/70 | 0/3.7/51.5 | 123/170/186 |
| | LS | 2/3.3/50 | 0/0.7/23.7 | 139/179/192 |

3. ***Removal of a processor or a switch from existing system architecture***: Fig. 7.6a and Fig. 7.6b show the similarities between the query and message schedules when a processor or a switch is removed from a bus network with a varying number of distributed clusters. There is a considerable difference between the average similarities for ILS and LS. The results show that ILS is more effective than LS in minimising the differences between schedules. The average similarity between schedules increases

Table 7.12: Case 2 - Results for Bus Topology

| **Added DMG Size** | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 20 | ILS | 26/79.2/100 | 6.6/68.4/100 | 20/36.5/81 |
| | LS | 2/27.8/60 | 0/11.3/37.2 | 52/77/104 |
| 40 | ILS | 6/37.8/98 | 0/22.4/95.4 | 42/89.7/118 |
| | LS | 2/11/60 | 0/4/36.3 | 69/110/127 |
| 60 | ILS | 6/21/52 | 0.2/5.8/31.7 | 97/122.5/140 |
| | LS | 2/2.5/18 | 0/0.2/5.4 | 118/137/146 |
| 80 | ILS | 6/19.5/90 | 0/6.3/82 | 90/144.6/156 |
| | LS | 2/3.6/56 | 0/0.9/30.9 | 116/157.7/167 |
| 100 | ILS | 6/14.2/70 | 0/3.3/50.6 | 122/168/78 |
| | LS | 2/3.6/58 | 0/1/36 | 130/177/185 |

when the number of distributed clusters, i.e. processors and switches, is increased. It is because when the possibility of choosing between processors or paths is little as in case of a smaller number of processors and switches, the processor and network load is high to accommodate all the queries and messages. Here when a processor or a switch is removed, it displaces a greater number of queries and messages compared to when the number of processors and switches is high. Therefore, the average similarity between the schedules increases with an increase in the number of distributed clusters. Another factor here is the parallelism in the DMG. If a greater number of queries are parallel to each other, then they have a high chance of being allocated to different processors that decreases the processor load. Here, when a processor is removed from the system, fewer tasks are rescheduled.

Fig. 7.6c shows the average reconfiguration cost computed when a processor or a switch is removed from a bus network comprising a different number of distributed clusters. There is a huge difference between the average reconfiguration costs of ILS and LS, specifically when the number of clusters is increased. The reconfiguration cost is smaller when the load on the processors and switches in the original schedule

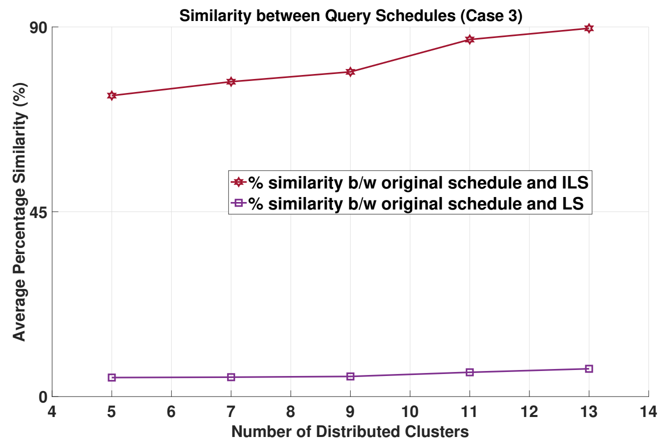
Table 7.13: Case 3 - Results for Ring Topology

| **System Architecture** (clus./proc. /switches) | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 5/10/5 | ILS | 0/67.3/100 | 0/62.6/100 | 0/87.2/314 |
| | LS | 0/4.4/10.6 | 0/1.3/9.2 | 189/235/314 |
| 7/14/7 | ILS | 0/58.5/100 | 0/49.7/100 | 0/108/314 |
| | LS | 0/4.6/11.2 | 0/1.7/8.4 | 164/231/314 |
| 9/18/9 | ILS | 0/61.7/100 | 0/52.7/100 | 0/103.3/311 |
| | LS | 0/4.8/14.4 | 0/1.3/5 | 173/227/311 |
| 11/22/11 | ILS | 0/81.5/100 | 0/77.3/100 | 0/46.4/305 |
| | LS | 0/6.6/13.8 | 0/2.4/20 | 143/204/305 |
| 13/26/13 | ILS | 0/80.2/100 | 0/74.4/100 | 0/50.9/311 |
| | LS | 0/6.5/26.2 | 0/2.3/14.1 | 122/213.4/313 |

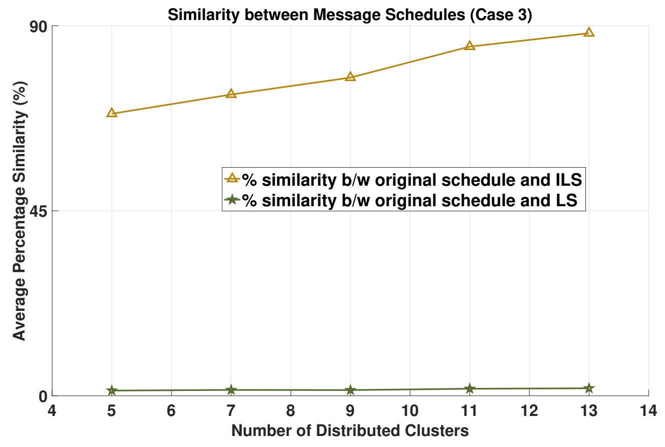
Table 7.14: Case 3 - Results for Star Topology

| **System Architecture** (clus./proc. /switches) | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 5/10/6 | ILS | 0/57.2/100 | 0/51/100 | 0/116.3/314 |
| | LS | 0/4/11.8 | 0/1.14/4.5 | 191/240.9/314 |
| 7/14/8 | ILS | 0/77.2/100 | 0/72.3/100 | 0/60.5/313 |
| | LS | 0/5.2/15.6 | 0/1.5/8.4 | 159/219.7/313 |
| 9/18/10 | ILS | 0/81.3/100 | 0/78.4/100 | 0/48.5/314 |
| | LS | 0/5.3/10 | 0/1.6/6.7 | 166/214.9/314 |
| 11/22/12 | ILS | 0/81.4/100 | 0/77.4/100 | 0/51.3/314 |
| | LS | 0/5.8/15 | 0/1.9/23 | 150/208.6/314 |
| 13/26/14 | ILS | 0/88.3/100 | 0/86.2/100 | 0/34.3/311 |
| | LS | 0/6.2/18.3 | 0/2.1/24 | 148/201/311 |

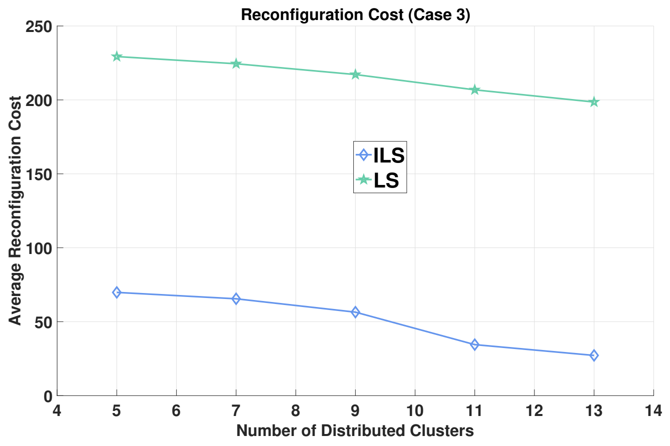
is less; therefore, it decreases when the number of processors and switches in the system are increased. The experiments were performed in this case for all three topologies. The detailed results are given in Tables 7.13, 7.14 and 7.15, respectively.



(a)



(b)



(c)

Figure 7.6: Case 3 - Removal of a processor or a switch from a Bus Network with varying number of processors and switches

Table 7.15: Case 3 - Results for Bus Topology

| **System Architecture** (clus./proc. /switches) | **Scheduler** | **% Sim$_{ST}$** (min./avg./max.) | **% Sim$_{MT}$** (min./avg./max.) | **Cost** (min./avg./max.) |
|---|---|---|---|---|
| 5/10/5 | ILS | 0/73.2/100 | 0/68.5/100 | 0/69.8/307 |
| | LS | 0/4.63/10 | 0/1.2/5.8 | 189/229/307 |
| 7/14/7 | ILS | 0/76.6/100 | 0/73.2/100 | 0/65.5/313 |
| | LS | 0/4.7/9.3 | 0/1.4/7.1 | 163/224/313 |
| 9/18/9 | ILS | 0/79/100 | 0/77.3/100 | 0/56.4/309 |
| | LS | 0/4.9/13.1 | 0/1.4/7.7 | 170/217/309 |
| 11/22/11 | ILS | 0/86.3/100 | 0/84.9/100 | 0/34.5/313 |
| | LS | 0/5.9/10.6 | 0/1.7/15.2 | 151/206.8/313 |
| 13/26/13 | ILS | 0/89.6/100 | 0/88/100 | 0/27.2/312 |
| | LS | 0/6.7/19.3 | 0/1.8/16.4 | 145/198.5/312 |

# CHAPTER 8
# REAL-WORLD USE CASE: FAULT DETECTION AND DIAGNOSIS USING DIAGNOSTIC MULTI-QUERIES IN HVAC SYSTEMS

THIS CHAPTER evaluates the results of the list scheduling algorithm proposed in Chapter 5 by implementing fault detection and diagnosis using DMGs in a real-world use case, i.e., the Heat, Ventilation, and Cooling (HVAC) system.

## 8.1 Background

In the U.S., HVAC systems account for roughly 43% of the overall energy consumption in buildings [134]. Researchers have tried to improve these numbers by deploying more embedded sensors in the systems to monitor temperature, $CO_2$ and humidity levels [135] but including electrical components has made the systems failure-prone. If a fault arises in one of the electrical components of such systems, the sensors may produce erroneous data, and the actuators may behave differently than what is expected. These failures usually lead to general human discomfort, excessive energy consumption, increased overall operation costs, and deterioration of equipment lifespan. Regular checks and maintenance solve this problem but because of the increased cost of on-site maintenance, preventive or predictive maintenance in the form of fault-tolerant systems has become much more significant in recent years [136, 137, 138, 139, 140, 141].

Along with controlling the indoor environment levels, the HVAC systems are generally integrated with hazard detectors such as smoke detectors and play a crucial role in restricting hazardous situations. In hospitals, for example, the HVAC systems are one of the primary responders in the case of a fire. They can contain the situation by staving off oxygen from the fire, expelling smoke from an area, using automatic sprinklers to extinguish the fire or providing a smoke-free pathway to safety [142]. In such situations, HVAC systems must discern a critical from a non-critical situation, i.e. they should identify if an actual fire has occurred or if there is some fault within the system. False alarms have a much more negative impact than expected. For example, in homes, occupants have to search for the detector that is sounding the alarm and determine themselves if the signal is false or an actual hazardous situation exists. This process is time-consuming, stressful and potentially dangerous. Ordinarily, the alarms may mistake regular situations for critical ones. For example, the fire alarms near bathrooms mistake steam from the shower as smoke, or smoke detectors near the kitchen are continuously buzzing because of the steam from the cooking. In these situations, inhabitants may shut the detectors off to avoid the annoyance of false alarms. The malfunctioning of the detection system itself is equally disastrous in life-threatening situations. For example, faulty readings from $CO$ sensors in HVAC systems lead to undetected air poisoning that kills thousands of people each year. Therefore, disabled or ineffective detectors may cause a high death toll and property damage from hazardous situations that otherwise could have easily been prevented [143].

In hazardous situations, thousands of lives depend upon the correct functioning and timely response of the HVAC systems. If a component is faulty, it should be detected before a critical situation occurs. It means that the HVAC systems are time-sensitive, and the fault detection and diagnosis technique (FDD) used should identify faults within the system's provided time-frame. This aspect of the HVAC systems, although essential, is usually not discussed in the literature. In this thesis, a time-critical approach for fault detection and diagnosis in HVAC systems is provided. The HVAC system considered is integrated into a complex building architecture with multiple rooms and floors. There are different sensors and actuators in each room. The HVAC system maintains the indoor environment levels and also provides detection for fire hazards. The faults in the sensors are detected using diagnostic queries that measure the values of the sensors for critical and non-critical situations. The critical situations are considered time-sensitive, and the proposed FDD technique determines whether the fire has occurred or a sensor has malfunctioned within the system's defined time-bound. The detection and diagnosis in non-critical situations have no time restriction and is done to minimize the energy consumption of the HVAC systems due to faulty equipment.

## 8.2 Related Work

Modern HVAC systems consist of an increasing number of remotely controlled actuators and sensors. The HVAC systems that contain these smart devices can minimize the overall energy consumption by keeping track of the overall cost. HVAC Fault Detection and Diagnosis (FDD) schemes that cannot detect the right fault obstruct the process of fault investigation. Therefore, there is a paramount requirement of designing the right FDD techniques tailored for HVAC systems, and this area has received a significant amount of pursuance of many researchers [144, 145, 146].

The authors in [138] classify fault detection approaches in HVAC systems into hardware-based and software-based solutions. In hardware-based solutions, the designers integrate smart components solely for actuator fault detection into the system. These solutions are far more expensive and difficult to reconfigure by introducing additional smart actuator devices. Software-based solutions, although much more appealing in theory than hardware-based fault detectors suffer from dependency on difficult to learn physical models or system-specific detector design specifications.

Du et al. [147] divide FDD methods in HVAC into three types i). Model-based, ii). Rule-based and iii). Data-driven. Model-based methods are designed by adding mass and energy balance based aspects of the system. The residues can be calculated by comparing the actual measurements with the values provided by the system at a certain time. Rule-based methods do not require any system model but rely on expert rules created based on expert knowledge. The data-driven models do not need any physical model or expert knowledge of the system. The information from sensors are compared with the threshold values of sensors, and a warning is generated if the values are not in agreement [126].

Different methods in the state-of-the-art are dealing with the process of FDD by applying techniques such as signal-based, model-based and analytical methods [148]. A Tree-structured Fault Dependence Kernel (TFDK) method is designed along with the online learning algorithm for data streaming in [149]. TFDK is an improvement of the traditional classification method known as Support Vector Machine (SVM). Experimental results in this context show that in comparison with other data-driven methods, TFDK improves the overall performance of the FDD process.

Recently, a wide range of machine learning-based techniques was implemented for the process of FDD in HVAC systems. These techniques include Statistical Process Control (SPC) [150, 151], Neural Networks (NN) [152, 153], Support Vector Machines (SVM) [154, 155], Principal Component Analysis (PCA) [156], and Fisher Discriminant Analysis (FDA) [157]. Among all these techniques, the SPC and PCA are methods of unsupervised learning which do not require expert knowledge for the labelling of faults. NN and FDA are supervised learning-based classification methods that require expert training data for the process of FDD. Existing work in the domain of data-driven FDD shows distinguished results in terms of both efficiency and accuracy. Therefore, two important issues, namely fault severity and fault interdependence, are ignored while considering the homogeneity based assumptions [158, 159].

## 8.3 Problem Formulation

In this thesis, Fault detection and diagnosis using Diagnostic Queries (FDQ) is proposed for an HVAC system. The considered HVAC system is located in a complex building architecture with multiple rooms and floors. Each room has a certain set of sensors and actuators depending upon its requirements and function. Fault detection and diagnosis in the HVAC systems is considered in two separate scenarios: i). in rooms that have hazardous fire equipment, e.g. kitchen and ii). in rooms where there is no possibility of a fire, e.g. study room. The functioning of the HVAC system in a fire-prone environment is considered

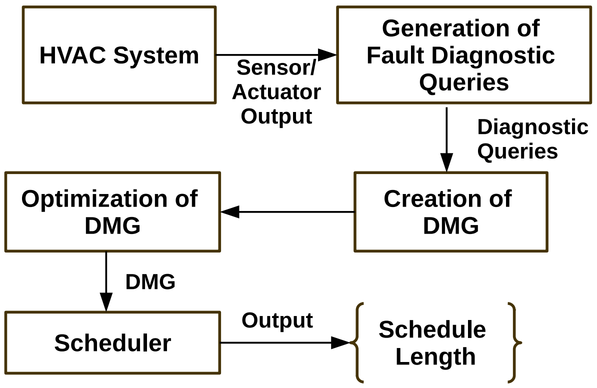


Figure 8.1: Flow diagram of schedule generation for Fault detection and diagnosis using Diagnostic Queries (FDQ)

a critical situation and is highly time-sensitive. It means that the faults in the electrical equipment in such situations should be identified and rectified within the system's defined time-bound. In contrast, the maintenance of comfort levels (e.g. room temperature) by the HVAC systems is considered non-critical and has no time restriction for fault detection and diagnosis. The sensors and actuators in each room provide output data that is stored in distributed local databases. The diagnostic queries are formulated that compare the sensor and actuator values with certain thresholds and concentration levels to determine faults in the system. Since multiple sensors and actuators are considered and in turn, multiple sets of queries, therefore a Diagnostic Multi-query Graph (DMG) is generated for a structured and timely diagnosis. Since highly time-critical situations are considered where a false output could cause a dangerous situation, so the maximum time FDQ takes to identify and diagnose faults is known beforehand, i.e. the total execution time of the DMG. For this purpose, the DMG is given to a scheduler that allocates, maps, and schedules the queries and gives a tentative time for fault detection. This pre-defined time should always be less than the deadline defined by the system for the detection of fire in the building. To ensure that this deadline is met, the DMG is optimized before giving it to the scheduler that computes a feasible schedule for the execution of the DMG. Fig. 8.1 gives the main steps of our FDQ approach.

### 8.3.1 Building Architecture

A building with multiple numbers of rooms, corridors, and floors is considered. There are two main components in the building, i). The HVAC model and ii). the FDQ execution model.

1. ***HVAC model:*** The rooms, corridors and floors of the building along with the sensors and actuators form the HVAC model. Each room and corridor in the building has a certain set of sensors and actuators depending upon its type, function, and requirements for fault detection. The rooms are divided into two categories,

    - **Catastrophic Rooms (CR):** All the rooms that contain hazardous fire equipment are sorted into this category. In such rooms, the HVAC system detects the occurrence of a fire and sounds an alarm corresponding a fire. The fault detection and diagnosis in such rooms is highly time-critical because a faulty component can either overshadow the existence of an actual fire or can sound a false alarm that creates a stressful and dangerous situation. Therefore, in such rooms, a defective component must be identified within the system's defined deadline before a catastrophic situation occurs. The following sensors and actuators are considered in such rooms.

        – *Temp:* A sensor that measures the temperature of the room and the adjacent corridor.

        – *$CO_2$:* A sensor that measures the concentration level of carbon dioxide in the room.

        – *Heater (thermostat):* An actuator that controls the amount of heat in the room.

      A fault can occur in any of the mentioned components. If the output from the three components point to the occurrence of fire, then the system sounds an alarm. In contrast, if one of the components is giving a value above its threshold or is asynchronous with the values of the other two components, then the system classifies it as defective. Here, the component needs fixing or replacement. To simplify the situation, it is assumed that only one of the electrical components may be defective at a time.

- **Normal Rooms (NR):** The non-possibility of fire hazards classifies a room into this category. These rooms require the normal functioning of the HVAC systems, i.e. controlling temperature and humidity levels to provide a comfortable environment for a living. The fault detection in such cases is not time-critical and thus is not restricted by any deadline. However, it is still essential to identify the faults since a defective component adds to the energy consumption of the HVAC system and causes general discomfort for the inhabitants. The following sensors and actuators are considered in such rooms.
  - *Temp:* Similar to CR, this sensor also measures the temperature of the room and the adjacent corridor.
  - *$CO_2$:* This sensor functions similarly to the carbon dioxide sensor in CR, i.e. it measures the level of carbon dioxide in the room. Here, it is used to measure the air concentration and humidity level.
  - *Damper:* It deals with the airflow calculations of the ventilation system and takes the input from the $CO_2$ sensor.

  The proposed technique detects the faults in all the above components. If the output of one of the above components is above its threshold or is not in agreement with the outputs of the other two components, then this component is defective and needs replacement. Similar to CR, it is assumed that only one component is faulty at a time.

For example, the chemistry lab in the HVAC model of a school building is classified as a catastrophic room (CR) because it has various sensitive chemicals that can cause

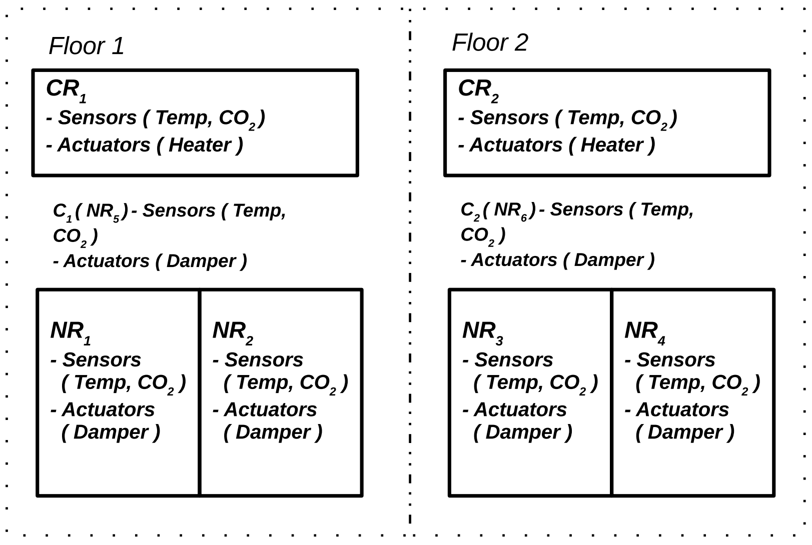


Figure 8.2: HVAC model of a 2 floor building each with 3 rooms and one corridor

explosions. In contrast, the gym is categorized as a normal room (NR) since it has no fire-prone equipment. The method described in [160] is used to compose the HVAC model in MATLAB/Simulink. For simplification of the model, the number of catastrophic rooms per floor is restricted to one. The rest of the rooms and the corridor (C) on the floor are classified as NRs. There is no restriction on the number of rooms per floor or the number of floors in the building, and the FDQ technique is scalable to any kind of building structure provided the rooms can be classified into one of the mentioned categories. Fig. 8.2 shows the HVAC model for a two-floor building, each with three rooms and one corridor.

2. ***FDQ execution model:*** FDQ execution model comprises the part of the building that stores the output data of the sensors and also provides the architecture platform for the scheduling and the execution of the diagnostic queries. This model has the following components.

    - *System architecture:* The system architecture provides the distribution platform for the execution and scheduling of the DMG. The system architecture defined in Section 4.1 is also used here. The architecture consists of processors,

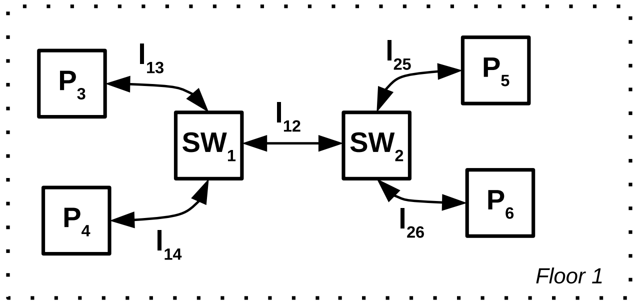


Figure 8.3: Example of the system architecture of floor one of the building

switches and bi-directional links and can be presented using Eq. 4.4. Each distributed system here is represented with *S*. It is assumed that each floor in the building has a separate platform to execute and schedule the DMG. An example of such a platform for floor one of the building used to create the HVAC model in Fig. 8.2 is given in Fig. 8.3.

- *Communication Network:* The building is assumed to have a dedicated wired communication network (e.g. LAN) that transmits the information from the HVAC model to the FDQ model and also transmits the data between the components of the FDQ model. It is essential for the information flow between the different components of our technique.
- *Distributed Database:* The database is distributed locally, i.e., each processor in the system has a local database. The sensors and actuators in the rooms and corridors transfer their values through the communication network mentioned above to their nearest distributed platform. Parts of the distributed databases are timely and consistently replicated to enable the distributed execution of diagnostic queries. The scheduling algorithm decides what parts of the database should be replicated and what parts should be deleted to conserve memory.

Fig. 8.4 shows different components of the building architecture and the flow of information between them.

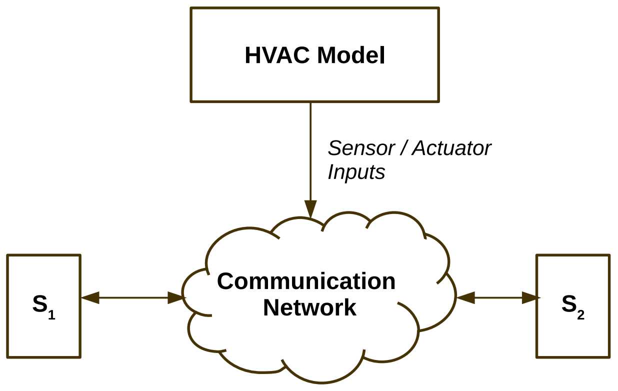


Figure 8.4: Different components of the building architecture

## 8.4 Formulation of Diagnostic Multi-query Graphs (DMGs)

With multiple rooms in the building, each with its own set of sensors and actuators, it is difficult to keep track of the data and the execution of the corresponding diagnostic queries. To simplify the process, a DMG is formulated that represents queries and data relationships in a structured form as a foundation for the scheduler. A DMG can be represented using Eq. 4.4 where $Q$ is a finite set of $|Q|$ queries and $M$ is a finite set of $|M|$ data directed edges between the queries. Here, the queries without incoming edges are features, queries with both incoming and outgoing edges are symptoms and queries without outgoing edges are faults. Each $q_i \in Q$ is represented by the tuple $< W(q_i), D(q_i), T(q_i) >$ where $W(q_i)$ is the worst-case execution time of the query, $D(q_i)$ is its relative deadline and $T(q_i)$ is its time period. Each $m_{ij} \in M$ is represented by the tuple $< D(m_{ij}), < a_{ij}, b_{ij} >>$ where $D(m_{ij})$ is the amount of output tuples transferred from $q_i$ to $q_j$ and $< a_{ij}, b_{ij} >$ is the history-interval of the edge $m_{ij}$. Two example DMGs are given in Fig. 4.8.

For each room (NR or CR), a separate DMG is created. This DMG is based on the faults that need to be detected in that particular room. Therefore, for each room, one DMG is created. All such DMGs from one floor are then combined to formulate the DMG for that particular floor. For simplification of the problem, only two floors are considered for fault detection and diagnosis.

1. *Creation of diagnostic queries for floor 1, $FR_1$:* The FDQs written for $NR_1$ and $NR_2$ present at $F_1$ are shown in Table 8.2 and Table 8.3. In the case of normal rooms, two types of situations are monitored (i). The occurrence of faults in the CO$_2$ sensor, (ii). Monitoring the concentration levels of CO$_2$ and temperature in a room. The concentration of CO$_2$ and temperature is only monitored to maintain the comfort level in the room. The fault detection in this scenario is not time-sensitive. Therefore critical faults like fire are not detected in such rooms. The main purpose of fault detection in normal rooms is to minimize the overall consumption of energy. The diagnostic queries given in Table 8.2, are used to identify faults in the $CO_2$ sensor. Whenever, $CO_2faultcounter$ is one it means that the corresponding sensor is faulty even if the other parameters i.e. *room_temperature1* and *CO$_2$concentration* have values greater than their threshold levels. In contrast, if the $CO_2faultcounter$ is zero, but other parameters have values greater than their thresholds then it means that the concentration of $CO_2$ and the room temperature are not at comfortable levels as shown in Table 8.3. In this scenario, a possible solution is for the occupant to open the window to maintain comfort levels. The FDQs written for $CR_1$ are given in Table 8.4. The diagnostic queries given in the mentioned table are used to detect fire in the catastrophic room on floor 1. Whenever, the concentration levels of temperature, $CO_2$, and heat are above their threshold values but the *HeaterWarningCounter* is zero, i.e. there is no fault in the heater actuator then it means a fire has occurred in the room. Upon the occurrence of a fire, the HVAC system sounds an alarm for the occupants to clear the building.

2. *Creation of diagnostic queries for floor 2, $FR_2$:* The FDQs written for rooms $NR_3$ and $NR_4$ present at floor $FR_2$ are shown in Table 8.5 and Table 8.6, respectively. The HVAC system monitors two scenarios in this room (i). Detecting faults in temperature sensor and (ii). Monitoring the CO$_2$ and temperature levels in the room. Similar to $FR_1$, both $NR_3$ and $NR_4$ are non-critical. Therefore, the main goal of the HVAC

system in this room is to minimize energy. The diagnostic queries in Table 8.5 are used to identify faults in the temperature sensor. The queries show that whenever $Temperaturefaultcounter$ is one, then it means the corresponding sensor is faulty irrespective of the values of the other parameters. On the other hand, if the value for $Temperaturefaultcounter$ is zero, but the other parameters, i.e. room temperature and concentration of $CO_2$ have values greater than their threshold. It means that the room is not at the desired comfort level. In this case, the occupant might need to open a window. The mentioned diagnostic queries for measuring the concentration levels in $NR_4$ are given in Table 8.6. The diagnostic queries for $CR_2$ at $FR_2$ are identical to that of the critical room at $FR_1$ and similarly, are used to detect fire in the room (Table 8.7).

3. *Creation of DMGs for each room on both floors:* After generating the diagnostic queries, the next step is to combine the FDQs for each room to create its DMG. For example, the FDQs given in Table 8.2 are used to create a DMG for room $NR_1$ at $FR_1$. The resultant DMG (shown in Fig. 8.5a) detects and diagnosis faults in the $CO_2$ sensor. Similarly, the DMG created for $NR_2$ monitors the comfort level in the room while the DMG created for $CR_1$ detects fire in the room. At $FR_2$, the DMG of $NR_3$ detects and diagnosis faults in temperature sensor while the other two DMGs work similarly to their counterpart at $FR_1$.

4. *Creation of DMGs for normal and catastrophic rooms at each floor:* After generating DMGs from the FDQs, the next step is to combine these DMGs to form the DMG for normal and catastrophic rooms of the floor. For example, in Fig. 8.5b, the DMGs for $NR_1$ and $NR_2$ are combined together using a source vertex to form the DMG for the normal rooms of $FR_1$. If $FR_1$ had more than one critical room, then their corresponding DMGs would have been combined to make the final DMG for catastrophic rooms at $FR_1$. The DMGs for critical and normal rooms are kept separate because

they have different queries and need to be compared with the DMGs of normal and catastrophic rooms from other floors. If the resultant tuples from the normal room DMG of one floor are different from the normal room DMG of the other floor, then one of the rooms have a faulty sensor.

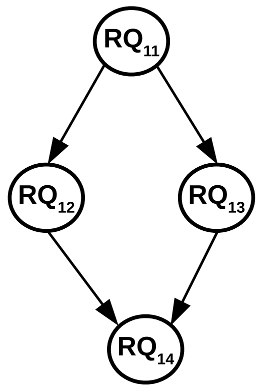
The parameters to detect the faults are the resultant tuples from the DMGs (RT), comparison between DMG of one floor to the other floor (CA), and the threshold values of the sensors (TV). When all of these values are one, then it means one of the sensors in one of the rooms on the floor is faulty. Sample parametric values are shown in Table 8.1.

Table 8.1: Fault Parameters

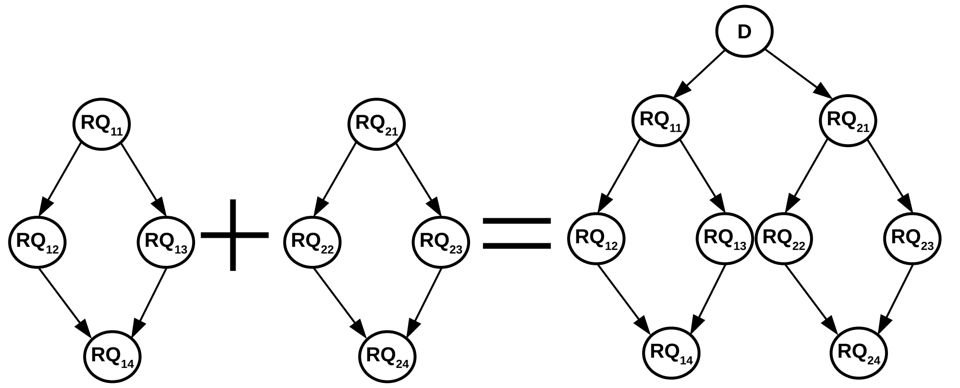
| | **RV** | **CA** | **TV** | **Fault (Yes/No)** |
|---|---|---|---|---|
| $DMG_{FR_1}$ | 1 | 1 | 1 | Yes |
| $DMG_{FR_2}$ | 1 | 1 | 1 | Yes |
| $DMG_{FR_1}$ | 1 | 0 | 0 | No |
| $DMG_{FR_2}$ | 0 | 1 | 0 | No |

## 8.5 Optimization

In the previous section, the method to create DMGs for a two-floor building with three rooms was explained. Since each room in the building has its own DMG, this means that an increasing number of rooms results in multiple DMGs with large inputs that need to be processed within stringent timing constraints. There is a possibility that the scheduler is not able to allocate and map these DMGs for the available resources or the diagnosis technique is unable to identify the faults within the mentioned timing constraints because the DMGs are processing a huge amount of input data. To overcome this problem, created DMGs are forwarded to an optimizer before giving them to the scheduler. The optimization technique proposed in [9] is used here to optimize the DMGs.



(a)



(b)

Figure 8.5: (a). DMG for $NR_1$ (b). DMG for normal rooms of floor $FR_1$

Table 8.2: Diagnostic queries for $CO_2$ sensor faults in $NR_1$ of $FR_1$

| Names | Diagnostic Queries |
|---|---|
| $RQ_{11}$ | Select $CO_2faultcounter$ from $CO_2sensor$ where $CO_2faultcounter$ = '1' |
| $RQ_{12}$ | Select $CO_2value, CO_2faultcounter$ from $CO_2sensor$ where $CO_2concentration > 399$ and $CO_2faultcounter$ = '1' |
| $RQ_{13}$ | Select *room_temperature1*, $CO_2faultcounter$ from *rooms*, $CO_2sensor$ where *rooms.id* = $CO_2sensor.id$ and *room_temperature1* $> 19$ and $CO_2faultcounter$ = '1' |
| $RQ_{14}$ | Select $CO_2faultcounter, CO_2value$, *room_temperature1* from $CO_2sensor$, *rooms*, where *rooms.id* = $CO_2sensor.id$ and *room_temperature1* $> 19$ and $CO_2concentration > 399$ and $CO_2faultcounter$ = '1' |

Table 8.3: Diagnostic queries to monitor the levels of $CO_2$ and *temperature* in $NR_2$ of $FR_1$

| Names | Diagnostic Queries |
|---|---|
| $RQ_{21}$ | Select $CO_2faultcounter$ from $CO_2sensor$ where $CO_2faultcounter$ = '0' |
| $RQ_{22}$ | Select $CO_2value, CO_2faultcounter$ from $CO_2sensor$ where $CO_2concentration > 399$ and $CO_2faultcounter$ = '0' |
| $RQ_{23}$ | Select *room_temperature2*, $CO_2faultcounter$ from *rooms*, $CO_2sensor$ where *rooms.id* = $CO_2sensor.id$ and *room_temperature2* $> 19$ and $CO_2faultcounter$ = '0' |
| $RQ_{24}$ | Select $CO_2faultcounter, CO_2value$, *room_temperature2* from $CO_2sensor$, *rooms*, where *rooms.id* = $CO_2sensor.id$ and *room_temperature2* $> 19$ and $CO_2concentration > 399$ and $CO_2faultcounter$ = '0' |

## 8.6 Scheduler

The method described in Algorithm 3 is used to schedule the diagnostic queries and communication messages to the target system architecture described in the previous section. The main steps of the scheduler are as follows,

1. Calculate the hyper-period of the DMG using Eq. 4.9 and the number of times each query executes within one $H_G$ using Eq. 4.10.

2. Translate the history-intervals to directed edges in the DMG using Algorithm 2.

3. Calculate the bottom-level of each query $q_i \in Q$ using Eq. 4.18.

4. Add the queries to the ready list. A query is said to be ready if all of its predecessor queries have completed their required executions. Order the ready list in descending order of the bottom-level. The ties are broken by prioritizing the query whose successors have the greater bottom-level.

Table 8.4: Diagnostic queries for fire detection in $CR_1$ of $FR_1$

| Names | Diagnostic Queries |
|---|---|
| $CQ_{11}$ | Select *HeaterWarningCounter* from *Heateractuator* where *HeaterWarningCounter* = '0' |
| $CQ_{12}$ | Select *TemperatureValue*, *HeaterWarningCounter* from *TemperatureSensor T*, *Heateractuator H* where $T.id = H.id$ and *TemperatureValue* $>$ 1100 and *HeaterWarningCounter* = '0' |
| $CQ_{13}$ | Select $CO_2$*concentration*, *TemperatureValue* from $CO_2$*sensor C* , *TemperatureSensor T* where $C.id = T.id$ and $CO2_c$*oncentration* $>$ 12800 and *HeaterWarningCounter* = '0' |
| $CQ_{14}$ | Select *Heatvalue*, $CO_2$*Sensorvalue*, *TemperatureValue* from *HeatSensor H*, $CO_2$*Sensor C*, *TemperatureSensor T* where $H.id = C.id$ and $C.id = T.id$ and *Heatvalue* $>$ 1100, $CO_2$*concentration* $>$ 12800 and *HeaterWarningCounter* = '0' |

Table 8.5: Diagnostic queries for Temperature sensor faults in $NR_3$ of $FR_2$

| Names | Diagnostic Queries |
|---|---|
| $RQ_{31}$ | Select *Temperaturefaultcounter* from *Temperaturesensor* where *Temperaturefaultcounter* = '1' |
| $RQ_{32}$ | Select $CO_2$*concentration*, *Temperaturefaultcounter* from $CO_2$*sensor C*, *Temperaturesensor T* where $C.id = T.id$ $CO_2$*concentration* $>$ 399 and *Temperaturefaultcounter* = '1' |
| $RQ_{33}$ | Select *room_temperature3*, *Temperaturefaultcounter* from *rooms*, $CO_2$*sensor* where *rooms.id* = *CO2sensor.id* and *room_temperature3* $>$ 19 and *Temperaturefaultcounter* = '1' |
| $RQ_{34}$ | Select *Temperaturefaultcounter*, $CO_2$*concentration*, *room_temperature3* from $CO_2$*sensor*, *rooms*, where *rooms.id* = $CO_2$*sensor.id* and *room_temperature3* $>$ 19 and $CO_2$*concentration* $>$ 399 and *Temperaturefaultcounter* = '1' |

5. Assign the query to the processor that gives it the earliest start time and on which they query fulfils its deadline. Moreover, assign the communication messages between the query and its predecessors on paths that give them the earliest finish time.

6. Repeat steps 4 and 5 until all the queries and messages are successfully scheduled.

A detailed version of the scheduler has been described in Chapter 4.

## 8.7 Experimentation and Results

For experimentation purposes, the technique proposed in [160] is used to design an HVAC system model in MATLAB/Simulink for building architectures containing 12, 18, 24, 48, 60, and 80 rooms each. The corresponding DMGs comprise of 12, 18, 24, 48, 60, and 80 diagnostic queries and the system architecture consists of 2 and 3 distributed clusters where each cluster contains two processors and one switch. Three different network topology were used: (i). Star, (ii). Ring and (iii). Bus. The designed FDQs are executed on a database created in distributed SQL servers. This database contains 85,000 values from

Table 8.6: Diagnostic queries to monitor the levels of $CO_2$ and *temperature* in $NR_4$ of $FR_2$

| Names | FDQS |
|---|---|
| $RQ_{41}$ | Select $Temperaturefaultcounter$ from $Temperaturesensor$ where $Temperaturefaultcounter$ = ’0’ |
| $RQ_{42}$ | Select $CO_2concentration$, $Temperaturefaultcounter$ from $CO_2sensor\ C$, $Temperaturesensor\ T$ where $C.id = T.id$ $CO_2concentration > 399$ and $Temperaturefaultcounter$ = ’0’ |
| $RQ_{43}$ | Select *room_temperature4*, $Temperaturefaultcounter$ from *rooms*, $CO_2sensor$ where *rooms.id* = *CO2sensor.id* and *room_temperature4* $> 19$ and $Temperaturefaultcounter$ = ’0’ |
| $RQ_{44}$ | Select $Temperaturefaultcounter$, $CO_2concentration$, *room_temperature4* from $CO_2sensor$, *rooms*, where *rooms.id* = $CO_2sensor.id$ and *room_temperature4* $> 19$ and $CO_2concentration > 399$ and $Temperaturefaultcounter$ = ’0’ |

Table 8.7: Diagnostic queries for fire detection in $CR_2$ of $FR_2$

| **Names** | **Diagnostic Queries** |
|---|---|
| $CQ_{21}$ | Select $HeaterWarningCounter$ from $Heateractuator$ where $HeaterWarningCounter$ = ’0’ |
| $CQ_{22}$ | Select $TemperatureValue$, $HeaterWarningCounter$ from $TemperatureSensor\ T$, $Heateractuator\ H$ where $T.id = H.id$ and $TemperatureValue > 1100$ and $HeaterWarningCounter$ = ’0’ |
| $CQ_{23}$ | Select $CO_2concentration$, $TemperatureValue$ from $CO_2sensor\ C$ , $TemperatureSensor\ T$ where $C.id = T.id$ and $CO_2concentration > 12800$ and $HeaterWarningCounter$ = ’0’ |
| $CQ_{24}$ | Select $Heatvalue$, $CO_2Sensorvalue$, $TemperatureValue$ from $HeatSensor\ H$, $CO_2Sensor\ C$, $TemperatureSensor\ T$ where $H.id = C.id$ and $C.id = T.id$ and $Heatvalue > 1100$, $CO_2concentration > 12800$ and $HeaterWarningCounter$ = ’0’ |

sensors and actuators. A comparative analysis was done between the unoptimized and the optimized versions of the DMG for the obtained schedule lengths.

Fig. 8.6a shows the results for the 4-processor ring topology network. The results show that the scheduling length increases with an increase in the number of queries and the scheduling length is considerably better for the optimized DMG. Similarly, in Fig. 8.6b (6-processor ring topology network), the results are better for the optimized DMG. The results also depict that increasing the number of resources decreases the scheduling length of the DMG. When the number of resources is high, there is a greater possibility for parallel execution of queries that, in turn, reduces the scheduling length. A similar trend was observed for star and bus network topologies, as shown in Fig. 8.7 and Fig. 8.8. The results for the ring topology were the best out of all three topologies followed by bus and then star. It is expected since ring topology has less number of hops compared to star topology and more direct connections than bus topology, which result in the successful allocation of messages with less execution time. The results show that the proposed FDQ technique can successfully diagnose and detect faults in HVAC systems under stringent

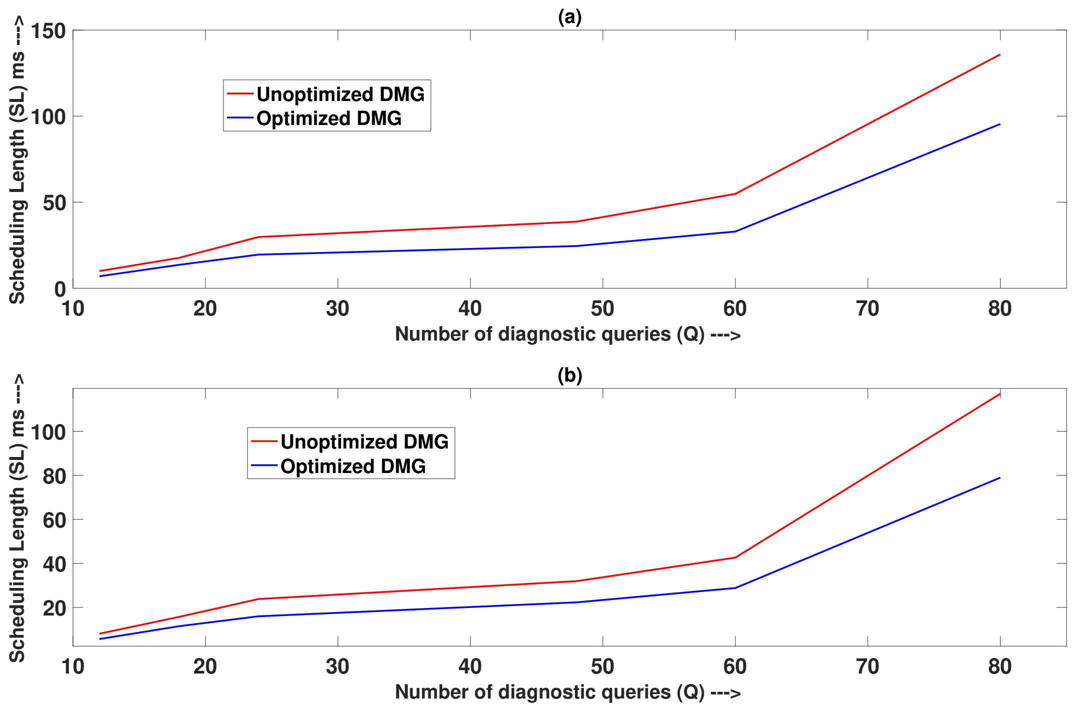


Figure 8.6: (a). 4-processor ring (b). 6-processor ring

timing constraints.

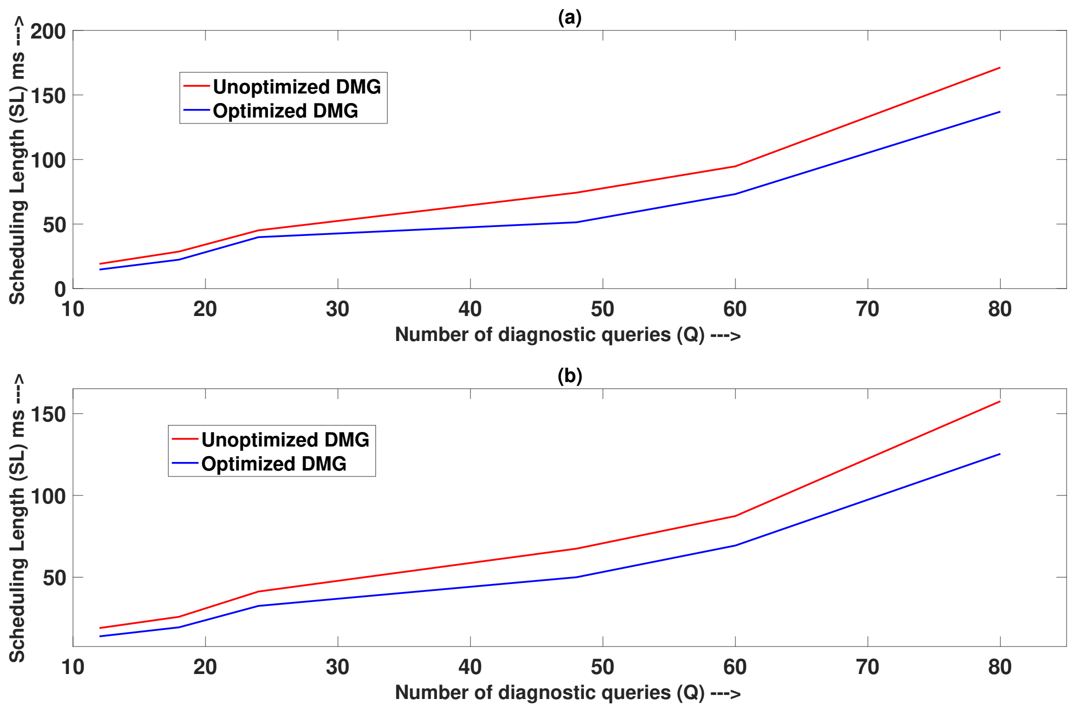


Figure 8.7: (a). 4-processor star (b). 6-processor star

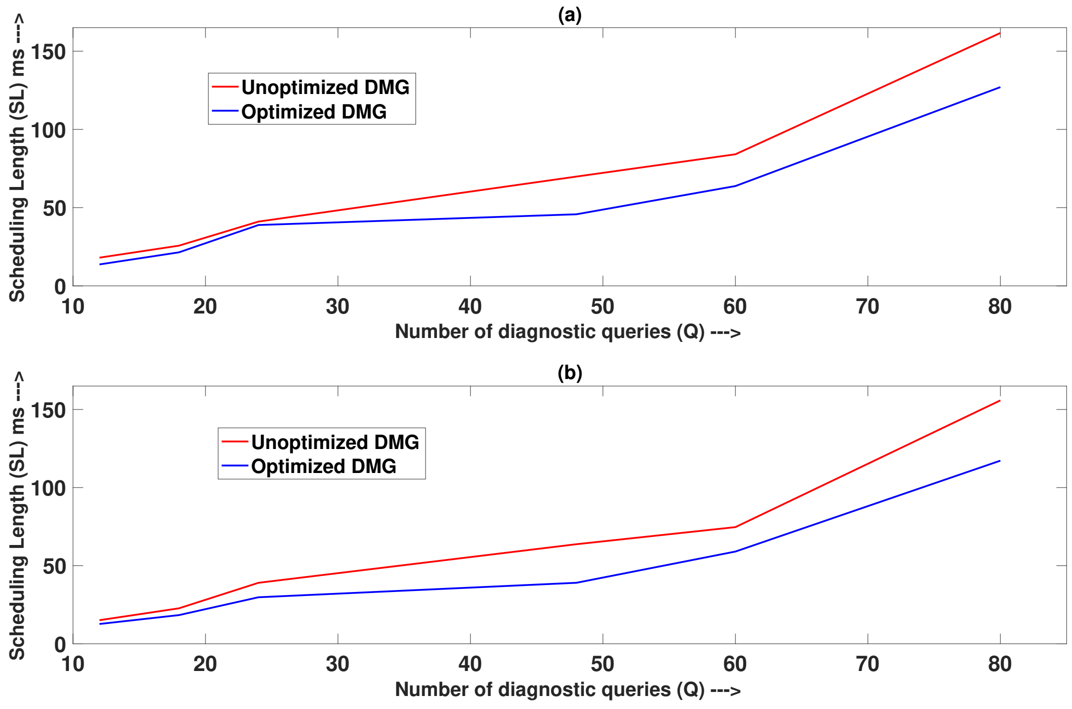


Figure 8.8: (a). 4-processor bus (b). 6-processor bus

# CHAPTER 9
# CONCLUSION

THIS CHAPTER gives the concluding remarks for the work presented in this thesis. This thesis proposes different algorithms to compute time-triggered schedules during runtime for diagnostic applications to detect and diagnose faults in Open Distributed Real-Time Embedded (ODRE) systems.

ODRE systems have requirements for reliable operations with strict timing constraints and an open-world assumption. In such systems, an embedded computer system has to provide its services with dependability that is better than the dependability of its constituent components. Considering the failure rate of electrical components, one way to achieve this level of dependability is to make the ODRE system fault-tolerant. Scheduling the diagnostic service before executing it ensures a predictable temporal behaviour of the diagnostic application and also bounds the time to infer faults. However, scheduling a real-time application in an ODRE system is complicated because of its dynamic nature. In ODRE systems, the term openness means that the electrical components can leave and enter the system at runtime, which means that a fixed schedule generated at the design time cannot be used throughout the functioning of the system. Thus, the schedule for the diagnostic queries and communication messages should be recomputed at runtime whenever there is a change in the system architecture or the diagnostic application. After that, the system can switch to this new schedule, and the message and diagnostic query dispatching can proceed according to the new schedule. The scheduling algorithm should, therefore, be fast enough to recompute a schedule during runtime and efficient enough to maximize the continuity of service and preserve the previous validation results and safety arguments by minimizing the changes in the system. In the state-of-the-art, there are few time-triggered scheduling algorithms that support both the stringent timing constraints and dynamic nature of ODRE systems. Existing algorithms for time-triggerd systems support dynamic changes of the ODRE systems by pre-computing the schedules for all the potential changes offline rather than during runtime of the system. The computation time of most of these algorithms is not suitable for invocation at runtime, or they compute schedules for real-time tasks with unrealistic assumptions, e.g., only considering the bus-based network instead of multi-hop network used in distributed real-time embedded systems.

This thesis proposes three different list scheduling algorithms to compute feasible schedules for real-time applications in ODRE systems. The time complexity and the observed execution time of the algorithms show that they can be invoked during runtime. The proposed algorithms consider the stringent timing and routing constraints while scheduling the tasks and messages in ODRE systems. Effectiveness of the scheduling algorithms is measured using different parameters, e.g., an increase in the size of the application or the total number of modifications in the already scheduled application. The results for the list scheduling algorithm proposed for homogeneous ODRE systems are also evaluated using the real-world use case of Heat, Ventilation and Cooling (HVAC) systems. The results show that the computed schedules are scalable for changes in the system and also fulfil the stringent timing constraints of the ODRE systems. The proposed algorithms are applicable to any domain that has requirements for reliable operations, strict real-time constraints and a need for an open-world assumption. Moreover, the algorithms can be used to schedule any type of application in ODRE systems and are not restricted to the scheduling of diagnostic services.

## 9.1 List Scheduling Algorithm for Homogeneous ODRE systems

The first list scheduling algorithm was proposed for homogeneous ODRE systems, i.e., all the processors in such ODRE systems have the same characteristics. The proposed algorithm elaborates the traditional list scheduling algorithm to schedule an application on an ODRE system such that all the timing constraints of the system are fulfilled. The algorithm has four steps. In the first step, the bottom-levels of the tasks are calculated. In the second step, a ready-list, containing all the tasks that are available for execution, is generated. In the third step, the computed ready-list is ordered according to the calculated bottom-levels. Here, the priority is given to the task that has a greater bottom-level. The ties are broken by prioritising the task whose child tasks have greater bottom-levels than the child tasks of the other conflicted task. In the fourth step, the ready task is allocated to a processor that gives it the earliest start time and on which it fulfils its deadline. In the same step, all the incoming communication messages of the task are scheduled on the paths that give them the earliest finish time. The last three steps of the algorithm are repeated until all the tasks in the application are scheduled. The effectiveness of the proposed algorithm was tested for different randomly generated applications and system architectures. The results show that the algorithm generates a feasible schedule for the applications provided the deadline constraints of the tasks are fulfilled and if enough computation and communication resources are available in the system.

## 9.2 List Scheduling Algorithm for Heterogeneous ODRE systems

The second list scheduling algorithm was proposed for heterogeneous ODRE systems, i.e., the processors in the system have different characteristics. In this case, each task in the application has a different WCET on each processor in the system. The proposed algorithm schedules an application on a heterogeneous ODRE system such that all the stringent timing constraints of the system are fulfilled. The algorithm has four steps. In the first step, a ready-list, containing all the tasks that are available for execution, is generated. In the second step, top-levels of ready tasks are calculated, and the ready list is ordered in increasing order of the computed top-levels. The ties are broken by prioritising the task whose child tasks have greater bottom-levels than the child tasks of other ready tasks in the list. In the third step, the incoming communications, of the ready task, are scheduled on the paths that give them the earliest finish time, and the ready task itself is scheduled on the processor that gives it the earliest finish time. For the selection of the processor, the ready task should also fulfil its deadline on the assigned processor. In the fourth step, the application is updated, i.e., the weight of the scheduled tasks is changed to its WCET on the assigned processor. Also, the weight of the edge between tasks that were assigned to the same processor is updated to zero. All the steps of the algorithm are repeated until all the tasks in the application are scheduled. The effectiveness of the proposed algorithm was tested for different randomly generated applications and system architectures. The results show that the algorithm computes a feasible schedule for the DMGs provided the deadline constraints of the tasks are fulfilled and if enough computation and communication resources are available in the system.

## 9.3 Incremental List Scheduling Algorithm for Homogeneous ODRE systems

The Incremental List Scheduling (ILS) algorithm is an extension of the list scheduling algorithm proposed previously for homogeneous ODRE systems. When the system is first initialised, the original list scheduling algorithm is used to generate a primary schedule for the initial application. The primary schedule is stored in the form of *schedule* and *message tables*. After that, whenever there are changes in the system, ILS recomputes the schedule while minimising the modifications to the already scheduled tasks and communication messages. The changes in the already scheduled application are minimised for the stability of prior applications and to preserve previous validation results and safety arguments. If a feasible schedule is not found, then the system discards the requested changes and keeps running the old schedule.

There are three main steps of the algorithm. In the first step, ILS tries to identify the change in the system by comparing the characteristics of the old and new applications and the old and new system architectures. In the second step, the rescheduling set, consisting of all the tasks that need rescheduling or that are not present in the primary schedule, is computed. The tasks from the rescheduling set are removed from the *schedule table*.

All such messages that are transmitting data to the tasks in the set are removed from the *message table*. The scheduling length of the primary schedule is used as the ready-time for the tasks in the rescheduling set. The final step of the algorithm is to schedule the tasks in the set using the calculated ready-time. There are two different scenarios here. In the first scenario, a task from the rescheduling set is successfully scheduled using the calculated ready-time. In the second scenario, the scheduler is unable to schedule the task with the calculated ready-time and the available resources. Therefore, the earliest ready-time of the task is calculated. ILS then schedules the query using this ready-time onto a processor that gives it the earliest start time and also fulfils its deadline. After each successful allocation of a task from the rescheduling set, the *schedule* and *message tables* are updated. The last step in the algorithm is repeated until all the tasks in the rescheduling set. The tasks in the set are scheduled following their bottom-level where the task with a greater bottom-level is scheduled first.

The algorithm was tested and evaluated for three types of changes in the system (i). Addition of a task/edge, (ii). Addition of a new application, and (iii). Removal of a processor/switch. The effectiveness of the algorithm was measured by the number of modifications in the primary schedule and total reconfiguration cost of the system. The results show that ILS is more effective than the list scheduling algorithm proposed previously in maximizing the continuity of service and preserving the previous validation results and safety arguments by minimizing the changes in the system. The results also show that the computed schedules are scalable with respect to changes in the system. ILS can also be applied to incrementally schedule applications in heterogeneous ODRE systems by employing the list scheduling algorithm proposed in Chapter 6 as the primary scheduling algorithm.

## 9.4 Real-World Use Case: HVAC Systems

The results of the list scheduling algorithm proposed for homogeneous ODRE systems are evaluated in a real-world use case, HVAC systems. Along with controlling the indoor environment levels, the HVAC systems also play a crucial role in restricting hazardous situations. In hazardous situations, thousand of lives depend upon the correct functioning and timely response of HVAC systems. Therefore, a faulty component must be detected before the occurrence of a critical situation. This means that the HVAC systems are time-critical and faults must be detected within the system's defined time-bound. This thesis proposes time-critical detection of faults in HVAC systems using multi-query based diagnostic applications. The diagnostic queries measure the values of the sensors for critical and non-critical situations. The detection of fire in the room is considered a critical situation, whereas the maintenance of comfort levels in the room is considered a non-critical situation. The model of the HVAC system is implemented in MATLAB/Simulink, and different types of diagnostic queries are generated using the features and symptoms of the sensors in the HVAC system. These diagnostic queries are used to formulate DMGs. The list scheduling algorithm computes a feasible schedule for the optimised DMG that depicts the temporal behaviour of the query and messages executions, thus bounding the time to infer faults. Different DMGs and system architectures were generated using MATLAB/Simulink model of the HVAC system to test the effectiveness of the proposed technique. The results show that the proposed list scheduling algorithm can compute feasible runtime schedules for DMGs for fault detection and diagnosis in HVAC systems.

# REFERENCES

[1] Y. Yu, “Modelling and reasoning timing constraints in open distributed real-time and embedded systems,” Ph.D. dissertation, Illinious Institute of Technology, 2009.

[2] T. Abdelzaher, C. Gill, R Rajkumar, and J. Stankovic, “Distributed real-time and embedded systems research in the context of geni,” Technical report, National Science Foundation Workshop, 2006.

[3] A.-M. Grisogono, “The implications of complex adaptive systems theory for c2,” DEFENCE SCIENCE and TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA) LAND . . ., Tech. Rep., 2006.

[4] C. McCann and R. Pigeau, “Clarifying the concepts of control and of command,” in *Proceedings of the 1999 Command and Control Research and Technology Symposium*, vol. 29, 1999.

[5] E. O. Schweitzer, D. Whitehead, A. Guzman, Y. Gong, and M. Donolo, “Advanced real-time synchrophasor applications,” in *proceedings of the 35th Annual Western Protective Relay Conference, Spokane, WA*, 2008.

[6] R. Obermaisser, R. I. Sadat, and F. Weber, “Active diagnosis in distributed embedded systems based on the time-triggered execution of semantic web queries,” in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, IEEE, 2014, pp. 222–229.

[7] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[8] Actian. (2009). “Actian psql,” (visited on 2019).

[9] N. Tabassam, S. Amin, and R. Obermaisser, “Minimizing the worst case execution time of diagnostic fault queries in real time systems using genetic algorithm,” in *Science and Information Conference*, Springer, 2019, pp. 564–582.

[10] A. Bagnato, L. S. Indrusiak, I. R. Quadri, M. Rossi, *et al.*, *Handbook of Research on Embedded System Design*. Information Science Reference, 2014.

[11] X. Fan, *Real-time embedded systems: design principles and engineering practices*. Newnes, 2015.

[12] T. Pop, “Analysis and optimisation of distributed embedded systems with heterogeneous scheduling policies,” Ph.D. dissertation, Institutionen för datavetenskap, 2007.

[13] K. Juvva. (1998). “Real-time systems, carnegie mellon university,” (visited on 2019).

[14] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[15] C. W. Leung, “Architecture of distributed real-time embedded system,” Ph.D. dissertation, KTH Information and Communication Technology, 2013.

[16] H Kopetz, “On the fault hypothesis for a safety-critical real-time system,” *Automotive Software–Connected Services in Mobile Networks*, pp. 31–42, 2006.

[17] J. Škach and I. Punčochář, “Active fault detection: A comparison of probabilistic methods,” in *Journal of Physics: Conference Series*, IOP Publishing, vol. 659, 2015, p. 012 046.

[18] G. Heiner and T. Thurner, “Time-triggered architecture for safety-related distributed real-time systems in transportation systems,” in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, IEEE, 1998, pp. 402–407.

[19] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.

[20] R. El Osta, “Contributions to real time scheduling for energy autonomous systems,” Ph.D. dissertation, Nantes, 2017.

[21] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[22] TTTech. (1998). “Ttp/c specification, website of time-triggered technology,” (visited on 2019).

[23] H. Topcuoglu, S. Hariri, and M.-y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[24] A. Darte, Y. Robert, and F. Vivien, *Scheduling and automatic Parallelization*. Springer Science & Business Media, 2012.

[25] T. L. Adam, K. M. Chandy, and J. Dickson, “A comparison of list schedules for parallel processing systems,” *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[26] E. G. Coffman and R. L. Graham, “Optimal scheduling for two-processor systems,” *Acta informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[27] R. L. Graham, “Bounds on multiprocessing timing anomalies,” *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[28] H. Kasahara and S. Narita, “Practical multiprocessor scheduling algorithms for efficient parallel processing,” *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 1023–1029, 1984.

[29] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, “Scheduling precedence graphs in systems with interprocessor communication times,” *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[30] B. Kruatrachue and T. Lewis, “Grain size determination for parallel processing,” *IEEE software*, vol. 5, no. 1, pp. 23–32, 1988.

[31] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger, “Multiprocessor scheduling with interprocessor communication delays,” *Operations Research Letters*, vol. 7, no. 3, pp. 141–147, 1988.

[32] Z. Liu, “A note on graham’s bound,” *Information Processing Letters*, vol. 36, no. 1, pp. 1–5, 1990.

[33] G. C. Sih and E. A. Lee, “A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures,” *IEEE transactions on Parallel and Distributed systems*, vol. 4, no. 2, pp. 175–187, 1993.

[34] M.-Y. Wu and D. D. Gajski, “Hypertool: A programming aid for message-passing systems,” *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[35] H. El-Rewini and T. G. Lewis, “Scheduling parallel program tasks onto arbitrary target machines,” *Journal of parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[36] Y. K. Kwok and I. Ahmad, “Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors,” *Cluster Computing*, vol. 3, no. 2, pp. 113–124, 2000.

[37] S. Kim and J. C. Browne, "General approach to mapping of parallel computations upon multiprocessor architectures," in *Proceedings of the International Conference on Parallel Processing*, vol. 3, 1988, pp. 1–8.

[38] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[39] J.-C. Liou and M. A. Palis, "An efficient task clustering heuristic for scheduling dags on multiprocessors," in *Workshop on Resource Management, Symposium on Parallel and Distributed Processing*, 1996, pp. 152–156.

[40] I. Ahmad and Y.-K. K. Y.-K. Kwok, "A new approach to scheduling parallel programs using task duplication," in *1994 Internatonal Conference on Parallel Processing Vol. 2*, IEEE, vol. 2, 1994, pp. 47–51.

[41] Y.-C. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, IEEE, 1992, pp. 512–521.

[42] G.-L. Park, B. Shirazi, and J. Marquis, "Dfrn: A new approach for duplication based scheduling for distributed memory multiprocessor systems," in *Proceedings 11th International Parallel Processing Symposium*, IEEE, 1997, pp. 157–166.

[43] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," in *5th IEEE heterogeneous computing workshop (HCW'96)*, 1996, pp. 86–97.

[44] L. Wang, H. J. Siegel, and V. P. Roychowdhury, "A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments," in *Proc. Heterogeneous Computing Workshop*, 1996, pp. 72–85.

[45] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed systems*, vol. 5, no. 2, pp. 113–120, 1994.

[46] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," in *5th Heterogeneous Computing Workshop (HCW'96)*, 1996, pp. 98–117.

[47] L Tao, B Narahari, and Y. Zhao, "Heuristics for mapping parallel computations to parallel architectures," in *Proceedings. Workshop on Heterogeneous Processing,*, IEEE, 1993, pp. 36–41.

[48] M.-Y. Wu, W. Shu, and J. Gu, "Local search for dag scheduling and task assignment," in *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No. 97TB100162)*, IEEE, 1997, pp. 174–180.

[49] Y.-K. Kwok, I. Ahmad, and J. Gu, "Fast: A low-complexity algorithm for efficient scheduling of dags on parallel processors," in *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, IEEE, vol. 2, 1996, pp. 150–157.

[50] P. Pop, P. Eles, Z. Peng, and T. Pop, "Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 8, pp. 793–811, 2004.

[51] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Design & Test of Computers*, vol. 15, no. 2, pp. 45–54, 1998.

[52] G De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, 1997.

[53] J. Staunstrup and W. Wolf, *Hardware/software co-design: principles and practice*. Springer Science & Business Media, 2013.

[54] A. Buiga, "Investigating the role of mqb platform in volkswagen group's strategy and automobile industry," *International Journal of Academic Research in Business and Social Sciences*, vol. 2, no. 9, pp. 391–399, 2012.

[55] A. Volkswagen, *New group strategy adopted: Volkswagen group to become a world-leading provider of sustainable mobility*, 2016.

[56] R. Hähnle and R. Muschevici, "Towards incremental validation of railway systems," in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2016, pp. 433–446.

[57] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, no. 3, pp. 8–18, 2003.

[58] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[59] T. Schwarzer, J. Falk, M. Glaß, J. Teich, C. Zebelein, and C. Haubelt, "Throughput-optimizing compilation of dataflow applications for multi-cores using quasi-static scheduling," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, ACM, 2015, pp. 68–75.

[60] Y. Wen, H. Xu, and J. Yang, "A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system," *Information Sciences*, vol. 181, no. 3, pp. 567–581, 2011.

[61] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2013.

[62] "Real-time scheduling," in *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston, MA: Springer US, 1997, pp. 227–243, ISBN: 978-0-306-47055-4.

[63] K. Singh, M. Alam, and S. K. Sharma, "A survey of static scheduling algorithm for distributed computing system," *International Journal of Computer Applications*, vol. 129, no. 2, pp. 25–30, 2015.

[64] R. Rajak, "Comparison of bounded number of processors (bnp) class of scheduling algorithms based on matrices," *Computer Sciences and Telecommunications*, no. 3, pp. 35–44, 2012.

[65] P. Cichowski and J. Keller, "Efficient and fault-tolerant static scheduling for grids," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, 2013, pp. 1439–1448.

[66] S. M. Shatz, J.-P. Wang, and M. Goto, "Task allocation for maximizing reliability of distributed computer systems," *IEEE Transactions on Computers*, no. 9, pp. 1156–1168, 1992.

[67] X. Qin, H. Jiang, and D. R. Swanson, "An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems," in *Proceedings International Conference on Parallel Processing*, IEEE, 2002, pp. 360–368.

[68] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of openmp task systems with tied tasks," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2017, pp. 92–103.

[69] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng, "Comparative evaluation of the robustness of dag scheduling heuristics," in *Grid Computing*, Springer, 2008, pp. 73–84.

[70] J.-Y. Colin and P. Chrétienne, "Cpm scheduling with small communication delays and task duplication," *Operations Research*, vol. 39, no. 4, pp. 680–684, 1991.

[71] N. Jazdi, "Cyber physical systems in the context of industry 4.0," in *2014 IEEE international conference on automation, quality and testing, robotics*, IEEE, 2014, pp. 1–4.

[72] G. Xie, R. Li, and K. Li, "Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems," *Journal of Parallel and Distributed Computing*, vol. 83, pp. 1–12, 2015.

[73] X.-M. Zhang and Q.-L. Han, "Event-triggered dynamic output feedback control for networked control systems," *IET Control Theory & Applications*, vol. 8, no. 4, pp. 226–234, 2014.

[74] A. Albert *et al.*, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *Embedded world*, vol. 2004, pp. 235–252, 2004.

[75] R. Bosch *et al.*, "Can specification version 2.0," *Rober Bousch GmbH, Postfach*, vol. 300240, p. 72, 1991.

[76] L. Echelon, *The lontalk protocol specification*, 2003.

[77] J. Berwanger, M. Peller, and R. Griessbach, "Byteflight–a new high-performance data bus system for safety-related applications," *BMW AG*, 2000.

[78] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[79] P. Miner, "Analysis of the spider fault-tolerance protocols," in *Proceedings of the 5th NASA Langley Formal Methods Workshop*, 2000.

[80] K. Hoyme and K. Driscoll, "Safebus," in *[1992] Proceedings IEEE/AIAA 11th Digital Avionics Systems Conference*, IEEE, 1992, pp. 68–73.

[81] W. Steiner, "Ttethernet specification," *TTTech Computertechnik AG, Nov*, vol. 39, p. 40, 2008.

[82] I. ISO, "11898-4-road vehicles-controller area network (can)-part 4: Time-triggered communication," *International Standard Organization*, pp. 11 898–4, 2000.

[83] J. Dvořák and Z. Hanzálek, "Multi-variant scheduling of critical time-triggered communication in incremental development process: Application to flexray," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 1, pp. 155–169, 2018.

[84] W. Steiner, "An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks," in *2010 31st IEEE Real-Time Systems Symposium*, IEEE, 2010, pp. 375–384.

[85] ——, "Synthesis of static communication schedules for mixed-criticality systems," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, IEEE, 2011, pp. 11–18.

[86] D. Tămaş-Selicean, P. Pop, and W. Steiner, "Design optimization of ttethernet-based distributed real-time systems," *Real-Time Systems*, vol. 51, no. 1, pp. 1–35, 2015.

[87] 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*, ACM, 2016, pp. 183–192.

[88] F. Pozo, G. Rodriguez-Navas, and H. Hansson, "Methods for large-scale time-triggered network scheduling," 2019.

[89] Z. Zheng, F. He, and Y. Xiong, "The research of scheduling algorithm for time-triggered ethernet based on path-hop," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, IEEE, 2016, pp. 1–6.

[90] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt, "Flexray schedule optimization of the static segment," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ACM, 2009, pp. 363–372.

[91] K. Schmidt and E. G. Schmidt, "Message scheduling for the flexray protocol: The static segment," *IEEE transactions on vehicular technology*, vol. 58, no. 5, pp. 2170–2179, 2008.

[92] Z. Hanzálek, D. Beneš, and D. Waraus, "Time constrained flexray static segment scheduling," in *Proc. 10th Int. Workshop Real-Time Netw.*, 2011, pp. 23–28.

[93] M. Kang, K. Park, and M.-K. Jeong, "Frame packing for minimizing the bandwidth consumption of the flexray static segment," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 9, pp. 4001–4008, 2012.

[94] M. Grenier, L. Havet, and N. Navet, "Configuring the communication on flexray-the case of the static segment," 2008.

[95] K. Schmidt and E. G. Schmidt, "Optimal message scheduling for the static segment of flexray," in *2010 IEEE 72nd Vehicular Technology Conference-Fall*, IEEE, 2010, pp. 1–5.

[96] R Zhao, G. Qin, and J. Liu, "Optimal scheduling of the flexray static segment based on two-dimensional bin-packing algorithm," *International Journal of Automotive Technology*, vol. 17, no. 4, pp. 703–715, 2016.

[97] R. Zhao, G.-h. Qin, and J.-q. Liu, "A rectangle bin packing optimization approach to the signal scheduling problem in the flexray static segment," *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 4, pp. 375–388, 2016.

[98] R Zhao, G. Qin, H. Chen, J Qin, and J Yan, "Security-aware scheduling for flexray-based real-time automotive systems," *Mathematical Problems in Engineering*, vol. 2019, 2019.

[99] B. Fateh and M. Govindarasu, "Joint scheduling of tasks and messages for energy minimization in interference-aware real-time sensor networks," *IEEE transactions on mobile computing*, vol. 14, no. 1, pp. 86–98, 2013.

[100] L. Yang, W. Liu, W. Jiang, M. Li, J. Yi, and E. H.-M. Sha, "Application mapping and scheduling for network-on-chip-based multiprocessor system-on-chip with fine-grain communication optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3027–3040, 2016.

[101] H. Zeng, M. Di Natale, A. Ghosal, and A. Sangiovanni-Vincentelli, "Schedule optimization of time-triggered systems communicating over the flexray static segment," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 1, pp. 1–17, 2010.

[102] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *17th Asia and South Pacific design automation conference*, IEEE, 2012, pp. 665–670.

[103] P. Pop, M. L. Raagaard, S. S. Craciunas, and W. Steiner, "Design optimisation of cyber-physical distributed systems using ieee time-sensitive networks," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 86–94, 2016.

[104] M. Hu, J. Luo, Y. Wang, M. Lukasiewycz, and Z. Zeng, "Holistic scheduling of real-time applications in time-triggered in-vehicle networks," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1817–1828, 2014.

[105] M. Pahlevan and R. Obermaisser, "Genetic algorithm for scheduling time-triggered traffic in time-sensitive networks," in *2018 IEEE 23rd International Conference*

*on Emerging Technologies and Factory Automation (ETFA)*, IEEE, vol. 1, 2018, pp. 337–344.

[106] B. Cheng, L. Wang, W. Liu, and L. Zeng, "Scheduling algorithm based on time-trigger bus," in *2016 IEEE 13th International Conference on Signal Processing (ICSP)*, IEEE, 2016, pp. 1787–1790.

[107] R. Bajaj and D. P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 107–118, 2004.

[108] S. Baskiyar and P. C. SaiRanga, "Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length," in *2003 International Conference on Parallel Processing Workshops, 2003. Proceedings.*, IEEE, 2003, pp. 97–103.

[109] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, IEEE, 2004, p. 107.

[110] X. Tang, K. Li, G. Liao, K. Fang, and F. Wu, "A stochastic scheduling algorithm for precedence constrained tasks on grid," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1083–1091, 2011.

[111] M. A. Khan, "Scheduling for heterogeneous systems using constrained critical paths," *Parallel Computing*, vol. 38, no. 4, pp. 175–193, 2012.

[112] O. Sinnen and L. Sousa, "List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures," *Parallel Computing*, vol. 30, no. 1, pp. 81–101, 2004.

[113] Y.-K. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, IEEE, 1998, pp. 531–537.

[114] M. K. Bhatti, I. Oz, S. Amin, M. Mushtaq, U. Farooq, K. Popov, and M. Brorsson, "Locality-aware task scheduling for homogeneous parallel computing systems," *Computing*, vol. 100, no. 6, pp. 557–595, 2018.

[115] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on parallel and distributed systems*, vol. 7, no. 5, pp. 506–521, 1996.

[116] O. Sinnen, A. To, and M. Kaur, “Contention-aware scheduling with task duplication,” *Journal of Parallel and Distributed Computing*, vol. 71, no. 1, pp. 77–86, 2011.

[117] M. Pahlevan, N. Tabassam, and R. Obermaisser, “Heuristic list scheduler for time triggered traffic in time sensitive networks,” *ACM Sigbed Review*, vol. 16, no. 1, pp. 15–20, 2019.

[118] R. Obermaisser and A. Murshed, “Incremental, distributed, and concurrent scheduling in systems-of-systems with real-time requirements,” in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, IEEE, 2015, pp. 1918–1927.

[119] N. G. Nayak, F. Dürr, and K. Rothermel, “Incremental flow scheduling and routing in time-sensitive software-defined networks,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, 2017.

[120] J. Dvorak and Z. Hanzalek, “Multi-variant time constrained flexray static segment scheduling,” in *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, IEEE, 2014, pp. 1–8.

[121] F. Sagstetter, P. Waszecki, S. Steinhorst, M. Lukasiewycz, and S. Chakraborty, “Multischedule synthesis for variant management in automotive time-triggered systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 637–650, 2015.

[122] L. F. Gonçalves, J. L. Bosa, T. R. Balen, M. S. Lubaszewski, E. L. Schneider, and R. V. Henriques, “Fault detection, diagnosis and prediction in electrical valves using self-organizing maps,” *Journal of Electronic Testing*, vol. 27, no. 4, pp. 551–564, 2011.

[123] A. Le Mortellec, J. Clarhaut, Y. Sallez, T. Berger, and D. Trentesaux, “Embedded holonic fault diagnosis of complex transportation systems,” *Engineering Applications of Artificial Intelligence*, vol. 26, no. 1, pp. 227–240, 2013.

[124] J. Gertler, *Fault detection and diagnosis in engineering systems*. Routledge, 2017.

[125] ——, “Fault detection and diagnosis,” *Encyclopedia of Systems and Control*, pp. 417–422, 2015.

[126] X. Dai and Z. Gao, “From model, signal to knowledge: A data-driven perspective of fault detection and diagnosis,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2226–2238, 2013.

[127] J. Ma and J. Jiang, “Applications of fault detection and diagnosis methods in nuclear power plants: A review,” *Progress in nuclear energy*, vol. 53, no. 3, pp. 255–266, 2011.

[128] O. Kermia and Y. Sorel, “A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor,” in *Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS’07*, 2007.

[129] A. Rădulescu and A. J. Van Gemund, “On the complexity of list scheduling algorithms for distributed-memory systems,” in *Proceedings of the 13th international conference on Supercomputing*, ACM, 1999, pp. 68–75.

[130] 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*, vol. 68, no. 4, pp. 399–409, 2008.

[131] H. Arabnejad and J. G. Barbosa, “List scheduling algorithm for heterogeneous systems by an optimistic cost table,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.

[132] R. Obermaisser, *Time-triggered communication*. CRC Press, 2018.

[133] J. Leskovec and R. Sosič, “Snap: A general-purpose network analysis and graph-mining library,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[134] U. D. of Energy. (2008). “Energy efficiency trends in residential and commercial buildings,” (visited on 09/19/2019).

[135] Y. Kim, T. Schmid, M. B. Srivastava, and Y. Wang, “Challenges in resource monitoring for residential spaces,” in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, ACM, 2009, pp. 1–6.

[136] M. Dey, S. P. Rana, and S. Dudley, “Smart building creation in large scale hvac environments through automated fault detection and diagnosis,” *Future Generation Computer Systems*, 2018.

[137] J. Weimer, S. A. Ahmadi, J. Araujo, F. M. Mele, D. Papale, I. Shames, H. Sandberg, and K. H. Johansson, “Active actuator fault detection and diagnostics in hvac systems,” in *Proceedings of the fourth ACM workshop on embedded sensing systems for energy-efficiency in buildings*, ACM, 2012, pp. 107–114.

[138] S. Katipamula and M. R. Brambley, “Methods for fault detection, diagnostics, and prognostics for building systems—a review, part i,” *Hvac&R Research*, vol. 11, no. 1, pp. 3–25, 2005.

[139] N. Djuric and V. Novakovic, "Review of possibilities and necessities for building lifetime commissioning," *Renewable and Sustainable Energy Reviews*, vol. 13, no. 2, pp. 486–492, 2009.

[140] N. Fernandez, M. R. Brambley, S. Katipamula, H. Cho, J. K. Goddard, and L. H. Dinh, "Self-correcting hvac controls project final report," Pacific Northwest National Lab.(PNNL), Richland, WA (United States), Tech. Rep., 2010.

[141] L. Jagemar, D. Olsson, and F Schmidt, "The epbd and continuous commissioning," *Project Report, Building EQ, EIE/06/038/SI2*, vol. 448300, 2007.

[142] S. W. Kramer and P. Fleck, "Maintaining building function during a fire event: Analysis of hospital fire and smoke control systems," 2018.

[143] D. Sloo, N. U. Webb, E. J. Fisher, Y. Matsuoka, A. Fadell, and M. Rogers, *Smart-home control system providing hvac system dependent responses to hazard detection events*, US Patent 9,905,122, 2018.

[144] V. L. Erickson, M. Á. Carreira-Perpiñán, and A. E. Cerpa, "Observe: Occupancy-based system for efficient reduction of hvac energy," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IEEE, 2011, pp. 258–269.

[145] F. Oldewurtel, A. Parisio, C. N. Jones, D. Gyalistras, M. Gwerder, V. Stauch, B. Lehmann, and M. Morari, "Use of model predictive control and weather forecasts for energy efficient building climate control," *Energy and Buildings*, vol. 45, pp. 15–27, 2012.

[146] A. Afram and F. Janabi-Sharifi, "Theory and applications of hvac control systems–a review of model predictive control (mpc)," *Building and Environment*, vol. 72, pp. 343–355, 2014.

[147] Z. Du, B. Fan, X. Jin, and J. Chi, "Fault detection and diagnosis for buildings and hvac systems using combined neural networks and subtractive clustering analysis," *Building and Environment*, vol. 73, pp. 1–11, 2014.

[148] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757–3767, 2015.

[149] D. Li, Y. Zhou, G. Hu, and C. J. Spanos, "Fault detection and diagnosis for building cooling system with a tree-structured learning method," *Energy and Buildings*, vol. 127, pp. 540–551, 2016.

[150] B. Sun, P. B. Luh, Q.-S. Jia, Z. O'Neill, and F. Song, "Building energy doctors: An spc and kalman filter-based method for system-level fault detection in hvac systems," *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 1, pp. 215–229, 2013.

[151] H. Wang, Y. Chen, C. W. Chan, and J. Qin, "An online fault diagnosis tool of vav terminals for building management and control systems," *Automation in Construction*, vol. 22, pp. 203–211, 2012.

[152] Y. Zhu, X. Jin, and Z. Du, "Fault diagnosis for sensors in air handling unit based on neural network pre-processed by wavelet and fractal," *Energy and buildings*, vol. 44, pp. 7–16, 2012.

[153] B. Fan, Z. Du, X. Jin, X. Yang, and Y. Guo, "A hybrid fdd strategy for local system of ahu based on artificial neural network and wavelet analysis," *Building and environment*, vol. 45, no. 12, pp. 2698–2708, 2010.

[154] H. Han, Z. Cao, B. Gu, and N. Ren, "Pca-svm-based automated fault detection and diagnosis (afdd) for vapor-compression refrigeration systems," *HVAC&R Research*, vol. 16, no. 3, pp. 295–313, 2010.

[155] K.-Y. Chen, L.-S. Chen, M.-C. Chen, and C.-L. Lee, "Using svm based method for equipment fault detection in a thermal power plant," *Computers in industry*, vol. 62, no. 1, pp. 42–50, 2011.

[156] Y. Hu, H. Chen, J. Xie, X. Yang, and C. Zhou, "Chiller sensor fault detection using a self-adaptive principal component analysis method," *Energy and buildings*, vol. 54, pp. 252–258, 2012.

[157] J. Yun and K.-H. Won, "Building environment analysis based on temperature and humidity for smart energy systems," *Sensors*, vol. 12, no. 10, pp. 13 458–13 470, 2012.

[158] D. Dietrich, D. Bruckner, G. Zucker, and P. Palensky, "Communication and computation in buildings: A short introduction and overview," *IEEE transactions on industrial electronics*, vol. 57, no. 11, pp. 3577–3584, 2010.

[159] S. Yin, S. X. Ding, X. Xie, and H. Luo, "A review on basic data-driven approaches for industrial process monitoring," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 11, pp. 6418–6428, 2014.

[160] A. Behravan, N. Tabassam, O. Al-Najjar, and R. Obermaisser, "Composability modeling for the use case of demand-controlled ventilation and heating system," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, IEEE, 2019, pp. 1998–2003.